Universitat de Girona
Escola Politècnica Superior

FINAL DEGREE PROJECT

Title:

WAVA: Design and development of a cinematic 2D puzzle-platformer game

Author:
Oriol Viu Duran

Studies: **Design and Development of Video Games**
Document: **Thesis**

Tutor: **Gustavo Patow**
Department: **Computer Science, Applied Mathematics and Statistics**
Area: **Computer Languages and Systems**

Convocation (month/year): **June 2022**

Acknowledgements

To Gustavo Patow, tutor of this final degree project, for his guidance, enthusiasm and for loving this project as much as I do.

To my precious family, for their outpouring love and support and for always believing in me.

To my friends, for their motivation and valuable input.

Index

Figures

Tables

1. Introduction

1.1. Introduction

In recent years, the video game industry has experienced a resurgence of cinematic 2D puzzle-platformer games. These games are often characterised by their short playtime, simple mechanics and a powerful ambience. In spite of the absence of dialogue or text, these types of games tell a complete and understandable story with a profound message. Some notable titles may include *Another World* (1991) and, more recently, *Limbo* (2010), *Inside* (2016) or *Gris* (2018).

Despite not being the most popular game genre, there is a very dedicated audience that enjoys each project in spite of their simplicity in gameplay. However, what this type of games lack in complexity, they make it up with very unique personalities, which make them more accessible for more unexperienced players.

The game that will be developed for this project is called WAVA (see Figure 1.1). It is a game that explores a very common and relatable topic: finding our place, whether it is physically or mentally. Although the game's scenery is somewhat realistic and plausible, the game itself is very conceptual.

The game loop itself helps to narrate the story and transmit the main message. This loop consists in reaching the end of the level by overcoming certain setbacks and getting knowledge from them, thus enabling us to go on further. In addition, by using different elements of design such as composition, weather or colour, the game emphasises the main character's feelings.

Figure 1.1. WAVA Logo

1.2. Motivations

The resurgence of 2D cinematic games was the main motivator for the creation of this project. After observing the commercial and critical success of similar titles, we decided to elaborate a product that can distinguish itself from others with very unique aesthetics and a message that many players, no matter their level of expertise, can empathise with.

On a more personal note, the main motives behind the development of this project are:

- Develop a cinematic video game from start to finish and work on different aspects of game design and development.
- Create an easy and accessible product with a relatable message that intends to help the player personally.
- Put into action all the knowledge gained along our time at university.
- Use this project as an official beginning to my own portfolio, thus aiding me in entering the video game industry.

1.3. Purpose and objectives of the project

The main purpose of this project is to fully develop a cinematic 2D game with puzzle and platforming elements. In the same manner as other similar games, this project intends to transmit a profound message, more precisely about overcoming our own problems and finding our place, with its design and unique aesthetic.

More specifically, the objectives set for this project include, but are not limited to:

- Deepening the knowledge already acquired about the game engine Unity and the programming language C#.
- Implement game mechanics that reinforce the story and the message transmitted by the game.
- Design an atmospheric environment that is captivating, unique and reinforces the game's predominant topics.
- Create different animations for the character and other assets that give the game a certain smoothness and fluidity.

1.4. Task distribution

Video games are often developed by a multidisciplinary team, assigning each task to different members based on their profile and area of expertise. Nonetheless, this project will be fully developed by a single person who will work on all the various tasks.

The chart at Table 1.1 shows a percentage distribution for every aspect of the development of the project based on their overall values and the amount of effort dedicated to each one.

Aesthetics	25%
Narrative	25%
Mechanics	25%
Technology	25%

Table 1.1. Self-assessment chart

The distribution that describes the development of this project in the fairest way possible is an even allocation across the four aspects.

Due to the visual nature of the game, designing and creating an atmosphere that transmits the game's message and distinguishes itself from other projects while still being captivating and coherent, is a difficult task in itself. The aesthetics of the game include several aspects, such as the world and environment, the main character/s and other elements. It also involves all the various animations and the game's interface.

The story needs to be written and structured in a way that advances the overall narrative while also not relying on any messages written on screen. It is very important to find the perfect plot points that are visual and are able to communicate the game's overall message.

As stated before, the game loop and the mechanics themselves are designed in a way that reinforce and explain the game's message. In addition, the composition of the world directly represents the main character's feelings. Therefore, it is extremely important to lay all these different aspects out properly before implementing anything else.

Last but not least, all the game mechanics we have previously designed need to be programmed and implemented. Furthermore, it is needed to build the world inside the game engine. We need to ensure that the game feels complete and flows throughout.

2. Viability study

In this section, a thorough study of the viability of the project will be made in order to determine if we are to proceed with it. Several aspects will be taken into account; first, we will focus on our own resources and capabilities; afterwards, we will examine the market to understand our opportunities and estimate a possible business model; lastly, we will define our target audience.

2.1. Necessary resources

The project cannot be viable without analysing our own resources beforehand. These include the human, economic and technical resources, for both software and hardware.

2.1.1. Technical resources

The entirety of the game will be developed using free or open source software. As for the hardware, only a mid-end to high-end computer is needed. The resources used for this project have been the following:

Personal computer

All of the game's development has been done using a sole laptop, model MSI PE72 8RC (see Figure 2.1). The main traits and specifications of this machine are:

- 8th Gen. Intel® Core™ i7 (Processor)
- Windows 10 Home 64 bit (Operating System)
- NVIDIA GeForce® GTX 1050 (Graphics card)
- 8 GB (RAM Memory)

Figure 2.1. MSI Logo

Unity

The game engine used for this project has been Unity, developed by the company Unity Technologies. Specifically, the version used has been *Unity 2020.3.11.f1* (see Figure 2.2).

Unity is widely known in the video game industry for its accessibility, since it is free, relatively easy to use and can support a variety of platforms, which is very beneficial for independent game developers. It can be used to create both three-dimensional and two-dimensional games, the latter corresponding to our project. For 2D games, Unity allows importation of sprites and an advanced 2D world renderer. In addition, it is also very well documented and provides the developer with a wide array of guides and examples for every function and aspect of its environment.

Figure 2.2. Unity Logo

Microsoft Visual Studio

Programming for the game has been done in the integrated development environment (IDE) Microsoft Visual Studio, which is also connected directly to Unity. Specifically, the version used has been *Visual Studio 2019* (see Figure 2.3).

The programming language used has been the object-oriented language C Sharp (C#), designed by Anders Hejlsberg (see Figure 2.4). The game engine Unity offers a primary scripting application programming interface (API) in C# using Mono, which is a free and open-source .NET Framework-compatible software framework.

Figure 2.3. Microsoft Visual Studio Logo

Figure 2.4. C Sharp Logo

Inkscape

All the visual and artistic aspects of the game have been made using the free and open-source vector graphics editor Inkscape (see Figure 2.5).

Inkscape lets the user create a wide variety of shapes and then manipulate them with different transformations such as moving, rotating, scaling and skewing. It can render both primitive vector shapes and text, which made it possible to even create aspects of the user interface (UI). Inkscape was also used to do the different animations for the game, via spritesheet animation.
This software allows the importation of different types of files, such as images. It also allows exporting Scalable Vector Graphics (SVG) into Portable Network Graphics (PNG), which was utilised to create the different sprites for the game.

Figure 2.5 Inkscape Logo

GitHub Desktop

The version control was done with Unity Collaborate at first, which is a service integrated in Unity. Nonetheless, in the midst of the game's development, Unity Collaborate merged into the platform Plastic SCM. Due to the convenience of having this already integrated into the game engine, it was attempted to port the project onto the new platform. Unfortunately, it was not possible as the version of Unity used was not supported yet by Plastic SCM. From there on, the version control was done via GitHub and GitHub Desktop (see Figure 2.6).

GitHub is a commonly used Internet hosting provider for software development and version control. It uses Git, a software for tracking changes in a set of files. Although this platform is mostly used for working collaboratively, within the frame of this project, which has been realised by a single developer, it has served solely as a version control and back up method.

Figure 2.6. GitHub Desktop Logo

HacknPlan

The project management and task planning was done with HacknPlan (see Figure 2.7). It is an online tool created by Chris Estevez specifically designed for game development. The app allows the user to create a board where all the different tasks are visible and can be organised depending on their current state: Planned, In progress, Testing and Completed. The tasks can be of one out of eight categories: Programming, Art, Design, Writing, Marketing, Sound, Ideas and Bug.

HacknPlan has many other functions and provides the user with loads of personalisation options, but they were not needed for the organisation of this project.

Figure 2.7. HacknPlan Logo

Diagrams.net

The diagrams and other graphics that appear in this document have been made with the free and open source online tool Diagrams.net, previously known as Draw.io (see Figure 2.8).

Figure 2.8. Diagrams.net Logo

2.1.2. Human resources

Video games are usually developed by teams which, despite varying in size from one project to another, tend to include different people with very different profiles and areas of knowledge. These profiles include programmers, designers and artists, among others.

However, for this project, all the different aspects of the game will be developed by a single person.

2.1.3. Economic resources

Since all the software used to carry out this project is free to use and the computer used was already owned prior to the start of the development, there is no real economic cost for this project.

Anyhow, we could calculate an approximate total cost by adding the salaries of hypothetical members for a team, each one with the same personal computer used for this project. The number of hypothetical members is three, considering one of them would focus on the programming, the second one on the designing and the last one on all the artistic aspects.

According to the webpage Glassdoor, the average yearly salary for a game developer in Spain as in 2022, is at approximately 35,000 EUR. Therefore, if we consider the development of the game to have taken a full fiscal year, we get the results shown in Table 2.1:

ITEM	COST
3 X MSI PE72 8RC	~ 3,000 EUR
Programmer	~ 35,000 EUR
Designer	~ 35,000 EUR
Artist	~ 35,000 EUR
TOTAL	**~ 108,000 EUR**

Table 2.1. Hypothetical economic cost

It is also worth noting that developing a game with Unity is free as long as the gross annual revenue for the previous fiscal year does not surpass 100,000 USD.

2.2. Market analysis

After a deep analysis of all our resources, it can be stated that, from this perspective, the project is viable. The next step is to look at the state-of-the-art and detect how our project can stand out from other cinematic puzzle-platformers, specifically, those with a two-dimensional view.

2.2.1. Search methodology

The search engine Google was the one used to find these similar games. The keywords chosen for the search, and the ones that returned the most significant results, were "*2d cinematic platformer games*". Other variations of these keywords were tested, but only to a similar or lesser outcome.

2.2.2. Similar games

From the results obtained, only the most notable titles were selected. The ones that were chosen were those we were already familiar with and had played, so as to be able to make a better comparison.

***Limbo* (2010)**

Limbo is a 2D puzzle-platform video game developed by the independent studio Playdead, based in Denmark. It is well known for its use of sound and colour or, in fact, its lack of it, since the game is in grayscale. Another notable aspect is its minimal story, as there is no explicit plot told along the game's runtime, leaving the audience to form their own interpretation (see Figure 2.9).

Figure 2.9. Image from *Limbo*

***Inside* (2016)**

Inside is a puzzle-platform game also developed by the Danish game studio Playdead. It has many similarities to its predecessor, *Limbo*, such as its minimal colour, untold story, and puzzle-platforming mechanics. Even though the character's movement is still two-dimensional, the game itself is three-dimensional, which allows the camera to move more cinematically (see Figure 2.10).

Figure 2.10. Image from *Inside*

***Gris* (2018)**

Gris is a cinematic puzzle-platformer game developed by the Catalan game company Nomada Studio and published by Devolver Digital. It is a very unique game, due to its gorgeous and distinct art style, created by the artist Conrad Roset (see Figure 2.11).

Figure 2.11. Image from *Gris*

***Another World* (1991)**

Another World is a cinematic platformer game designed by Éric Chahi and published by the French company Delphine Software. It is one of the first instances of this category of games and, perhaps, the most iconic. Its captivating art style and fluid animations were a huge achievement for its time (see Figure 2.12).

Since the game was released a long time ago, for comparisons and other important data, we will use the *20th Anniversary Edition*, released in 2013.

Figure 2.12. Image from *Another World*

2.2.3. Conclusions for the state of the art

Since the search that was done to find possible competitors was very specific, we see lots of similarities between the games, from minimalistic art styles, to a complete visual narrative, to a similar gameplay. On the Table 2.2 we can see the comparison with our competitors.

Title	Movement	Art	Narrative	Duration	Price (EUR)	Engine
Limbo	2D	2D	Visual	3h	9,99	Own
Inside	2D	3D	Visual	3h	19,99	Unity
Gris	2D	2D	Visual	3h	16,99	Unity
Another World	2D	2D	Visual	30m	9,99	Own
WAVA	**2D**	**2D**	**Visual**	**30m**	**FREE**	**Unity**

Table 2.2. Comparison with other similar games

Even though the games are very similar to one another, each one of them has had great commercial success and very positive critical acclaim. Cinematic puzzle-platformers are not one of the most common types of games released in this day and age; however, they are very well received and have a very dedicated audience.

Despite the common and simple gameplay mechanics, each game has a unique narrative, art style and overall atmosphere that distinguishes itself from others. This is, of course, also the case for this project, WAVA.

2.2.4. Business model

One of the most common complaints for our competitors is their high price despite their short playtime. As we can see, the prices range from 10 EUR to 20 EUR in the platform Steam.

As for our project, we believe the best course of action is to publish the game for free, so as to make it function as some sort of “presentation letter” for ourselves. By doing this, a wider audience can be reached and, if the game is well received, this audience might be on board with future projects, some of which may be monetised. One notable instance of this business model is the Dutch company Total Mayhem Games.

2.3. Target audience

Last but not least, in order to see the full viability of this project, we must take into account all the aspects of our game and define our target consumers. Richard Bartle released in 1996 the *Bartle's taxonomy of player types*, which classifies all players into the following four groups (see Figure 2.13):

- **Killers:** These types of players are very competitive; they would rather fight against other players than artificially intelligent entities.
- **Achievers:** These types of players like completion; they want to gain points, levels and other concrete measurements of success in a game.
- **Socializers:** These types of players enjoy interaction with other players; they also socialise outside of the game, in forums and social media.
- **Explorers:** These types of players long for immersion; they like to discover areas and wander through the game's world at their own pace.

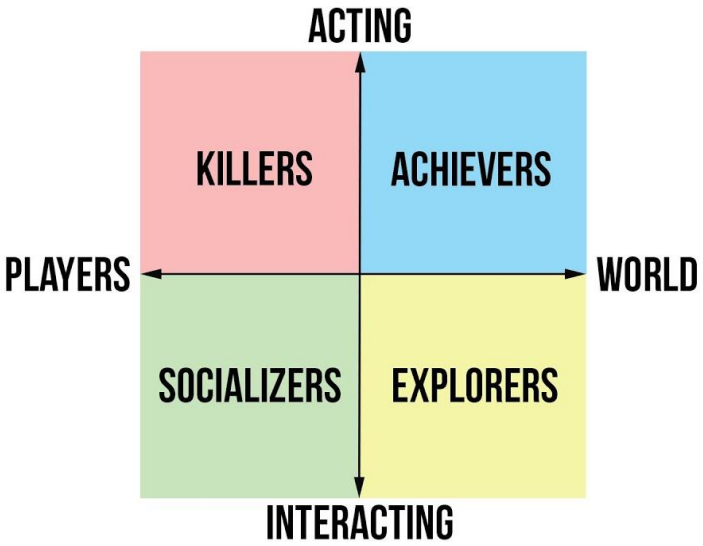

Figure 2.13. Bartle's taxonomy of player types

Due to our game's calm nature and overall non-violent mechanics, we think the profile of player that best adapts to it is the **Explorer**. WAVA is a very visual game, with a very simple yet enchanting atmosphere and, to our belief, the game is enjoyed the most when consumed slowly. Moreover, we believe that the sharing aspect of a **Socializer** would also be hugely beneficial to our project. The game wants to convey a deep message that, in a way, is open to interpretation to every single player. Therefore, it would be terrific to have players discussing online their afterthoughts.

Lastly, it is very important to select the age range we want to cater. WAVA is a slow paced game, and one that requires time and patience, which we think is more adequate for a more adult audience. The controls are fairly simple and the game loop follows an easy to understand pattern, so it can be enjoyed by anybody, no matter their level of expertise.

3. Planning

In this section we will define the work strategy and work plan that will enable us to reach our goals and develop the proposed project. Due to the fact that the game is fully developed by a single person and we don't depend on other team members or specific deadlines, we are allowed a little liberty when deciding on what aspect of the game we are focusing our attention on.

3.1. Work methodology

Video game development can generally be divided into three blocks: the design of the project, the implementation of the designed aspects and, lastly, the testing of the previous stages. Although it may seem that these blocks follow a sequential order, they are, in most cases, an iterative sequence. This iteration happens due to the fact that sometimes what we design does not work when put into practice, forcing ourselves to rethink parts of the process.

Design is unavoidably the first stage of the process, since it is the part where we figure out how we want the game to behave, look and feel. It includes many different aspects, such as the general game design, story design, art design and even level design.

Implementation can be summed up as the materialisation of the designs. This includes both the more technological parts of the game, such as the programming of the mechanics; and the more artistic aspects, such as the creation and animation of all the different assets.

Testing happens along all of the game's development. Once our designs have been implemented, we need to try them out to see if they work in the "real world".

Overall organisation and planning will be achieved by meeting with the project's tutor every two weeks and by keeping the project management tool HacknPlan up to date according to this document.

3.2. Work plan

In order to develop the game fluently, all the different tasks that are to be done must be defined and mapped out. These tasks will be distributed into six blocks: design, technology, narrative, art, testing and documentation. For an easier understanding, it is represented in the following diagram (see Figure 3.1):

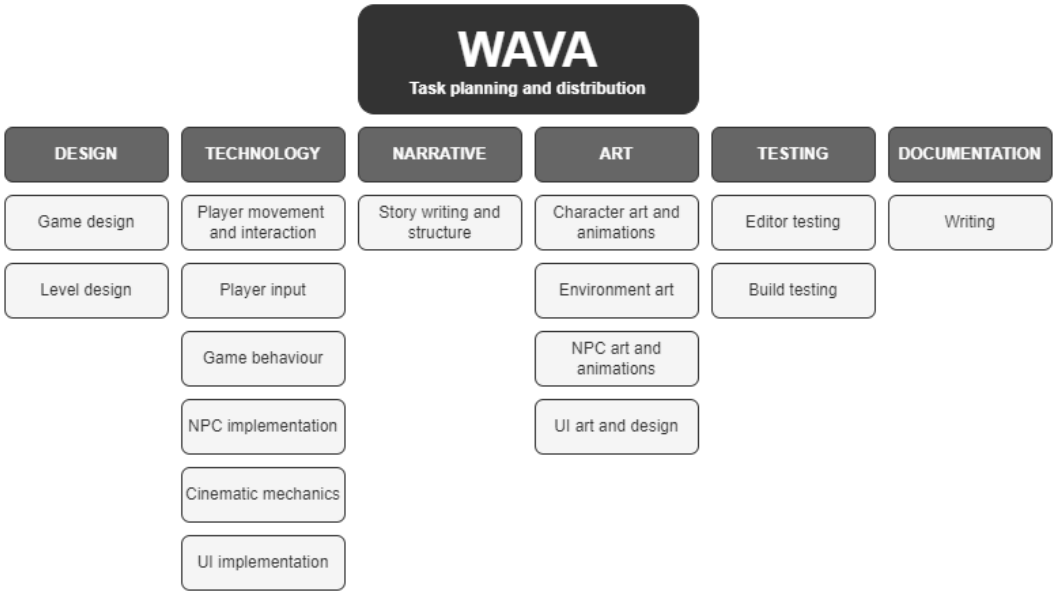

Figure 3.1. Task planning and distribution

3.2.1. Task description

In this section, we will give a more defined description for every task that has been stated in the previous diagram (Figure 3.1). Along with the description, an estimated duration will be set for each one.

Design

- **Game design**
  September 2021 – February 2022
  Game design focuses on the more general aspects of the gameplay, like defining the game mechanics, but also, a quick draft of how the game looks and feels. This process takes a long time since, as stated before, the game's design may vary with the many iterations of implementing and testing.

- **Level design**
  February 2022 – May 2022
  Level design involves the ideation of possible puzzles and zones and, afterwards, creating the levels inside the game engine.

Technology

- **Player movement and interaction**
  October 2021 – February 2022
  This task includes the implementation of all the different ways in which the character can move and interact with the environment. The main interactions are the general movement, climbing, crouching, protecting and picking up elements, among others.

- **Player input**
  October 2021
  This consists of making the necessary preparations to detect player input and making it adaptable in case the input device is changed. For instance, if the player was playing with the keyboard and suddenly decides to switch to a console controller.

- **Game behaviour**
  January 2022 – April 2022
  Implementing the game behaviour includes setting the fail states and managing the general workflow of the game, such as respawning the player when killed or transitioning between scenes.

- **NPC implementation**
  December 2021 – January 2022
  NPC stands for “non-playable character”, so this task includes creating a responsive artificial intelligence for every character that is not directly controlled by the player. NPC implementation includes not only the game’s enemies, but also the characters that accompany the player along the journey.

- **Cinematic mechanics**
  Mid-January 2022 – April 2022
  There must be some extra mechanics that give the game a cinematic feeling, such as animations and cinematics, smooth camera movement or fluid transitioning between levels and chapters.

- **UI implementation**
  February 2022 and April 2022
  UI stands for “user interface”, so this task includes creating a responsive heads-up display, or HUD, developing text prompts as a tutorial for the player, and also designing the main menu and pause menu.

Narrative

- **Story writing and structure**
  September 2022 – February 2022
  Structuring the story correctly for this type of game with meaningful plot points is very important in order to transmit the game’s overall message. This task goes hand in hand with the game design, since the game mechanics are very tied to the story and topic.

Art

- **Character art and animations**
  Mid-September 2021 – February 2022
  Although we have opted towards a minimalist art style, creating smooth character animations is essential so as to give the game a complete and fluid feeling.

- **Environment art**
  Mid-September 2021 – October 2022 and January 2022 – March 2022
  To give the game a proper atmosphere and make it feel unique while keeping a simplistic look, we must take into account several aspects, for instance, choosing the right colours. This task also involves designing and animating the weather for each chapter, which is very tied to the story and overall topic.

- **NPC art and animations**
  December 2021 and February 2022
  This task comprises the design and animation of both the game's enemies' and other side characters.

- **UI art and design**
  February 2022 and April 2022
  This task consists of creating a simplistic user interface that doesn't take any cinematic feeling away from the game, while also being completely understandable and aesthetically tied to the game's overall art style.

Testing

- **Editor testing**
  October 2021 – April 2022
  Testing inside the game engine takes place from the development's start to end. Each new mechanic that has been implemented needs to be tried out and, in some instances, a more general testing of the game is required so as to find possible bugs that need fixing.

- **Build testing**
  Mid-April 2022 – Mid-May 2022
  When the game is starting to take shape and is in its final stages, it is recommended to export it and create early builds to see if it also works as an executable.

Documentation

- **Writing**

  October 2021 – Mid-June 2022

  Last but not least, the game's documentation takes place along all the game's development. When the due date is closer, some revisions and tweaks to the document may be in order.

3.2.2. Gantt chart

After having defined all the tasks that will be developed for this project and their duration, we will now distribute them across the months they will take place in by using a Gantt chart (see Figure 3.2).

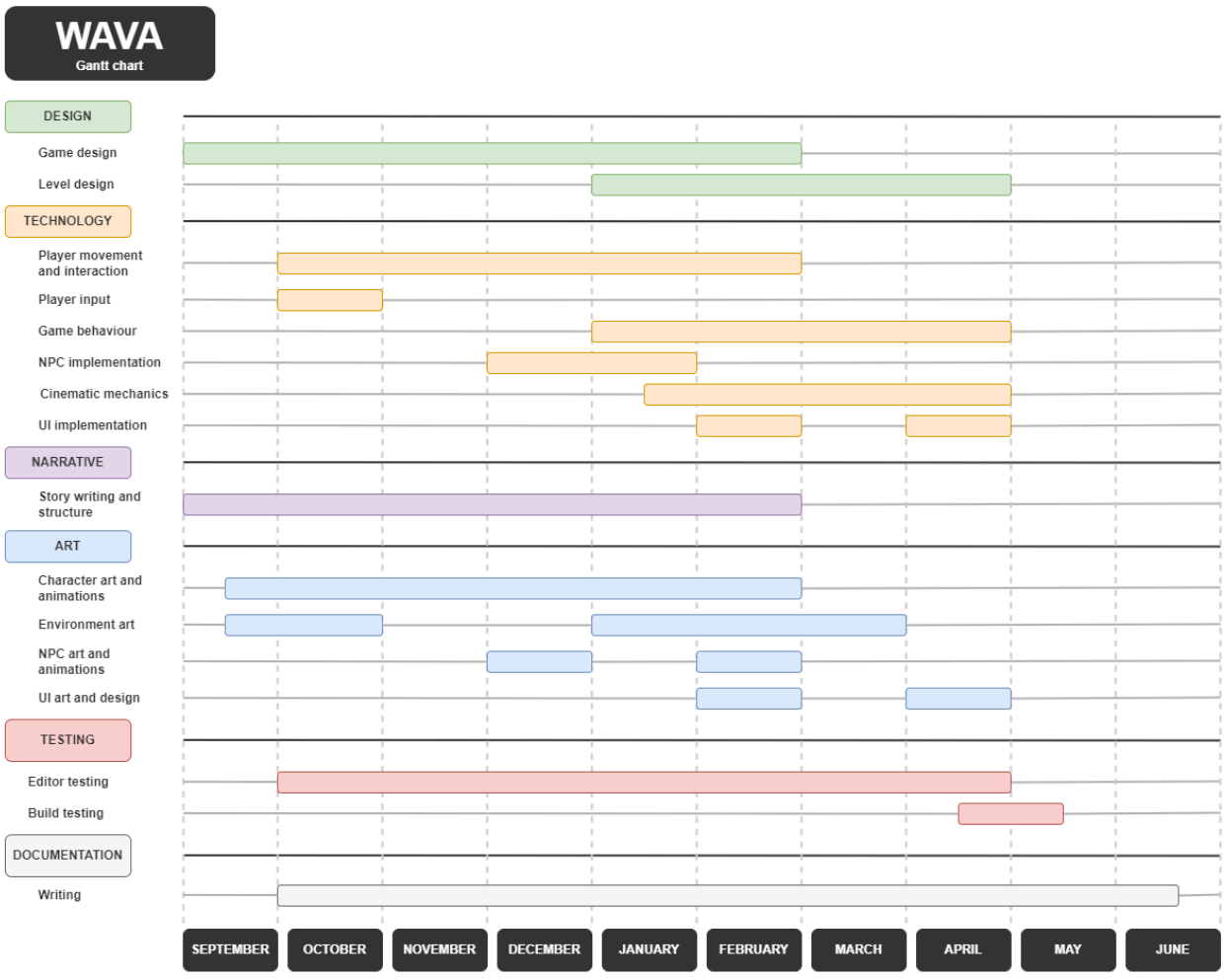

Figure 3.2. Project's Gantt chart

4. Framework and previous concepts

In order to start explaining WAVA's design and development in full detail, it is needed to state where we are coming from. In this section we are going to list the main references that inspired this project and also some key words and concepts that will be used further on.

4.1. References

Video game ideas do not start from scratch, since there are always previous pieces and concepts that have influenced and inspired us. In this section we will state the main sources of inspiration for both this project's gameplay and art style.

4.1.1. Gameplay references

Some of the main game mechanics and overall gameplay were inspired by other similar games, such as *Limbo*, *Inside,* or *Another World* (see section 2.2.2).

4.1.2. Art style references

The main source of inspiration for the main character's design was the illustration named *Familia*, made by the Argentinian illustrator and animator Sebastian Curi (see Figure 4.1).

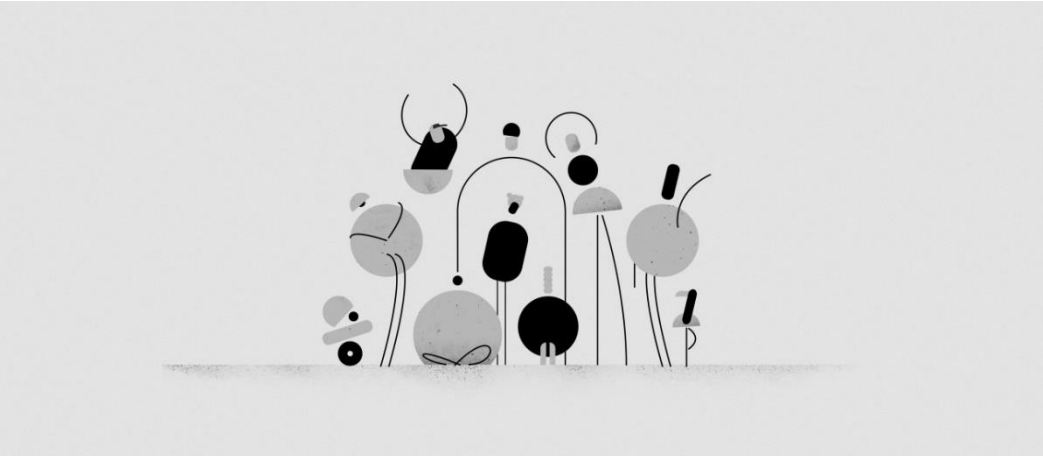

Figure 4.1. *Familia*, by Sebastian Curi

The minimalist aesthetics for the environment and the rest of the game was selected based upon other games from the same genre, such as *Limbo* or *Another World* (see section 2.2.2).

4.2. Vocabulary

Before starting a more thorough explanation of this project's design and implementation, we are going to list a series of concepts and important vocabulary in order to ease the comprehension later on. These concepts are divided into three categories: game engine-related concepts, art-related concepts and general game concepts and acronyms.

4.2.1. Game engine-related concepts

- **Scene:** Scenes in Unity can transition from one another and store all the different game elements. For our project, each chapter or level is stored inside a different Scene. There is also a Scene for the Main Menu and one for the Credits.

- **Prefab:** Prefabs are pre-fabricated GameObjects the user can create and store in order to use them several times throughout all scenes. The different values and components of the object can be edited separately for each instance of the prefab.

- **GameObject:** GameObject is the base class for all entities in a Scene. Scripts and other components can be added onto GameObjects in order to change their behaviour.

- **Transform:** Transform is a basic component found in every GameObject that is used to store and manipulate the position, rotation and scale of the object.

- **Collider2D:** Collider2D is the parent class of the component used to detect all collisions. These colliders can have different shapes and can be configured to be triggers. If a Collider2D is trigger, the collider does not apply any forces on other GameObjects and is just used for detection purposes.

- **Rigidbody2D:** Rigidbody2D is the component that, when added to a GameObject, puts it under the control of Unity's physics engine. For example, when this component is added, the GameObject is then affected by gravity.

- **Coroutine and IEnumerator:** A Coroutine is a function that can suspend its execution (yield) and can be executed over a number of frames or time. For our project, it is used to fade in and out interface elements, set delays and timers, among many other cases. A Coroutine in Unity is implemented inside a function with an IEnumerator type of return.

4.2.2. Art-related concepts

- **Sprite:** A Sprite is a two-dimensional graphic element. In Unity, a component of the same name can be added to a GameObject in order to give it a 2D texture.

- **Spritesheet:** A Spritesheet is an array of Sprites arranged in an ordered manner used to create animations. The animation is based upon quickly switching between the different Sprites in order to create the illusion of movement (see Figure 4.2).

Figure 4.2. Spritesheet from *Megaman* for a running animation

4.2.3. General game concepts and acronyms

- **NPC:** Stands for Non-Playable Character. It is any character that is not directly controlled by the user. It is usually used to refer only to non-hostile entities, but it can also include enemies.

- **AI:** Stands for Artificial Intelligence. In video games, it is the logic that different game entities follow so as to act reasonably and rationally.

- **UI:** Stands for User Interface. It is the collection of visual elements used to show information and allow the player to interact with the game directly. Some examples of UI in games include the main menu and the pause menu (see Figure 4.3).

Figure 4.3. User Interface (UI) for the game Red Dead Redemption 2 (2018)

- **HUD:** Stands for Heads-Up Display. It is the display by which the most important information is shown to the player in-game. An example could be a health bar. In cinematic games, it is very common not to have a HUD.

4.3. North Oriole Games

The final project will be published under the name North Oriole Games, which will appear on later sections of this document. This name corresponds to the solo game studio that was created to publish our games. It is worth noting that this brand was created prior to the beginning project (see Figure 4.4).

Figure 4.4. North Oriole Games primary and secondary logo

5. Design of the videogame

In this section the full design of WAVA will be explained. It will be divided into three different parts, starting with the narrative and artistic aspects. Afterwards, we will define all the different game mechanics and the overall gameplay.

5.1. Narrative

5.1.1. Overview

WAVA's story mainly explores themes of finding our place and learning from our setbacks and problems. Topics of family and friendship are also discussed by using the relationship between the two main characters, who work together as a dyad that represents the human mind and soul.

The narrative is structured across six different chapters with mostly weather related titles. The state of the weather in each chapter is a direct representation of the character's mood. The chapters follow a loop pattern and transition seamlessly from one to the other. The game begins with a vague question, which is contextualised and completed right at the end of the game. Therefore, it is finished, like our characters' journey.

After the story is completed and the credits are shown, the game loops fluidly back to the main menu, which is located in the first zone of the game, characterised by its rainy weather. This cycle mimics a person's cycle of trial and error until they find their definitive place.

5.1.2. Characters

In this section we will examine the game's characters from a narrative perspective. For a more detailed explanation of each character's art design, see section 5.2.2.

The Mind

The Mind is the main character of the game and the one that the user is able to control along the game. Like its name indicates, it represents the human mind. Its main narrative purpose is to display the message of overcoming our own problems, learning from them and growing internally.

The Light

The Light is the other main character and is a representation of the human soul. Its narrative purpose is to first accompany and protect the Mind and, in the end, showcase the message of finding our place.

The Setbacks

The Setbacks are the main enemies of the game and their main purpose is to impede the protagonists to advance. They are a representation of all the problems we may encounter along our lives.

5.1.3. Story structure

In this section we will lay out WAVA's story in full detail, describing its granularity and how the main plot points are presented.

As stated before, the story is spread out across six chapters, two of which are cinematics. These chapters include a prologue, an epilogue and an interlude in order to make the game more theatrical and cinematic.

Beginning cinematic

The game begins in the same rainy forest where the main menu is located. Before starting the first chapter, a message that reads "*What is your place?*" is shown (see Figure 5.1).

Figure 5.1. Screenshot from the beginning cinematic

Chapter 1: The Light & The Mind (Prologue)

The first chapter is named after the main protagonists and it is the only one with a title not related to the weather or environment. Its main function is to serve as a tutorial for the player to get familiarised with the controls and gameplay (see Figure 5.2).

This level is characterised by its rainy weather and for being the shortest and easiest playable chapter. Along this level the protagonists encounter some Setbacks and are presented with the main game mechanics, such as climbing, crouching, protecting or building an auxiliary platform.

When the end is reached, the Mind pulls a lever that elevates a platform that places both main characters inside a cave.

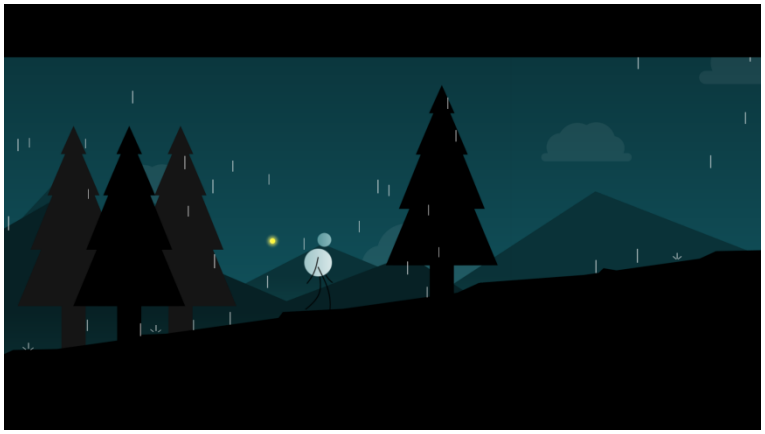

Figure 5.2. Screenshot from the first chapter

Chapter 2: The Rain

The second chapter, as the title itself implies, is also characterised by rain pouring down along the level. Its main function is to reinforce the previous chapter, since they have lots of similarities.

The level starts inside a cave and, once the protagonists get out, they start encountering several Setbacks. These enemies allow the player to gather enough Knowledge in order to go through a large section where the player has to constantly place and recover an auxiliary platform in order to advance.

As they keep moving forward, the characters encounter a small body of water and the player is taught that the Light extinguishes when being in contact with it. After finding another way through, the characters encounter three more Setbacks.

At the end of the chapter, a rock falls onto the Mind and collapses the floor underneath, which submerges the Mind into a large body of water. The Light stays still for a little without being able to aid its friend. After a while, the Light ascends rapidly onto the sky, leaving the Mind behind (see Figure 5.3).

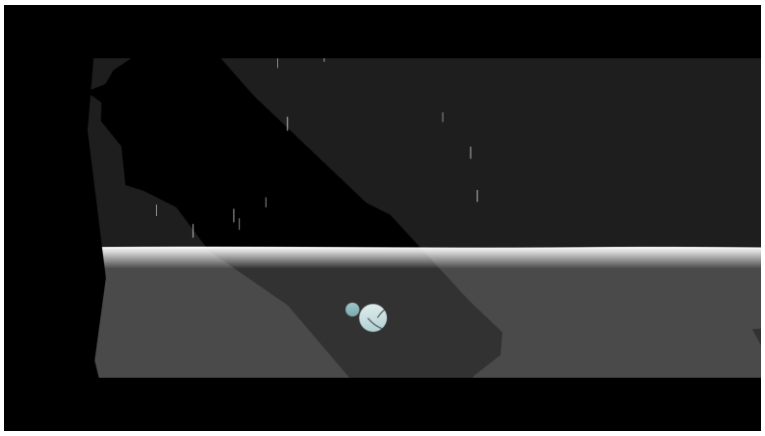

Figure 5.3. Screenshot from the second chapter

Chapter 3: The Gloom

The third chapter begins with the Mind submerged in the water, but it is not unconscious anymore. The player is taught that the Mind is able to swim underwater as it makes its way out of the water and the cave it fell into. Outside of the cave, the rain has stopped, but gloom and darkness are still invading the environment (see Figure 5.4).

Since the Light is gone, the Mind cannot protect itself from Setbacks, thus forcing it to go past them by diving into water. The Mind explores its environment and, luckily, finds enough Knowledge to build an auxiliary platform and move forward. After a while inside the cave, the Mind stops abruptly, as if it had seen something special.

Figure 5.4. Screenshot from the third chapter

Chapter 4: The Moon (Interlude)

The fourth part of the story is the first cinematic chapter. We see the Mind is standing still, but suddenly starts walking out of frame.

Shortly after, the camera follows the same path as the Mind, revealing an exit to the cave and a gorgeous starry night sky with a huge moon (see Figure 5.5). The Mind is stunned admiring the scenery when the player gets a button prompt which, when pressed, triggers a bubble to emerge from our protagonist, symbolising that it can now protect itself without the assistance of the Light.

Much like Plato's *Allegory of the Cave*, the Mind has completed its catharsis, found its own truth and is now free and self-sufficient.

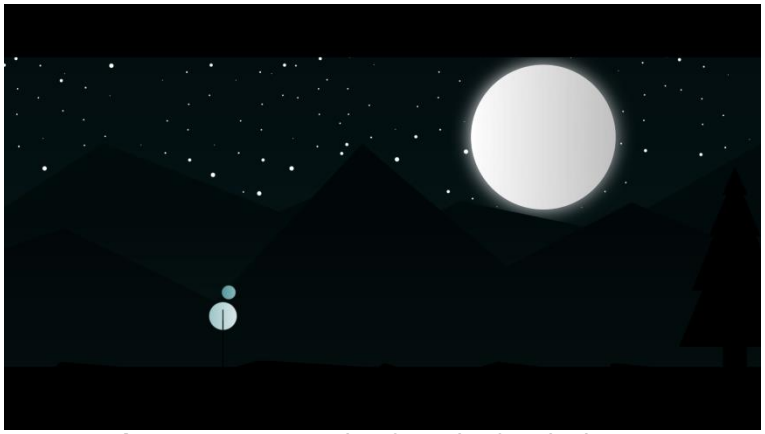

Figure 5.5. Screenshot from the fourth chapter

Chapter 5: The Breeze

The fifth part is the one with the clearest weather. It is also characterized by a slight breeze that incites the Mind to keep moving forward (see Figure 5.6).

After learning how to protect itself, the Mind is now able to stand up to Setbacks, swim and overcome any sort of obstacle. By applying all that it has learnt along its journey, the Mind encounters a lever that lifts it up to a very high cave. Much like in Chapter 4, after a while inside the cave, the Mind stops abruptly.

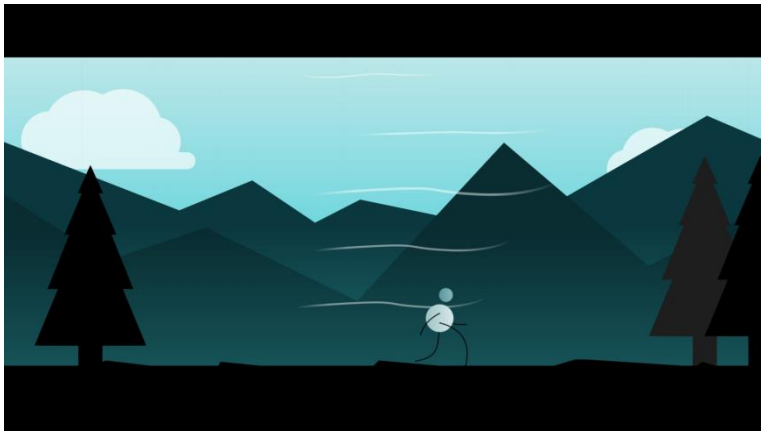

Figure 5.6. Screenshot from the fifth chapter

Chapter 6: The Sun (Epilogue)

The sixth part of the story is the second cinematic chapter. Similarly to Chapter 4, the Mind starts walking out of frame. The camera pans towards the Mind but, this time, it reveals a crack on the cave's wall and a huge sun peeking through (see Figure 5.7).

While the Mind is watching the sun, the player gets a button prompt. When pressed, the sun starts shaking, revealing that it is indeed the Light, which has grown and found its place. Although their paths have been different, their love for one another is so strong that they still have that special bond that united them in the beginning.

The intention behind this interaction is to portray themes of family, friendship and always being present when needed, no matter our journeys.

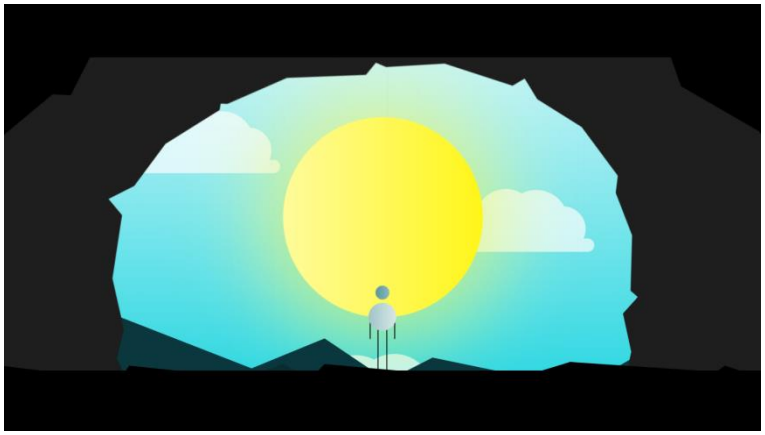

Figure 5.7. Screenshot from the sixth chapter

Ending cinematic

At the end, the screen turns yellow and a text cinematic with these messages is played (see Figure 5.8):
"*From all setbacks, we learn, we grow.*"
"*And no matter how lost we feel, we find our home, where we belong.*"
"*So let me ask you again.*"
"*What is your place…*"
"*... Where All Vanishes Away?*"

After showing the meaning behind the acronym WAVA, the game's logo is shown before transitioning to the credits.

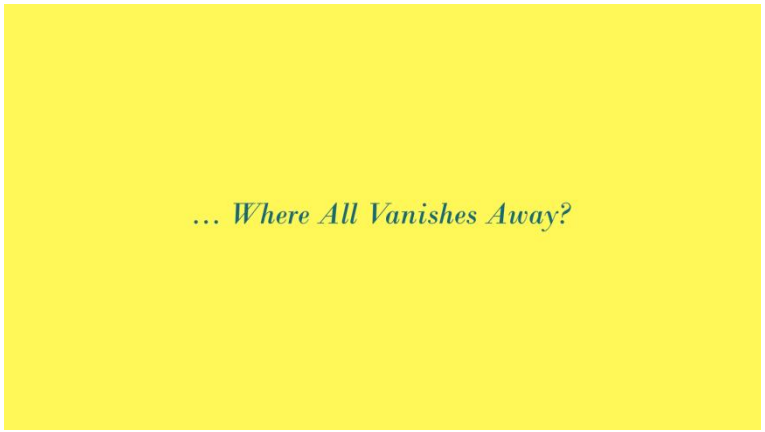

Figure 5.8. Screenshot from the ending cinematic

5.1.4. Narrative devices

As WAVA is mainly visual, the narrative of the game is told through a series of methods and devices.

- **Mechanics:** The main narrative devices used are the game's mechanics and the overall gameplay. The Mind needs to overcome Setbacks in order to gain Knowledge. When it gains a certain amount of Knowledge, it can create an auxiliary platform that allows it to get to unreachable places and advance further, until it finds its place. The names for each game element have been carefully chosen so they are self-explanatory and the player can easily understand the game's message.

- **Text Cinematics:** Apart from the prompts that teach the player the controls, the only texts that appear on screen are the text cinematics shown at the beginning and end of the game. These texts are meant to make the player reflect on their own interpretation of their place in the world.

- **Visual Cinematics:** Along the game, there are a number of visual cinematics and animations that help illustrate the main plot points. There are also two chapters which are mainly cinematic due to their importance to the story. This allows us to be in control of the pace and assure the player does not miss any important story pieces.

- **Weather:** The weather and title of the chapters reflect the current emotions and feelings of the main character. The constant improvement towards a more pleasant weather represents the catharsis of the Mind and how it is getting closer to finding its place. The rain pouring down in the first two chapters symbolises the Mind being in an undesirable place that makes it feel trapped and constantly bombarded. After the rain, there is gloom, which represents the Mind still being clouded with thoughts and worries. Last but not least, the breeze in the last chapter is a representation of the sensation of freedom the Mind feels due to finally being self-sufficient.

- **Level design:** The level design for each chapter also has certain importance on the narrative. The different challenges the Mind encounters along each chapter reinforce its capabilities, but also its shortcomings. For example, when the Mind can swim but not protect itself, there are easily avoidable Setbacks placed in the water.
  The composition of the levels is also key; for instance, since in Chapter 5 the Mind is closer to reaching its place, the route it has to follow goes upwards. On the other hand, since in Chapter 3 the Mind is alone and lost, most of the level happens inside underwater caves that force the Mind to dive downwards.

5.2. Art

5.2.1. Overview

The general art style chosen for the game can be described as minimalist and geometric. All the different game assets have very simple and basic shapes and are painted with a flat colour, except for the more important entities, which have a slight gradient.

The colours chosen play a huge role into the artistic aspect of the game. Everything is blue-toned, black or white, including one of the main characters, the Mind. The only entity that has a different and unique colour is the other protagonist, the Light, which is painted yellow in order to distinguish it easily in the final chapter.

The main character animations were done in Inkscape by using the technique of spritesheet animation. Each animation has a different number of frames depending on its complexity. Other simpler animations were achieved by simply moving the GameObjects via Unity's co-routines.

5.2.2. Characters

In this section we will examine the game's characters from a visual and artistic perspective. For a more detailed explanation of each character's narrative importance, see section 5.1.2.

The Mind

The controllable character of the game, the Mind, is formed by two blue-toned circles of different sizes to represent its head and body, and four black lines to represent its extremities (see Figure 5.9).

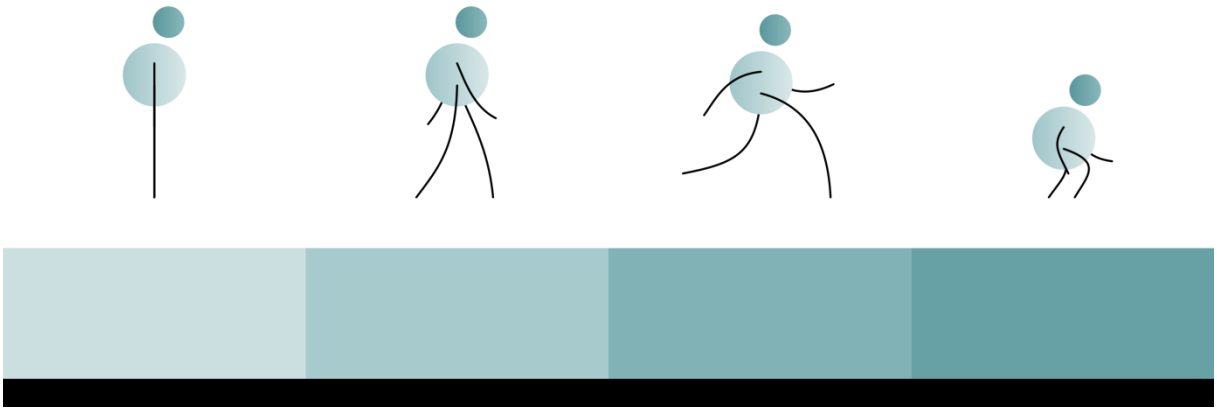

Figure 5.9. Design and colour palette of The Mind

It has fourteen different and unique animations that bring the character to life. These animations include idling, walking, running, crouching, climbing, hurting and dying, among others (see Figure 5.10).

Figure 5.10. Spritesheet of The Mind's walking animation

As previously noted, it is inspired on a design by the Argentinian artist Sebastian Curi (see Figure 5.11).

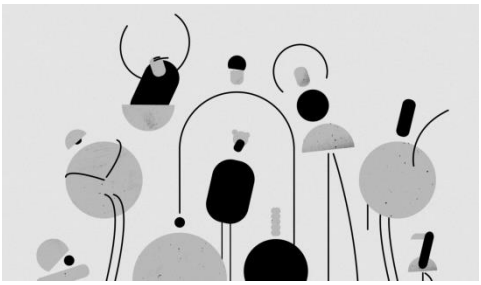

Figure 5.11. Close-up of *Familia*, by Sebastian Curi

The Light

The other main character is the Light, which is formed by a yellow circle surrounded by a slight yellow glow. It is the only game entity that is not blue-toned, black or white (see Figure 5.12).

Figure 5.12 Design and colour palette of The Light

It has three different animations, which include one looping idle animation that moves the Light up and down, and two interaction animations. These interactions make the light either shake or circle around (see Figure 5.13).

Figure 5.13. Spritesheet of The Light's idling animation

The Setbacks and Knowledge

The enemies the protagonists protect themselves from, the Setbacks, are characterized by being black with a glowing white core. This white orb represents the Knowledge they have inside, which can be picked up by the player once the Setback has been defeated (see Figure 5.14).

There are two different types of Setback, the *bomber* and the *kamikaze*. The first one attacks the player by emitting a lethal explosion and is shaped like a square, while the latter is a star polygon and attacks the players by hunting them down.

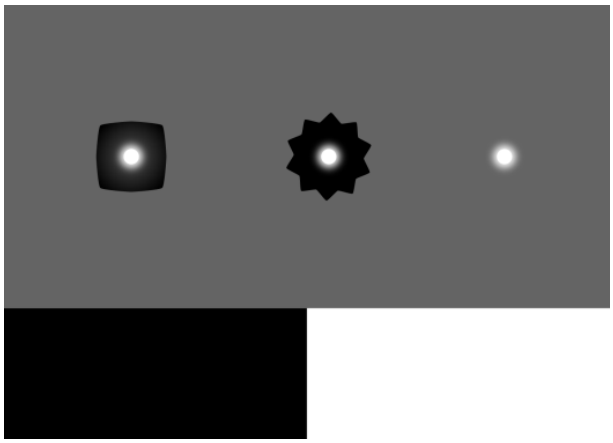

Figure 5.14 Design and colour palette of The Setbacks and Knowledge

The Halo

The Halo is a glowing bubble that surrounds the main character in order to protect it from the Setbacks. In chapters one and two, when the Light is the one that spawns it, the Halo is yellow. On the other hand, in chapters four and five, when the Mind is the one that does so, the Halo is white (see Figure 5.15).

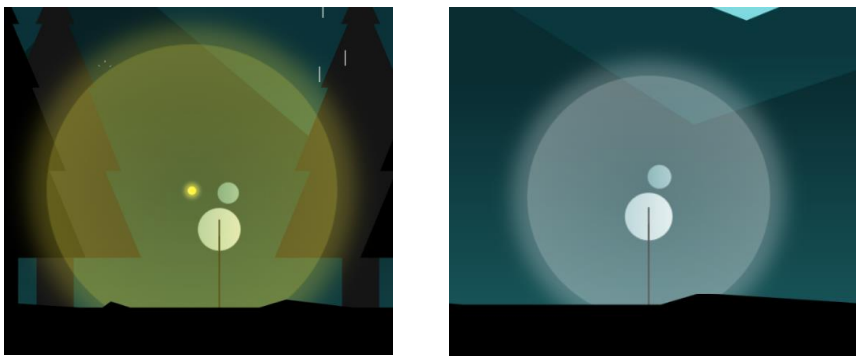

Figure 5.15. View of the Halo depending on the chapter

5.2.3. Environment

The environment for the game is also very minimalist and is based on a somewhat plausible forest. It is divided into a blue-toned background and a black, white and grey foreground. The background includes the sky, mountains and clouds, while the foreground includes trees, ground, platforms, spikes, levers and caves (see Figure 5.16).

In order to create a more realistic feel, the different environment elements scroll across the screen with a parallax effect. So, the further the element is from the camera, the slower it moves in relation to the character.

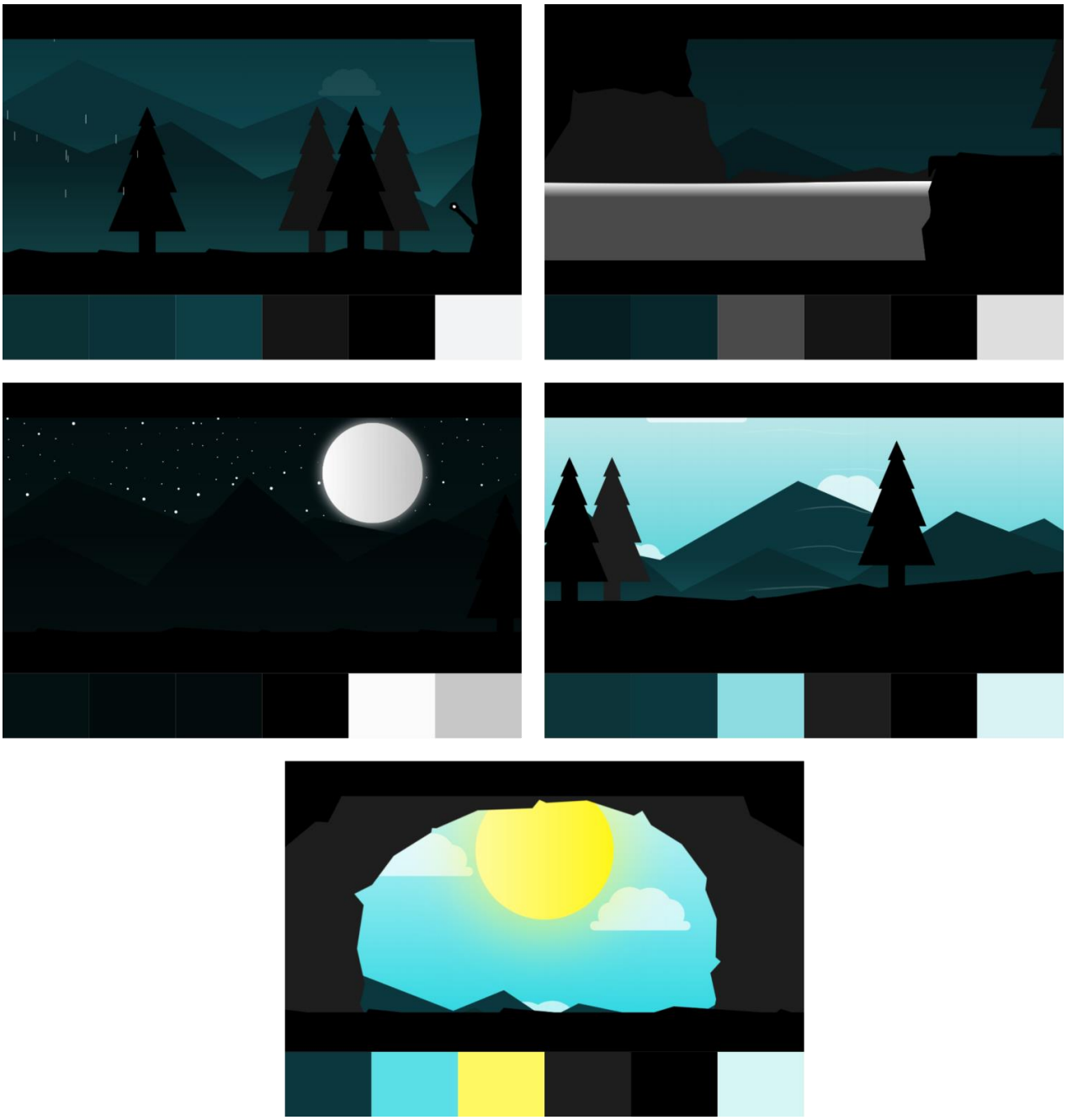

Figure 5.16. Screenshots and colour palettes for five different environments of the game

5.2.4. Weather

The weather in each level plays an important role in the game, so it should be represented properly while still keeping a simplistic look. The animations for the different weather types were also done via spritesheet animation in Inkscape.

Rain

The rain appears in the first and second chapter. There are several raindrop particles that are spawned along the levels which, when in contact with the ground, character or other world elements, do a splash animation (see Figure 5.17).

Figure 5.17. View in game and spritesheet of the rain animation

Breeze

The breeze appears in the fifth chapter. After a short delay, five gusts of wind are spawned along the level repeatedly. These gusts of wind fade in and out and do a swift swirl animation (see Figure 5.18).

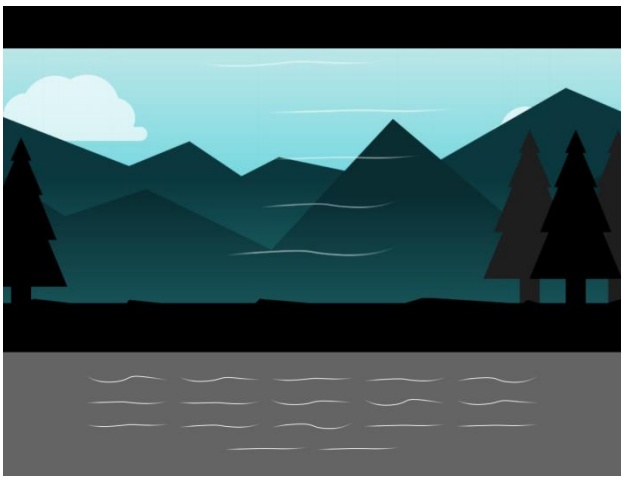

Figure 5.18. View in game and spritesheet of the breeze animation

5.2.5. Interface

Branding

In order to create a visually coherent product, it is very important to create the right branding and establish our game as a unique piece.

The font used for all texts in the game is the serif font named Bodoni MT. In order to give the text more dynamism, the font is always italicized.

This is an example of the appearance of the font.

The logo for the game is very geometric and uses different blue tones and gradients, similarly to the game itself. The letters also combine perfectly and form a shape that resembles the mountains found in-game (see Figure 5.19).

Figure 5.19. WAVA's final logo design

Menus

The main menu is located in the rainy section of the game, right before where the first chapter begins. It consists of the game logo and four buttons the player can interact with: one for playing, another one for changing the options, the next one to go to the credits scene and the last one to exit the game (see Figure 5.20).

Figure 5.20. Main menu of WAVA

Once pressed the play button, the players can select which level they want to start from, allowing them to play the full story or just replaying a certain chapter. The options menu has a toggle that allows the players to decide if they want the game in full screen or not. This option is on by default (see Figure 5.21).

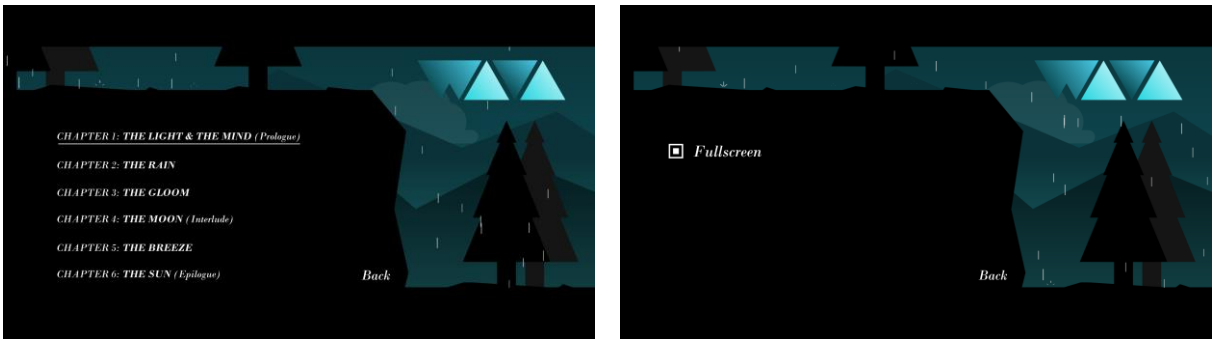

Figure 5.21. Play and options screens in the main menu

The pause menu consists of a semi-transparent black background with four different buttons: one to resume the game, another one for the options, the other one for viewing the game's controls and the last one to exit to the main menu (see Figure 5.22).

Figure 5.22. Pause menu of WAVA

Inside the controls' menu, the players can check the different buttons that need to be pressed in order to do each action, depending on the used controller.
The options in the pause menu also include a toggle to set the game in full screen, and a toggle to hide the HUD can also be found. This option gives the game a more cinematic feel and is better for taking screenshots (see Figure 5.23).

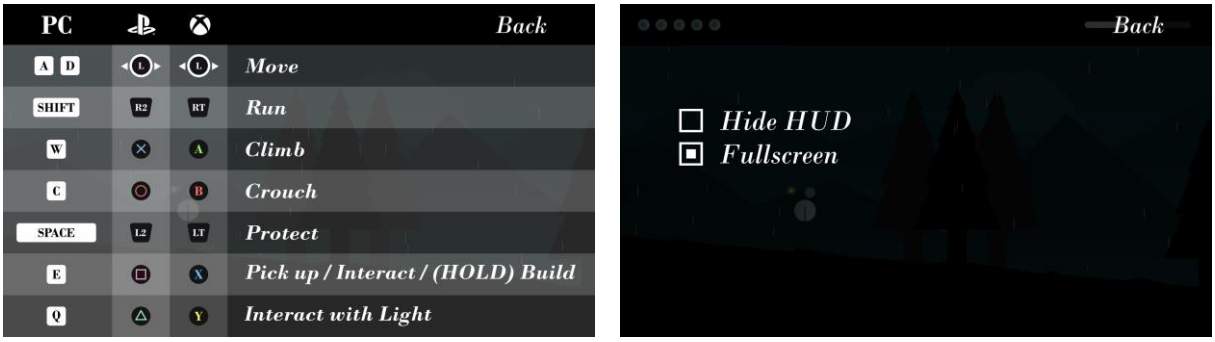

Figure 5.23. Controls and options screens in the pause menu

For both menus, when a button is hovered on or selected, the button's text is underlined.

HUD

The heads up display for the game is very simple so as not to take away any immersion from the player. It can be divided in three parts (see Figure 5.24):

- Two black bars at the top and bottom of the screen that give the game a more cinematic aspect.
- An array of five circles the same colour as the Mind on the top left corner that represents the lives.
- A white slider on the top right that represents the current Knowledge gathered by the players.

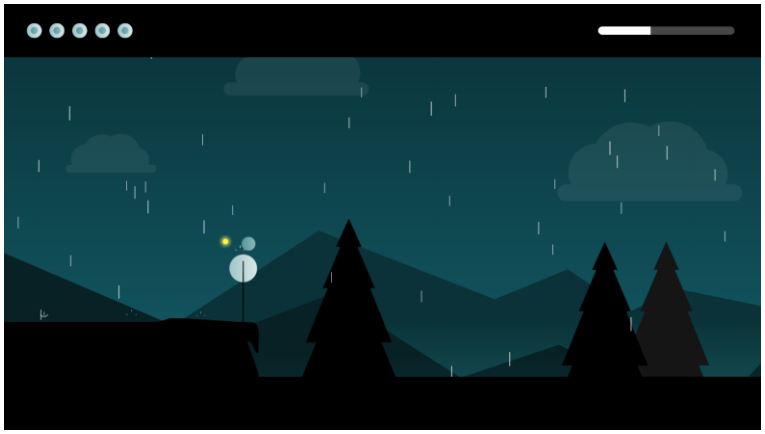

Figure 5.24. Heads up menu (HUD) of WAVA

5.3. Gameplay

5.3.1. Overview

The general gameplay of WAVA is very tied to its narrative and, like other cinematic puzzle-platformers, it follows a very simple and easy to understand game loop. In our case, this loop consists of defeating the Setbacks by protecting yourself, pick up the Knowledge they leave behind and then build auxiliary platforms that allow the character to go further and advance to the next level.

The game's movement and view is two-dimensional, using a side-scrolling format for the camera. The action is viewed from a side-view camera angle that smoothly follows the player with a slight horizontal and vertical offset, incrementing the view of what's in front and on top of the main character. There is only one difficulty level, which has been tested in order to be accessible for all players, no matter their level of expertise.

5.3.2. Challenge hierarchy

WAVA follows a very linear structure, as the player advances from one chapter to the next one. The main challenge is to reach the end of each chapter and complete the game. However, inside each chapter there are obstacles the player has to overcome. The main challenges found inside each chapter are:

- Protect the character from the Setbacks, defeat them and pick up the Knowledge they leave behind.
- Build auxiliary platforms by using the Knowledge in order to climb up to unreachable places.
- Avoid falling onto spikes and other world elements that might be lethal.

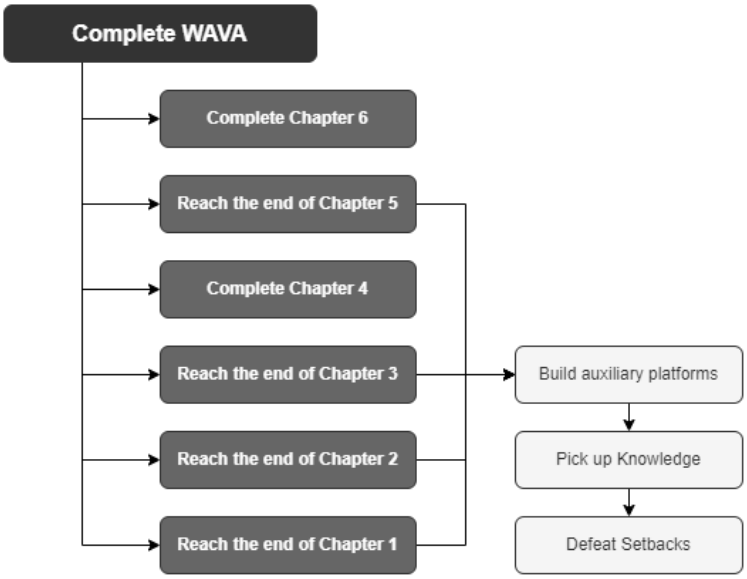

Figure 5.25. Challenge hierarchy of WAVA

Even though some challenges and obstacles may vary depending on each chapter, Figure 5.25 visually represents the main challenge hierarchy of WAVA.

5.3.3. Mechanics and player actions

Similarly to other games, the game mechanics in WAVA are fairly common and easy to understand in order to be more accessible for less experienced players. There are a set of actions the players can do to control the main characters and complete the levels so as to advance the plot.

Movement

Across all chapters, the players can move the main character around the horizontal axis of the game world by either walking or running, depending on the desired speed. In addition, from chapter three to chapter five, the Mind can also swim underwater, which adds vertical movement to the player.

Climbing

Unlike other platformers, the main character of WAVA cannot do impressive jumps. In order to move vertically, the players can climb onto ledges when located next to them and as long as the character is not falling down.

Crouching

To enter caves and reach places with short entrances, the players can crouch and move in the horizontal axis while squatting.

Protecting from Setbacks

All Setbacks will try to harm the players and, although the main character cannot attack directly, it can protect itself by creating a bubble, named Halo, which wraps the players around and is lethal to the Setbacks.

In chapters one and two, the Light is the one that spawns the Halo, while in chapters four and five it is the Mind that does so.

Picking up Knowledge

After defeating a Setback, it will leave Knowledge behind, which can be picked up by the player. This Knowledge comes in the shape of a glowing white orb.

Building an auxiliary platform

When five Knowledge orbs have been picked up, the player is able to build an auxiliary platform in front of them in order to reach higher surfaces. However, this is only possible when a world element is not on the way and the character is on a building zone, which is represented by a white glow on the ground.

If an auxiliary platform is placed nearby and the player is on a building zone or on top of said platform, the user is able to recover it in order to build another one later on.

Interacting with a lever
There are some levers placed around the world that, when interacted with, move a certain world object that enables the player to keep advancing.

Interacting with the Light
The Mind can interact with the Light, triggering one of two interaction animations. Even though this action does not have any value gameplay-wise, it is intended to reinforce the overall narrative and the relationship between the main characters.

Most game mechanics are presented along the first chapter, which is designed to be a tutorial for all users and familiarize them with the controls and overall game behaviour. However, some mechanics and other game aspects are introduced in latter chapters, so as not to overwhelm the player with excessive new information.

5.3.4. Economy and resources

The internal economy of WAVA is very simple and is based on very few resources. These resources can be tangible or intangible, depending on whether they have physical properties in the game world. They can also be categorized as concrete or abstract, the latter being those that do not really exist in the game but are used for internal computation. In WAVA, the three main resources are:

- **Lives:** These are intangible and concrete resources that represent the player's health. The default number of lives is five (see Figure 5.26).

Figure 5.26. Visual representation of the lives on the HUD

- **Knowledge:** This is a resource that is left behind by Setbacks once defeated and can be picked up by the player. Therefore, Knowledge is tangible and concrete (see Figure 5.27).

Figure 5.27. Visual representation of the Knowledge in-game and on the HUD

- **Auxiliary platform:** This resource can be spawned in building zones once the player has accumulated five Knowledge orbs. It can be climbed on and is also a tangible and concrete resource (see Figure 5.28).

Figure 5.28. Visual representation of the auxiliary platform in-game

Economies typically include four types of functions that affect resources and move them around. These mechanics are called sources, drains, converters, and traders.

We will now list for each of these four types of mechanics, their corresponding examples in WAVA's internal economy:

- **Sources:** These are mechanics that create new resources out of nothing.
  - **In WAVA:**
    - After losing all five lives and once the player respawns, the five lives are fully restored.

- **Drains:** Contrary to the sources, they reduce the current resources.
  - **In WAVA:**
    - When the player is hurt, a specific amount of lives are taken out. When hurt by a Setback, only one life is removed. On the other hand, all five lives are erased when hurt by spikes or water, the latter only affecting the player when accompanied by the Light (see Figure 5.29).

Figure 5.29. All lives being removed after being hurt by spikes

- **Converters:** These mechanics turn resources of one kind into another.
    - **In WAVA:**
        - Once defeated, the Setbacks, main enemies in the game, turn into Knowledge that can be picked up.
        - When the player has gathered five Knowledge orbs and is on a building zone, the player can turn this Knowledge into an auxiliary platform. This action also goes the other way around, as the player can recover the box and convert it into full Knowledge (see Figure 5.30).

Figure 5.30. An auxiliary platform being built in exchange for full Knowledge

- **Traders:** These are mechanics that move a resource from one entity to another and another resource back in the opposite direction, according to an exchange rule.
    - **In WAVA:**
        - Due to the game's cinematic nature, the trader mechanic did not make sense within the design of the game. Therefore, there are no traders in WAVA.

5.3.5. Level design

In this section we will see a general visual view of the level design for each one of the four playable chapters, along with a description of each level's zones and challenges.

Chapter 1: The Light & The Mind (Prologue)

The first playable chapter of the game functions as a prologue for the narrative and a tutorial of the game mechanics and general gameplay for the player. As the title indicates, it is also deeply centred on the relationship between the Mind and the Light. This level can be divided into four parts:

- The first zone has no real threats for the players and is designed to teach them the most basic motion controls, which include walking, running and climbing onto ledges (see Figure 5.31).

Figure 5.31. View of the first zone in Chapter 1

- The second zone focuses on the "combat" aspect of the game. Both types of Setback are presented, along with the controls to protect and pick up Knowledge. The player must defeat four Setbacks, two *kamikazes* and two *bombers* (see Figure 5.32).

Figure 5.32. View of the second zone in Chapter 1

- The third zone combines two new game mechanics: crouching and building an auxiliary platform. This is achieved by placing the last Knowledge orb needed to build the auxiliary platform inside a cave that requires the player to crouch in order to enter (see Figure 5.33).

Figure 5.33. View of the third zone in Chapter 1

- The fourth zone is centred on the interaction between the Mind and the Light and also introduces the lever interaction mechanic. The lever triggers a platform that lifts the character upwards (see Figure 5.34).

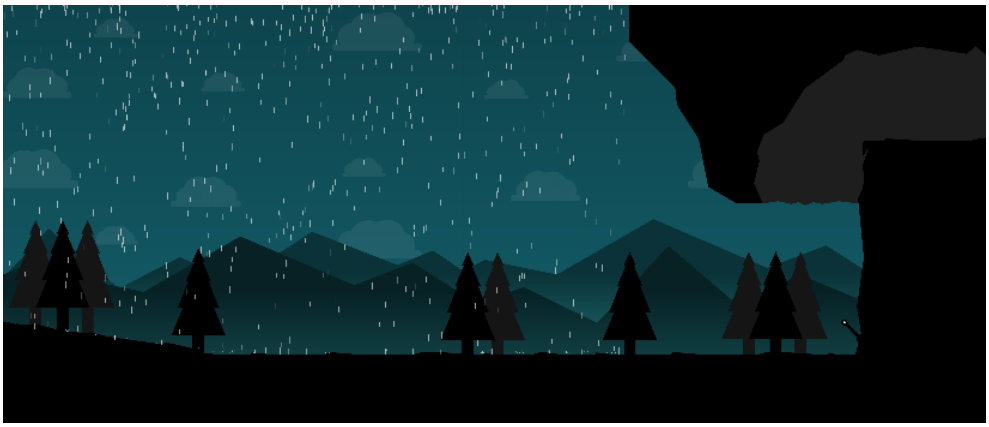

Figure 5.34. View of the fourth zone in Chapter 1

Chapter 2: The Rain

The second chapter is visually equal to the first chapter, since the rainy weather is still present; however, the gameplay is slightly more difficult. This level can be divided into four different parts:

- The first zone starts by dropping the player downwards to face five different Setbacks, placed strategically to be somewhat challenging for the user (see Figure 5.35).

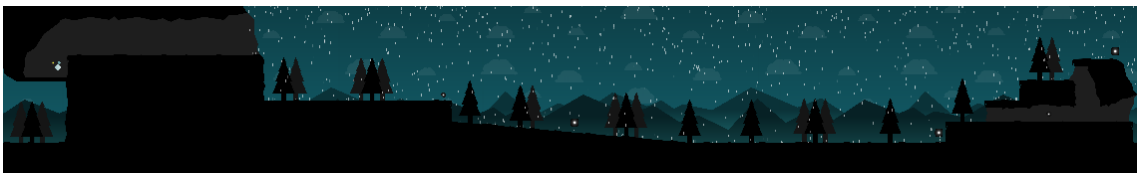

Figure 5.35. View of the first zone in Chapter 2

- With full Knowledge, the second zone is fully focused on the game mechanic of building an auxiliary platform and recovering it in order to advance. In addition, spikes are presented as a new threat to the main character. The zone ends on top of a high mountain (see Figure 5.36).

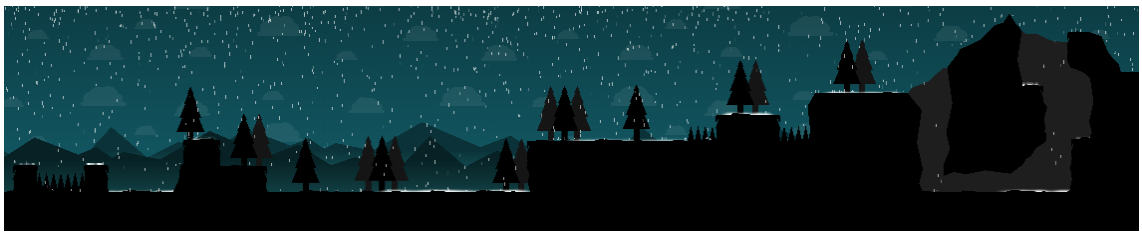

Figure 5.36. View of the second zone in Chapter 2

- The third zone introduces the incapability of the Light to go into water. This instigates the player to find another way around, which forces the protagonists to go downwards, pull a lever and crouch through a cave (see Figure 5.37).

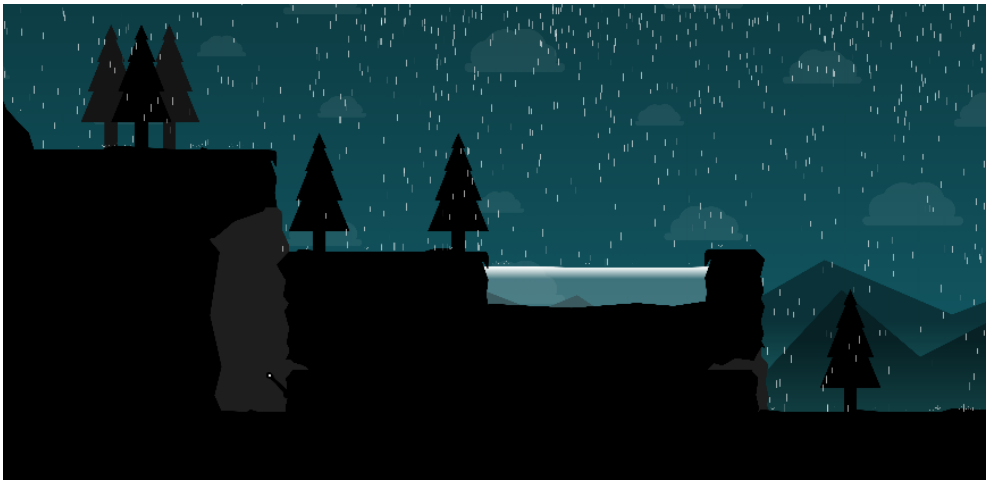

Figure 5.37. View of the third zone in Chapter 2

- The fourth and last zone is uphill and, in order to reach the top, the player must defeat three Setbacks (see Figure 5.38). At the top, a rock falls onto the protagonists and collapses the floor underneath, submerging the Mind into a large body of water. After a while, the Light ascends rapidly to the sky, leaving the Mind behind.

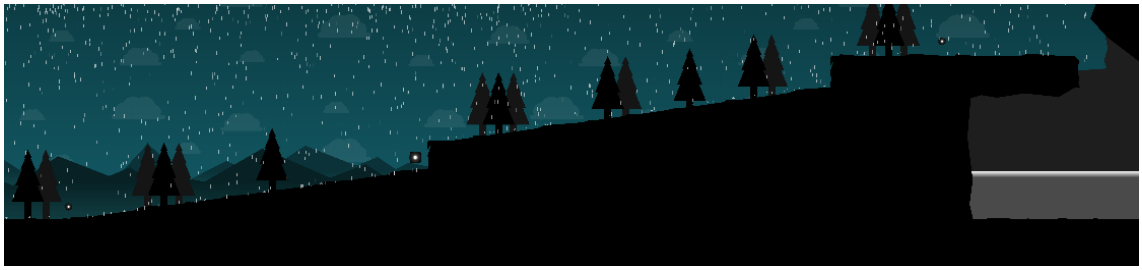

Figure 5.38. View of the fourth zone in Chapter 2

Chapter 3: The Gloom

The third chapter is the first one where the Mind is alone, thus it is heavily designed to accentuate its new capabilities and shortcomings. This is achieved by relying heavily on underwater movement and decreasing the number of Setbacks encountered, as well as placing them in easily avoidable places. This level can be divided into five parts:

- The first zone begins in the same body of water the Mind fell into in the last chapter. Its main function is to introduce the underwater movement mechanic (see Figure 5.39).

Figure 5.39. View of the first zone in Chapter 3

- The second zone is uphill and, since it is set in the exterior of the forest, the player sees the rain has stopped. It also includes a *kamikaze* Setback which is intended to make the player realize they can no longer protect themselves, which forces them to crouch into a cave (see Figure 5.40).

Figure 5.40. View of the second zone in Chapter 3

- The third zone is designed to reinforce the underwater movement mechanic by putting an obstacle for the player to get through. After exiting the water, the level continues going upwards (see Figure 5.41).

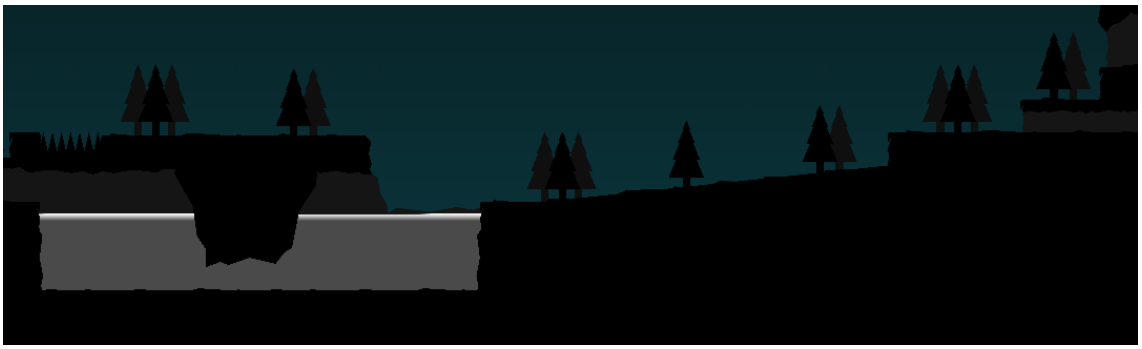

Figure 5.41. View of the third zone in Chapter 3

- The fourth zone is a more complex underwater cave section, with three *bomber* Setbacks scattered around. Along this zone, players can find and pick up two Knowledge orbs that allow them to build an auxiliary box and advance to the next zone (see Figure 5.42).

Figure 5.42. View of the fourth zone in Chapter 3

- The fifth and final zone is a cave section that combines spikes, crouching and other game mechanics. The zone's layout forces the player to go downwards (see Figure 5.43).

Figure 5.43. View of the fifth zone in Chapter 3

Chapter 5: The Breeze

The last playable chapter combines all the different game mechanics learnt along the previous levels, since the Mind is now able to swim and protect itself. This level can be divided into six parts:

- The first section has a peaceful beginning. After a tranquil stroll, the player encounters two Setbacks, both of which can be easily defeated (see Figure 5.44).

Figure 5.44. View of the first zone in Chapter 5

- The second zone is an underwater section that combines the swimming and protecting mechanics, as there is one Setback inside the water (see Figure 5.45).

Figure 5.45. View of the second zone in Chapter 5

- The third zone has two more Setbacks, two building zones and spikes, leading the player onto a cave (see Figure 5.46).

Figure 5.46. View of the third zone in Chapter 5

- The fourth zone contains the second underwater zone, along with three Setbacks in the player's way (see Figure 5.47).

Figure 5.47. View of the fourth zone in Chapter 5

- The fifth zone forces the player downwards to face one last Setback. However, at the end of this section the player encounters a lever that lifts the character upwards, similarly to the one in Chapter 2 (see Figure 5.48).

Figure 5.48. View of the fifth zone in Chapter 5

- The sixth and final zone is a cave section which starts by going downhill. At the bottom of the hill, the player can pick up the last Knowledge orb needed to build an auxiliary platform and go upwards. The section ends with an ascending hill, placing the Mind higher than ever before (see Figure 5.49).

Figure 5.49. View of the sixth zone in Chapter 5

As it can be seen in the figures from Figure 5.31. View of the first zone in Chapter 1 to Figure 5.49, each level begins in the same place the last one ended in order to give the game continuity and make the world coherent.

Chapters 4 and 6 are mainly cinematic and not controlled by the user, as they serve a predominantly narrative purpose. The only player interaction in these chapters is pressing a button when a prompt appears, which triggers an animation that reinforces the story.

5.3.6. Game layout chart

In this section, we will see a representation of the game's overall layout, analysing how every scene and every chapter of the game is connected to one another.

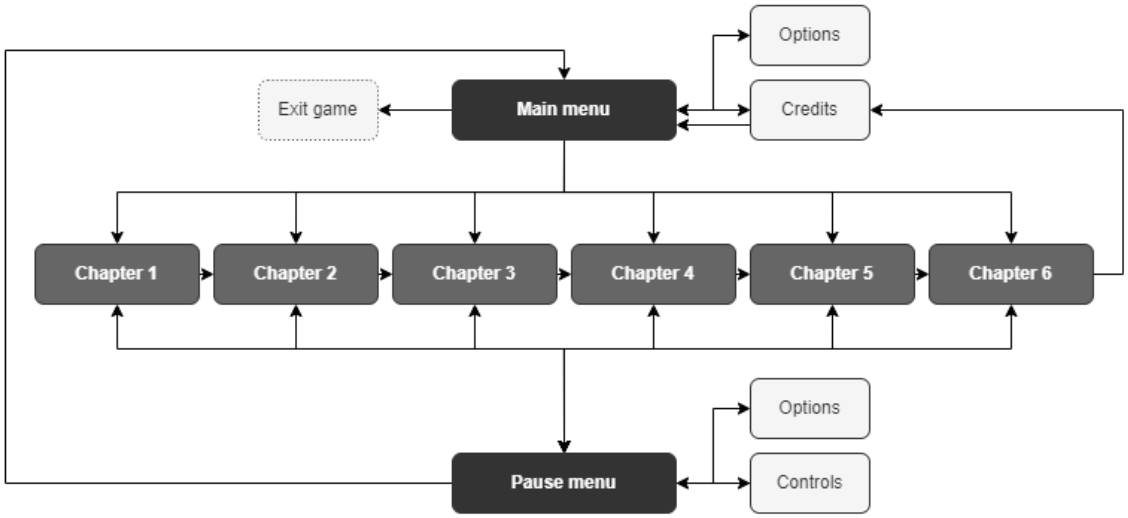

Figure 5.50. Game layout chart of WAVA

As it can be seen in the chart represented in Figure 5.50, the main menu of the game allows the player to select the desired chapter, change the options for the game, watch the credits or exit the game.

Every chapter is played in an ordered linear sequence. After chapter 6 is completed, the credits are displayed and, once these are also completed, the user goes back to the main menu.

In addition, the player is able to pause the game at any time to change the options, check the game's controls or leave to the main menu. Alternatively, the player can resume the game and go back to the chapter.

6. Implementation and testing

After having fleshed out the full game's design, in this section we will give a detailed explanation of the game's implementation, programming, and other aspects related to the game engine. Afterwards, we will also present how these implementations were tested along the development of the project.

6.1. Class diagram

The implementation of WAVA is distributed across 38 scripts, each one defining a different class. In Figure 6.1 we can visualise all the game's classes divided by their category and how some of them are related to the others. In the following parts of this section, we will give a detailed definition for each class and their respective variables and functions.

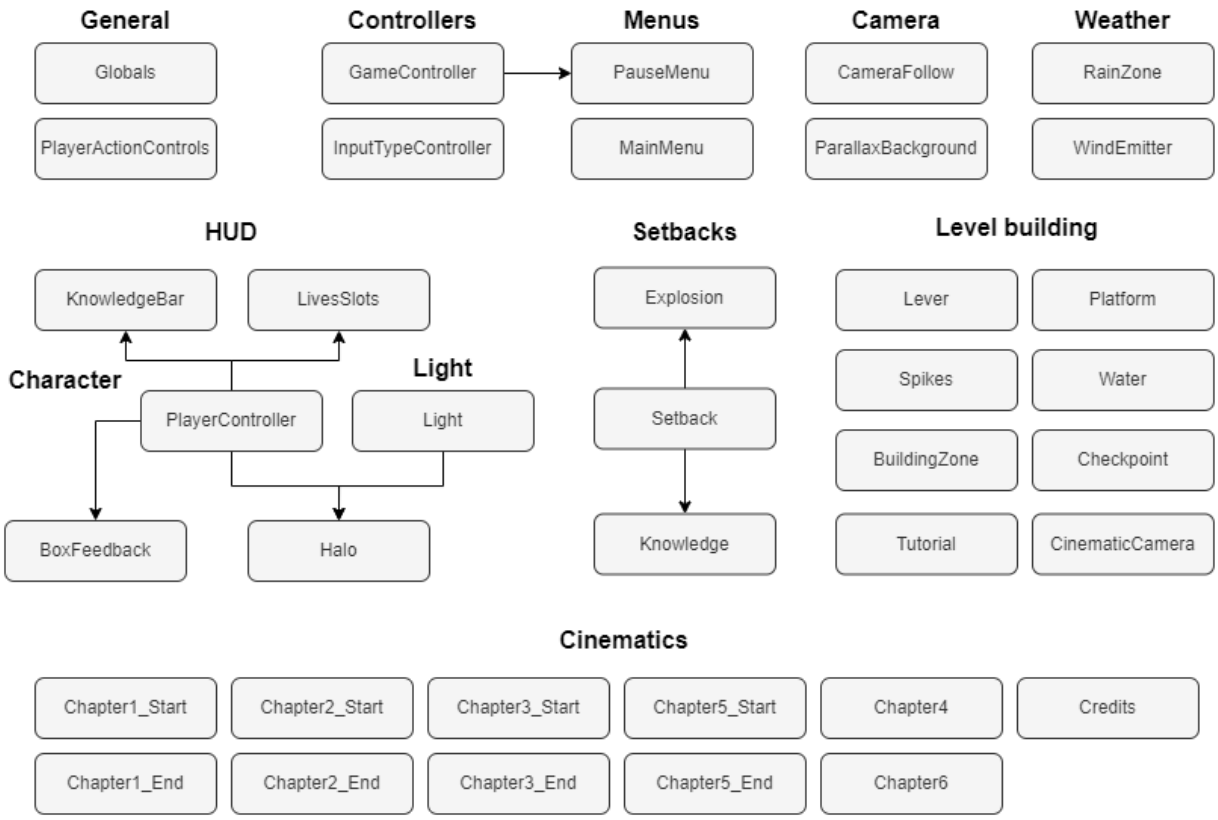

Figure 6.1. Class diagram and structure of the project

6.2. File structure

The project's file organisation follows the most common pattern used for 2D games developed with Unity, which is to create, at least, the following four main folders: Prefabs, Scenes, Scripts and Sprites. Each main folder includes other subdirectories that distribute every asset according to their category. In addition, a folder to store the font Bodoni MT was added (see Figure 6.2).

Figure 6.2. Folder organization of the project

The folder Prefabs contains all the different prefabricated GameObjects and their respective children, components and parameters. Prefabs allow the user to create and store objects that are used several times throughout all scenes. The different values and components of the object can be edited separately for each instance of the prefab.

There are eight different Scenes that compose the full project: one for each of the six chapters or levels, one for the main menu and one for the game credits. The internal structure of each scene, as well as the transitions between them, will be discussed later on.

The folder Sprites stores all the different visual and artistic elements created for the game. Animation-related assets are also included in this folder.

All the scripts and other technical assets are held in the same folder. Each script will be commented in full detail when the object they affect is explained; however, there are two scripts which have an effect on the whole game:

- ***Globals.cs***: This script contains static variables and functions that can be accessed by every other script. As of now, there is only one global variable in the game, named InputType. This variable is an integer that stores the current input type the player is using, so as to adapt the interface depending on the controller used. If the integer's value is 0, the player is currently using mouse and keyboard; if the value is 1, the player is using a PlayStation controller and, if it is 2, the user is playing with an XBOX controller.

- ***PlayerActionControls.cs***: This script is made and updated after creating and editing the Input Action asset of the same name. This asset contains all the different input actions the player can do, as well as their associated bindings and control schemes.

  There are ten different actions the player can do, each with two bindings; one for keyboard and one for gamepad, be it a PlayStation or an XBOX controller. For each action several properties can be edited, such as the type of interaction; in case the button has to be pressed or held. The ten input actions include the general movement, running, climbing, crouching, interacting, using the Knowledge, protecting, interacting with the Light, underwater movement and pausing the game (see Figure 6.3).

  Control schemes are useful to detect the current input type or to change properties depending on the controller the player is using. For our project, there are three different control schemes; one for PC, one for PlayStation 4 and one for XBOX. Although the properties do not differ from one another, having three separate control schemes is needed in order to detect a change in the current input type. To see the final game controls and the binding for each action, see section 11.3.

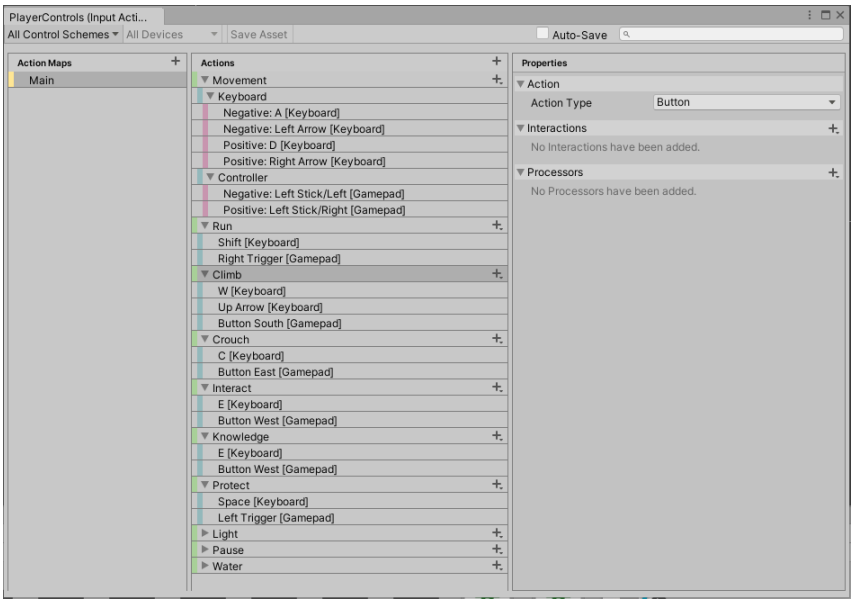

Figure 6.3. Input Action asset for WAVA

Unity incorporates a system of layering and tagging GameObjects, which is used to detect the object's type in triggers or when a collision occurs. Apart from the default ones, eight new tags and fifteen layers were added for this project. Provided that it has any importance or relevance, when defining an object, its tag and layer will be stated.

6.3. Scene structure

The scenes that correspond to the different levels or chapters follow a very similar hierarchy, with only a few variations from one to another. The overall structure for these scenes is (see Figure 6.4):

- Controller, which controls the game's behaviour, user input and cinematics.
- Character, which corresponds to the Mind, the user controlled character.
- Light, which corresponds to the Light, the AI controlled character. Due to the game's narrative, this object is exclusive to chapters one and two.
- Main Camera, which corresponds to the main rendering camera.
- Base, which stores general level objects, such as background elements or checkpoints.
- Zones, which distributes all the different world elements in several zones in order to facilitate the level building.
- Canvas, which corresponds to the user interface and stores the different images, menus and HUD elements.
- EventSystem, which is tied to the Canvas and is responsible for processing and handling events in the scene.

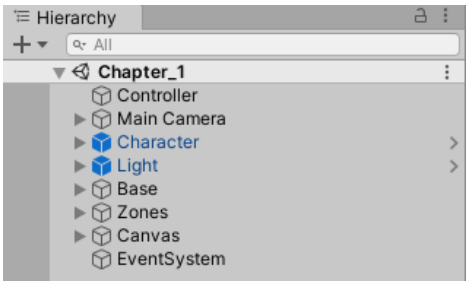

Figure 6.4. Scene structure and hierarchy of Chapter 1

In this section we will not analyse the main characters' GameObjects, as they will be defined in full detail on sections 6.4.1 and 0.

6.3.1. Controller

The GameObject Controller is present in every scene. It includes mainly two scripts; *GameController.cs* and *InputTypeController.cs*, the latter accompanied by a Player Input component. It also includes the scripts related to any start of level cinematics or, in chapters four and six, the full chapter animation.

Class GameController

The class defined in the script *GameController.cs* is in charge of controlling the general functioning of the game. As of now, the GameController's main purpose is to handle the game's pausing and unpausing.

For simplicity purposes, some of the actions the GameController could also be in charge of, such as managing the HUD or the respawn of the player, are controlled by the same player object (see section 6.4.1).

From now on, for all different classes we will list every variable and function, along with their type, visibility and a description of their behaviour. The length of this description will depend on the importance and complexity of every element.

- **Variables**
  - **`Chapter`** (`Integer/Public`): The current chapter.
  - **`PauseMenu`** (`GameObject/Public`): Pause menu's prefab in the scene.
  - **`Character`** (`GameObject/Public`): Character's prefab in the scene.
  - **`playerActionControls`** (`PlayerActionControls/Private`): Instance of our Input Action class.
  - **`paused`** (`Boolean/Private`): Is the game currently paused?

- **Functions**

  `Start` (`Void/Private`)

  Initializes the Character, hides the PauseMenu and sets the PauseGame function to be executed when the pause button is pressed. The expression to connect the method PauseGame with its respective input action from PlayerActionControls is:

```
playerActionControls.Main.Pause.performed += _ => PauseGame();
```

  `PauseGame` (`Void/Public`)

  Depending on whether the game is currently paused or not, toggles the character's activeness, and shows or hides the pause menu.

  `Awake/OnEnable/OnDisable` (`Void/Private`)

  Creates an instance of PlayerActionControls, enables and disables it, respectively.

Class InputTypeController

The class defined in the script *InputTypeController.cs* is in charge of detecting and handling changes on the type of controlling device the player is using. This class is mainly used to adapt the tutorial prompts according to the current controller used by the player.

- **Variables**
  - **`playerInput`** (`PlayerInput/Public`): A wrapper around the input system that takes care of managing input actions and players.

- **Functions**

  `OnEnable/OnDisable` (`Void/Private`)

  Subscribes and unsubscribes the method onInputDeviceChange when the user input is changed.

  `onInputDeviceChange` (`Void/Private`)

  Arguments: `InputUser, InputUserChange, InputDevice`

  Listens for changes in the control scheme being used. If a change is detected, changes the global variable InputType according to the current control scheme being used in the PlayerInput.

```
        void onInputDeviceChange(InputUser user, InputUserChange change, InputDevice
device)
        {
         if (change == InputUserChange.ControlSchemeChanged)
             {
                 switch (playerInput.currentControlScheme)
                 {
                     case "PC":
                         Globals.InputType = 0;
                         break;
                     case "PS4":
                         Globals.InputType = 1;
                         break;
                     case "XBOX":
                         Globals.InputType = 2;
                         break;
                   }
              }
        }
```

6.3.2. Camera

The main rendering camera in each scene uses most of Unity's default settings, except a slight increment of its size. Its hierarchy consists of two children GameObjects; the first one includes the scene's sky sprite, while the second one includes the two cinematic bars placed on both ends of the screen. In addition, the camera also includes a script component, named CameraFollow (see Figure 6.5 and Figure 6.6).

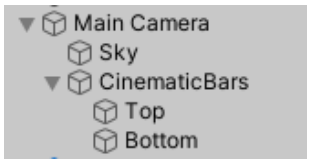

Figure 6.5. Hierarchy of the scene's main camera

Figure 6.6. Camera object in Unity's editor

Class CameraFollow

The class defined in the script *CameraFollow.cs* is in charge of moving the camera around in order to constantly follow and keep the player in frame along all playable chapters.

- **Variables**
  - **`target`** (`GameObject/Public`): The object to follow around, in our case, the main character.
  - **`offset`** (`Vector3/Public`): The camera position's offset in relation to the target.
  - **`smoothFactor`** (`Float/Public`): Multiplier to make the interpolation between camera positions less abrupt.
  - **`inCinematic`** (`Boolean/Private`): Is the camera in a cinematic?
  - **`xOffset/yOffset`** (`Float/ Private`): Initial horizontal and vertical camera position's offset.

- **Functions**

  `Start` (Void/Private)
  Initialize xOffset and yOffset according to the initial camera horizontal and vertical offset.

  `Update` (Void/Private)
  If the camera is not in a cinematic, move it to follow the target.

  `Follow` (Void/Private)
  Changes the horizontal and vertical offset of the camera depending on the direction the player is facing and its current vertical position. On the horizontal axis, the camera always shows a greater portion of what is in front of the player. As for the vertical axis, the higher the main character is, the lower the vertical offset's value is.
  These values are smoothly changed by using the function Lerp inside the Mathf library, which interpolates linearly between two values in a given time. For instance:

  ```
  offset.x = Mathf.Lerp(offset.x, xOffset, Time.deltaTime*smoothFactor/5);
  ```

  After both offsets are set, the camera position is interpolated to its new target position by also using the function Lerp. Due to our game's cinematic nature, creating a gentle camera movement is very important; hence all interpolations being regulated by the smoothFactor value.

  `toggleCinematic` (Void/Private)
  Switches the value of the inCinematic variable and changes the vertical offset depending on the target's current position.

  `getOffset` (Void/Private)
  Returns the current camera's position offset.

6.3.3. Background

All six chapter scenes, as well as the main menu, share the same background composition, which includes two layers of mountains, trees and clouds.

Class ParallaxBackground

The background elements appear livelier due to a parallax effect. This causes the different layers to move quicker the closer they are supposed to be from the camera, as well as repeating themselves infinitely. In order to do this infinite scrolling effect, the draw mode of the background sprites must be set to tiled. These effects are implemented in the script *ParallaxBackground.cs*.

- **Variables**
  - **`parallaxEffectMultiplier`** (`Vector2/Public`): The horizontal and vertical values of the parallax multiplier. These values range from zero to one and, the closer they are to one, the smaller the difference between the object and camera's movement is.
  - **`cameraTransform`** (`Transform/Public`): Main camera's transform component.
  - **`lastCameraPosition`** (`Vector3/Private`): Last position of the camera.
  - **`textureUnitSizeX/textureUnitSizeY`** (`Float/Private`): Texture's unit size, obtained by dividing the texture's width and height by the sprite's pixels per unit

- **Functions**

  `Start` (`Void/Private`)

  Initializes the camera's transform, its last position, and computes its texture unit size.

  `LateUpdate` (`Void/Private`)

  Calculates how much the camera has moved since the last frame and adds this value, multiplied by the parallax effect factors, in order to move the object. In addition, if the player has moved past the texture's unit size, the background element is relocated to a new position in order to create a seamless and infinite scrolling effect.

The three playable chapters' contain weather elements, which are an addition to the scene's overall background. The two weather types implemented for this game are rain and wind.

Class RainZone

Rain falls down through the whole of chapters one and two. In order to create an efficient rain effect, the rain is separated into different zones which are implemented in the script *RainZone.cs*. Each RainZone object has a BoxCollider2D component to detect the player and a particle system that constantly emits rain drops from the top of the screen until there are up to two thousand particles. Each one of these rain particles collides with all different environment elements, the main characters and the Setbacks. When coming into contact with these objects, a splash spritesheet animation is started and, once finished, the particle is deleted (see Figure 6.7).

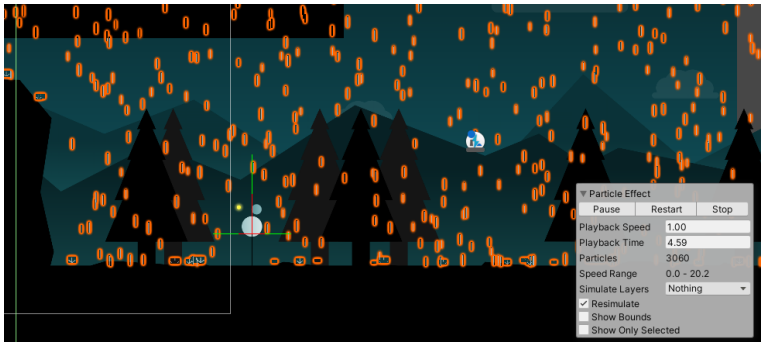

Figure 6.7. RainZone prefab in Unity's editor

- **Variables**
  - **`Character`** (`GameObject/Public`): Character's prefab in the scene.
  - **`isActive`** (`Boolean/Public`): Is the character inside the collider and should the particle system emit rain drops?
  - **`ps`** (`ParticleSystem/Private`): The rain zone's particle system.
  - **`em`** (`ParticleSystem.EmissionModule/Private`): The rain zone's particle system emission module.

- **Functions**

  `Awake` (`Void/Private`)

  Initialize the particle system and its emission module.

  `Start` (`Void/Private`)

  Finds the Character in the scene by its tag and enables the particle system's emission module if the rain zone should start as active.

  `OnTriggerEnter2D` (`Void/Private`)

  Arguments: `Collider2D`

  If the Collider2D is tagged as "Player" and it has entered the rain zone, it enables the emission of the particles.

`OnTriggerExit2D` `(Void/Private)`
Arguments: `Collider2D`
If the Collider2D is tagged as “Player” and it has exited the rain zone, it disables the emission of the particles.

Class WindEmitter

Along the fifth chapter, five gusts of wind are emitted and swirl across the screen every few seconds. This wind effect is made possible by a particle system component and a simple method implemented in the script *WindEmitter.cs* (see Figure 6.8).

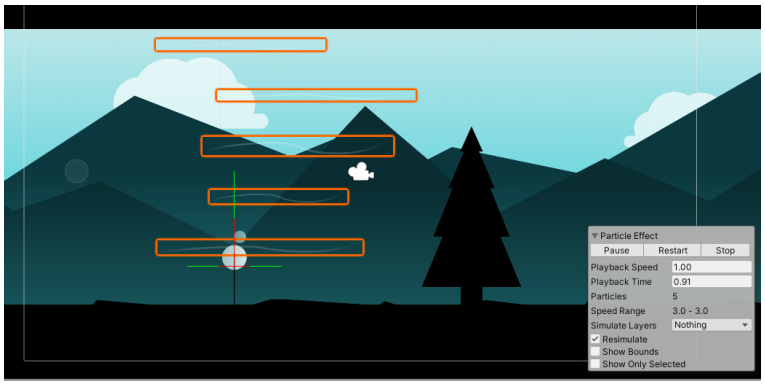

Figure 6.8. WindEmitter prefab in Unity's editor

- **Variables**
  - **`mainCamera`** `(Camera/Public)`: The scene’s rendering camera.
  - **`ps`** `(ParticleSystem/Private)`: The wind emitter’s particle system.

- **Functions**

  `Start` `(Void/Private)`
  Gets the particle system component and sets the function EmitWind to execute every five seconds.

  `EmitWind` `(Void/Private)`
  Moves the emitter to the main camera's position, with a slight offset to the left, and plays the particle system’s animation.

6.3.4. Canvas

All the scenes include a Canvas object, which stores the different user interface elements. This object is scalable with screen size, with a reference resolution of 1920 x 1080, so as to preserve the UI's composition and aspect in different screens.

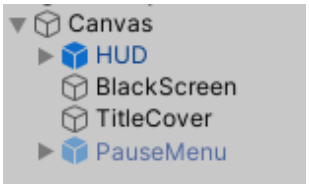

Figure 6.9. Hierarchy of the Canvas in a chapter scene

The scenes for the six chapters have an identical hierarchy of the Canvas, having as children the following elements (see Figure 6.9):

- The HUD, which will be explained in full detail in section 6.6.3.
- A black screen, which is faded in and out in order to create smoother transitions between chapters.
- A cover with the chapter's number and title, which is shown at the start of the scene.
- The pause menu, which will be explained in full detail in section 6.6.2.

6.4. Objects

Several GameObjects, along with their respective components and default values, were saved as prefabs in order to use them in various scenes.

6.4.1. Character

The main character, the Mind, is the most complex GameObject and contains the largest script, named *PlayerController.cs*, which includes a wide array of variables and functions.

Figure 6.10. Hierarchy of the Character prefab

The prefab has three children objects: a Halo prefab (see section 6.4.4) and two trigger BoxCollider2Ds that represent the positions where the auxiliary platform can be built (see Figure 6.10). In Figure 6.11 and Figure 6.12 we can see the Character prefab's appearance in Unity's editor and in-game, respectively.

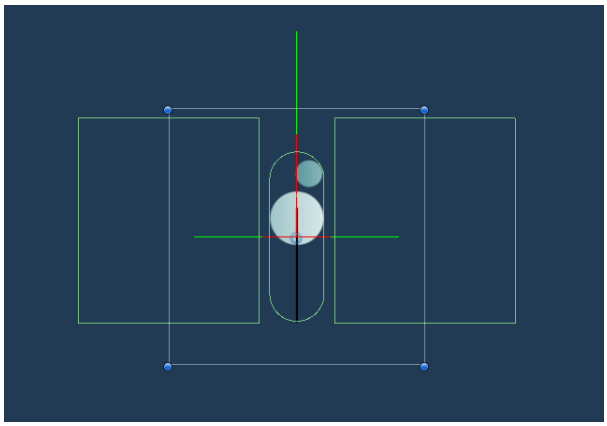

Figure 6.11. Character prefab in Unity's editor

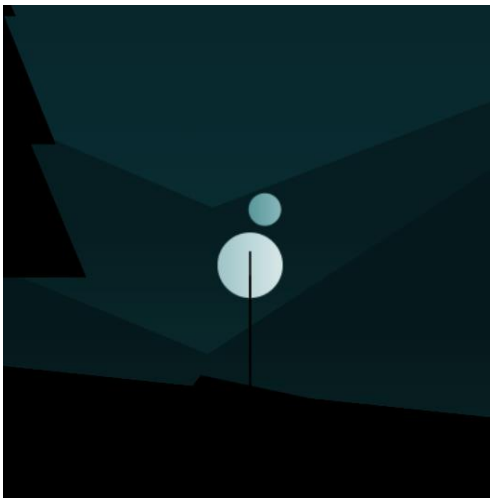

Figure 6.12. Character prefab in-game

The principal object is tagged as Player, layered as Character and contains the following components:

- SpriteRenderer, which renders the current animation's frame.
- CapsuleCollider2D, which provokes the character to collide with the environment.
- Rigidbody2D (Dynamic), which causes the character to be affected by physics and other forces.
- Animator, which controls the different character animations. There are ten animations or states that transition from one to the other based on the values of seven different parameters. Apart from these, eight additional separate states are also included (see Figure 6.13 and Figure 6.14).

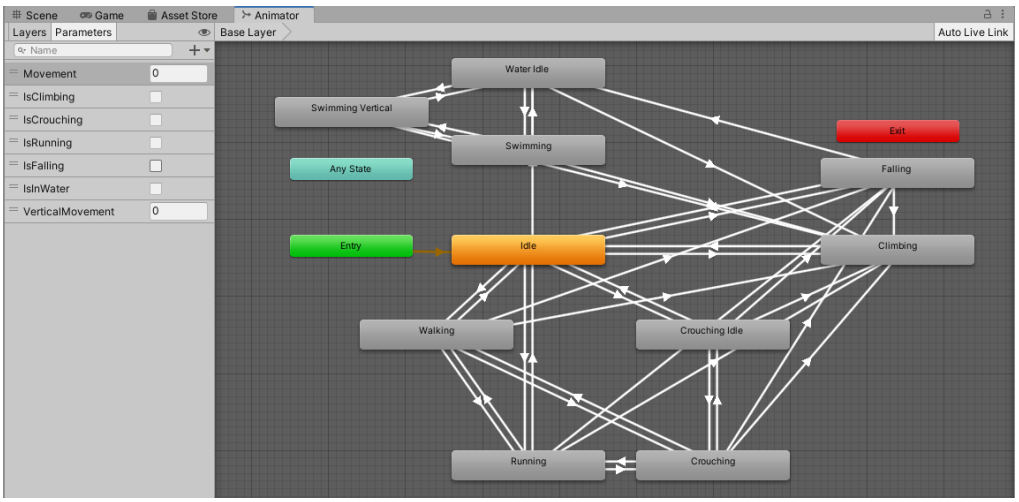

Figure 6.13. Character's animator controller states, transitions and parameters

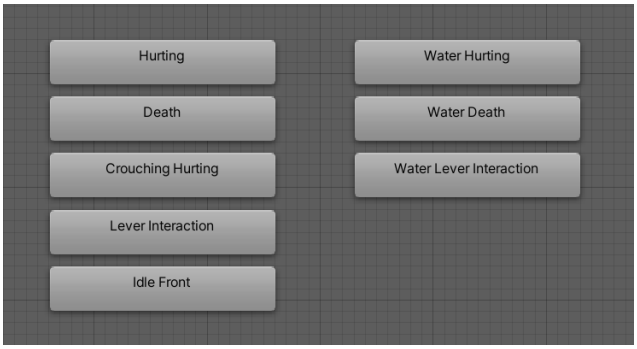

Figure 6.14. Character's animator controller separate

- **Variables**

Since the number of variables is quite considerable, they have been split into different groups depending on their type.

Controls:

- **playerActionControls** (PlayerActionControls/Private): Instance of our Input Action class.

Object components:

- **rb** (Rigidbody2D/Private): The character's rigid body component.
- **col** (CapsuleCollider2D/Private): The character's collider component.
- **sr** (SpriteRenderer/Private): The character's sprite renderer component.
- **anim** (Animator/Private): The character's animator component.

Layer masks:

- **climbLayerMask** (LayerMask/Private): GameObjects with the layers included in this mask can be climbed on.
- **crouchLayerMask** (LayerMask/Private): GameObjects with the layers included in this mask can be crouched through.
- **moveLayerMask** (LayerMask/Private): GameObjects with the layers included in this mask stop the player's movement.
- **groundLayerMask** (LayerMask/Private): GameObjects with the layers included in this mask can be walked on.
- **buildLayerMask** (LayerMask/Private): GameObjects with the layers included in this mask impede the character from building an auxiliary platform.

General values:

- **speed** (Float/Public): The character's movement speed.
- **rayLength** (Float/Public): The default length of the ray when ray casting.
- **knowledge** (Float/Public): The current amount of Knowledge gathered by the player.
- **knowledgeForBox** (Float/Private): Knowledge orbs that need to be collected in order to build an auxiliary platform.
- **lives** (Integer/Private): The current character's lives.

Checkpoints:

- **initialPosition** (Vector3/Private): The character's start position in the current scene.
- **lastCheckpoint** (Vector3/Private): The position of the last reached checkpoint.

User interface:

- **knowledgeBar** (KnowledgeBar/Public): Reference to the HUD's Knowledge slider.
- **livesSlots** (LivesSlots/Public): Reference to the HUD's lives slots.

References to objects and prefabs:

- **theHalo** (GameObject/Public): Reference to the character's Halo child object.
- **theLight** (GameObject/Public): Reference to the Light's prefab in the scene.
- **characterPlatform** (GameObject/Public): Reference to the auxiliary platform's prefab that can spawned in building zones.
- **boxFeedback** (GameObject/Public): Reference to the prefab that is spawned when an auxiliary platform cannot be built.
- **knowledgeBox** (GameObject/Private): Reference to the auxiliary platform that is currently placed in the scene.
- **rightBox/leftBox** (Transform/Public): The transform components of the character's child BoxCollider2D triggers, which represent the places where the auxiliary platform can be spawned.

References to currently interactable world objects:

- **climbablePlatform** (GameObject/Private): Reference to the platform object that is currently within reach and can be climbed onto.
- **interactableLever** (GameObject/Private): Reference to the lever object that is currently within reach and can be interacted with.
- **pickableKnowledge** (GameObject/Private): Reference to the Knowledge object that is within reach and can be picked up.

The Light:

- **hasLight** (Boolean/Public): Is the Light present in the current scene?
- **isInteractingWithLight** (Boolean/Private): Is the character interacting with the Light?

General boolean behaviour control:

- **isCurrentlyActive** (Boolean/Private): Can the user control the character right now?
- **isClimbing** (Boolean/Private): Is the player climbing a platform?
- **isFalling** (Boolean/Private): Is the player falling down?
- **isCrouching** (Boolean/Private): Is the player crouching?
- **isRunning** (Boolean/Private): Is the player running?
- **isProtecting** (Boolean/Private): Is the player protecting itself?
- **isDying** (Boolean/Private): Is the dying animation being played?

- **isInWater** (Boolean/Private): Is the player swimming underwater?
- **isInCinematic**(Boolean/Private): Is the player in a cinematic?
- **facingLeft** (Boolean/Private): Is the player facing towards the left of the screen?
- **canClimbLeft/canClimbRight** (Boolean/Private): Is there a platform within reach on the left/right side of the player?
- **canProtect** (Boolean/Public): Is the player able to protect itself currently?

Auxiliary platform:

- **boxPlaced** (Boolean/Private): Is there an auxiliary platform in the scene that can be recovered by the player?
- **inBuildingZone** (Boolean/Private): Is the player inside a building zone?

- **Functions**

  Awake (Void/Private)

  Creates a PlayerActionControls instance and initializes the object's components.

  OnEnable/OnDisable (Void/Private)

  Enables and disables the PlayerActionControls instance.

  Start (Void/Private)

  Called before the first frame update. Initializes all layer masks, updates the heads up display, saves the start position as the first checkpoint and connects a method of the class to a specific input action.

```
void Start()
{
    climbLayerMask = LayerMask.GetMask("Platform", "Tunneled Platform");
    moveLayerMask = LayerMask.GetMask( "Platform", "Tunneled Platform", "Environment",
                                       "Tunnel", "Knowledge", "Lever", "Ground");
    buildLayerMask = LayerMask.GetMask("Platform", "Tunneled Platform", "Environment",
                                       "Tunnel", "Knowledge", "Lever", "Ground",
                                       "Build Block");
    crouchLayerMask = LayerMask.GetMask("Tunneled Platform", "Tunnel");
    groundLayerMask = LayerMask.GetMask("Ground", "Platform", "Tunneled Platform",
                                        "Environment");

    […]

    playerActionControls.Main.Run.started += _ => StartRunning();
    playerActionControls.Main.Run.canceled += _ => StopRunning();
    playerActionControls.Main.Protect.performed += _ => Protect();
    playerActionControls.Main.Crouch.performed += _ => Crouch();
    playerActionControls.Main.Climb.performed += _ => Climb();
    playerActionControls.Main.Interact.performed += _ => Interact();
    playerActionControls.Main.Knowledge.performed += _ => UseKnowledge();
    playerActionControls.Main.Light.performed += _ => LightInteraction()
}
```

Update (Void/Private)
Called once per frame. If the player is active and not climbing, reads the value of the user's movement input and moves the character. If the character is underwater, does the same for the underwater movement input. On the other hand, if the player is on solid ground, detects if it is falling.

```
void Update()
{
    if (isCurrentlyActive)
    {
        if (!isClimbing)
        {
            float movementInput=playerActionControls.Main.Movement.ReadValue<float>();
            MoveCharacter(movementInput);

            if (isInWater)
            {
                float waterMovementInput=playerActionControls.Main.Water.ReadValue<float>();
                WaterMoveCharacter(waterMovementInput, movementInput);
            }

            else
                DetectFalling();
        }
    }
}
```

MakeActive (Void/Public)
Sets the variable isCurrentlyActive to true.

MakeInactive (Void/Public)
Sets the variable isCurrentlyActive to false and sets the animator's movement parameter's value to zero so as to transition to an idle state.

IsActive (Boolean/Public)
Returns the variable isCurrentlyActive.

MoveCharacter (Void/Private)
Arguments: Float
Changes the direction the player is facing depending on the value of the float argument's value; if the movementInput is negative, the character faces left; else, it faces right. If the player can move, changes its current horizontal position. Also, updates the necessary animator parameters' values.

```
private void MoveCharacter(float movementInput)
{
    […]

    if (CanMove(movementInput))
    {
       anim.SetFloat("Movement", Mathf.Abs(movementInput));
       if (movementInput < -0.1 || movementInput > 0.1)
       {
           Vector3 currentPosition = transform.position;
           currentPosition.x += movementInput * speed * Time.deltaTime;
           transform.position = currentPosition;
       }
    }

    […]
}
```

CanMove (Boolean/Private)

Arguments: `Float`

Checks the character's ability to move based on the movementInput argument's value. Emits a ray in each horizontal direction to detect objects belonging to the moveLayerMask. If the character is crouching and a collision is found in its way, the collision's layer is checked in case it is also included in the crouchLayerMask.

```
private bool CanMove(float movementInput)
{
   […]

   RaycastHit2D hitL = Physics2D.Raycast(transform.position, Vector2.left, length,
                                         moveLayerMask);
   RaycastHit2D hitR = Physics2D.Raycast(transform.position, Vector2.right, length,
                                         moveLayerMask);

    if ((hitL.collider != null && movementInput < 0) ||
        (hitR.collider != null && movementInput > 0))
    {
        if (isCrouching)
        {
            bool isCrouchable;
            if (hitL.collider != null)
              isCrouchable=((crouchLayerMask.value & (1 << hitL.collider.gameObject.layer))
                            > 0);
            else
               […]
            return isCrouchable;
        }

        return false;
     }

     return true;
}
```

`WaterMoveCharacter` `(Void/Private)`
Arguments: `Float, Float`
Similar to the regular MoveCharacter function; however, only moves the character in the vertical axis. Depending on the vertical and horizontal movement input, the character's sprite has to be flipped in different directions (see Figure 6.15).

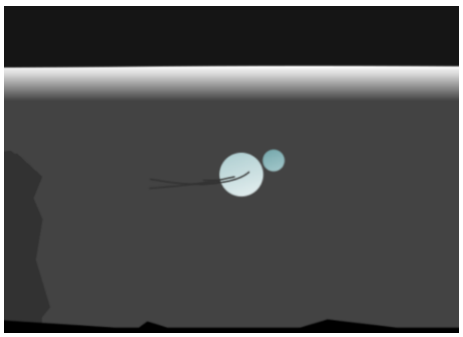

Figure 6.15. Character moving through water in-game

`WaterCanMove` `(Boolean/Private)`
Arguments: `Float, Float`
Similar to the regular CanMove function; however, only emits the rays in the vertical axis.

`StartRunning` `(Void/Private)`
Called when the Running input action's button is pressed. Emits a ray upwards to look for environment elements blocking the player. If nothing is detected, changes the character's speed and updates the necessary variables and animator's parameters (see Figure 6.16 and Figure 6.17).

Figure 6.16. Spritesheet of the character's running animation

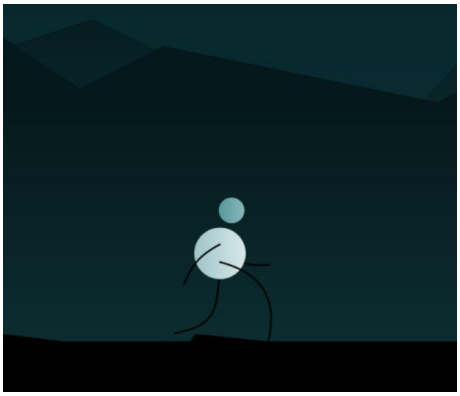

Figure 6.17. Character running in-game

`StopRunning` `(Void/Private)`
Called when the Running input action's button is released. Changes the character's speed and updates the necessary variables and parameters.

`DetectFalling` `(Void/Private)`
Emits a ray downwards to look for ground objects. If nothing is detected, updates the necessary variables and animator's parameters (see Figure 6.18).

Figure 6.18. Spritesheet of the character's falling animation

`faceLeft` `(Void/Private)`
Flips the character's sprite to the left and updates the variable isFacingLeft.

`faceRight` `(Void/Private)`
Flips the character's sprite to the right and updates the variable isFacingLeft.

`isFacingLeft` `(Boolean/Public)`
Returns the variable isFacingLeft.

`Interact` `(Void/Private)`
Called when the Interact input action's button is pressed. If the pickableKnowledge is not null, picks the Knowledge up. Alternatively, if the interactableLever is not null and it is active, the lever is activated.

```
private void Interact()
{
    if (isCurrentlyActive && !isFalling)
    {
         if (pickableKnowledge != null)
         {
            pickableKnowledge.GetComponent<Knowledge>().PickUp();
         }
         else if (interactableLever != null)
         {
             if (interactableLever.GetComponent<Lever>().isCurrentlyInteractable())
             {
                 if (isCrouching) Crouch();
                 interactableLever.GetComponent<Lever>().Activate();
                 StartCoroutine(LeverInteraction());
             }
         }
[…]
```

`setPickableKnowledge` `(Void/Public)`
Arguments: `GameObject, Boolean`
Sets the GameObject argument as the pickableKnowledge if the boolean argument is true; null otherwise (see Figure 6.19).

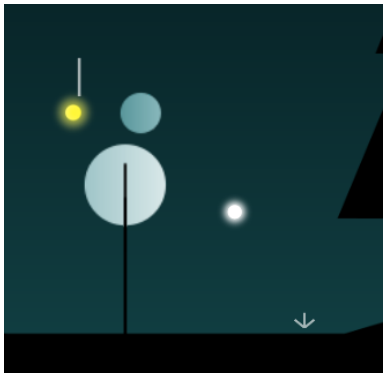

Figure 6.19. Character with pickable Knowledge in-game

`setInteractableLever` `(Void/Public)`
Arguments: `GameObject, Boolean`
Sets the GameObject argument as the interactableLever if the boolean argument is true; null otherwise.

`LeverInteraction` `(IEnumerator/Private)`
Makes the character inactive and moves next to the lever. Sets the animator to play the LeverInteraction state, stops for a few seconds and then goes back to its Idle state. If the lever does not stop the player while its associated object is moving, the character is made active again (see Figure 6.20 and Figure 6.21).

Figure 6.20. Spritesheet of the character's lever interaction animation

Figure 6.21. Character interacting with lever in-game

`Crouch` `(Void/Private)`

Called when the Crouch input action's button is pressed. Emits a ray upwards to look for environment elements blocking the player. If nothing is detected and the player is crouching, toggles the player to not crouch anymore. Otherwise, sets the character to be crouching (see Figure 6.22 and Figure 6.23).

```
private void Crouch()
{
    if (isCurrentlyActive && !isClimbing && !isInWater)
    {
        RaycastHit2D hitUp = Physics2D.Raycast(transform.position, Vector2.up, rayLength,
                                               moveLayerMask);

        if (isCrouching && hitUp.collider == null)
        {
            isCrouching = false;
            anim.SetBool("IsCrouching", false);
            if (isRunning) speed = 8;
            if (hasLight) theLight.GetComponent<Light>().toggleIsCrouching(false);
        }

        else
        {
            if (isRunning) StopRunning();
            isCrouching = true;
            anim.SetBool("IsCrouching", true);
            speed = 4;
            if (hasLight) theLight.GetComponent<Light>().toggleIsCrouching(true);
        }
    }
}
```

Figure 6.22. Spritesheet of the character's crouching idle and moving animations

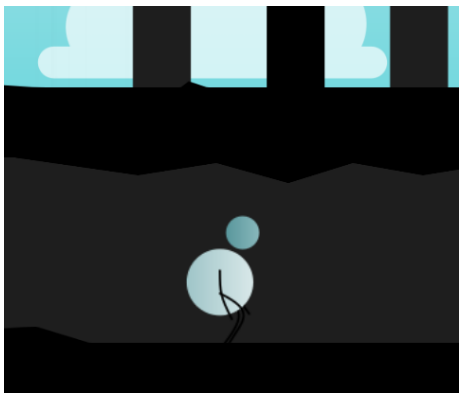

Figure 6.23. Character crouching in-game

`Climb` (Void/Private)

Called when the Climb input action's button is pressed. Detects nearby climbable platforms and, if one is found, prepares the character to climb by updating the necessary variables and animator's parameters and moving it next to the platform (see Figure 6.24 and Figure 6.25).

Figure 6.24. Spritesheet of the character's climbing animation

```
private void Climb()
{
   if (isCurrentlyActive && !isClimbing && !isFalling)
   {
       DetectClimbablePlatforms();

       if (canClimbLeft || canClimbRight)
       {
          if (isProtecting) theHalo.GetComponent<Halo>().HideHalo(true);
          if (isCrouching) Crouch();

          isClimbing = true;
          anim.SetFloat("Movement", 0);
          anim.SetBool("IsClimbing", true);

          rb.isKinematic = true;
          rb.velocity = Vector2.zero;

          if (canClimbLeft)
          {
             faceLeft();
             transform.position =
              climbablePlatform.GetComponent<Platform>().getDownRightPosition();
          }
          else if (canClimbRight)
          {
              faceRight();
              transform.position =
               climbablePlatform.GetComponent<Platform>().getDownLeftPosition();
          }
 […]
```

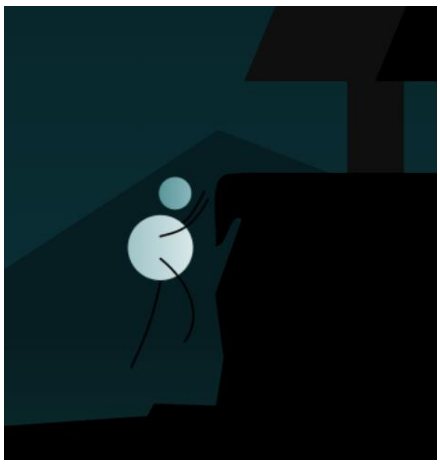

Figure 6.25. Character climbing in-game

StartClimb (Void/Private)
Called as an event at the first frame of the Climbing animation. Changes the character's position in order to adapt to the increment of size of the climbing spritesheet.

FinishClimb (Void/Private)
Called as an event at the last frame of the Climbing animation. Updates the necessary variables and animator parameters and moves the character on top of the platform.

DetectClimbablePlatforms (Void/Private)
Emits a ray in each horizontal direction to detect objects belonging to the climbLayerMask. In addition, emits a ray upwards to detect possible environment blockings.

```
private void DetectClimbablePlatforms()
{
    […]

    RaycastHit2D hitL=Physics2D.Raycast(transform.position,Vector2.left,length,climbLayerMask);
    RaycastHit2D hitR=Physics2D.Raycast(transform.position,Vector2.right,length,climbLayerMask);
    RaycastHit2D hitUp=Physics2D.Raycast(transform.position,Vector2.up,rayLength*2,moveLayerMask);

    if (hitL.collider != null && (hitUp.collider == null || isInWater))
    {
       if (pickableKnowledge==null || pickableKnowledge.transform.position.x>transform.position.x)
       {
           canClimbLeft = true;
           climbablePlatform = hitL.collider.gameObject;
       }
     }

     else if (hitR.collider != null && (hitUp.collider == null || isInWater))
     {
       if (pickableKnowledge==null || pickableKnowledge.transform.position.x<transform.position.x)
       {
           canClimbRight = true;
           climbablePlatform = hitR.collider.gameObject;
       }
     }

     else
     {
        canClimbLeft = canClimbRight = false;
        climbablePlatform = null;
      }
}
```

FullKnowledge (Boolean/Public)
Returns true if the player has the necessary Knowledge to build a platform.

AddKnowledge (Void/Public)
Adds a Knowledge unit and updates the HUD's Knowledge slider.

UseKnowledge (Void/Private)

Called when the Knowledge input button is pressed and held. If an auxiliary is placed and the character is in a building zone, that box is recovered and the HUD is updated, as well as the necessary variables. On the other hand, if an auxiliary is not placed and the character has full Knowledge, two rays are emitted in order to detect objects that may block the auxiliary platform that will be built. If there are no blockings in front of the character or on top of it, an auxiliary platform is instantiated and the HUD is updated, as well as the necessary variables. However, if the platform cannot be built, a CharacterPlatformFeedback prefab is instantiated (see Figure 6.26).

```
private void UseKnowledge()
{
   if (boxPlaced)
   {
      if (isCurrentlyActive && inBuildingZone)
      {
         Destroy(knowledgeBox);
         boxPlaced = false;
         knowledge = knowledgeForBox;
         knowledgeBar.SetKnowledge(knowledge);
       }
    }

    else
    {
       if (FullKnowledge() && isCurrentlyActive && inBuildingZone)
       {
          if (facingLeft)
          {
              RaycastHit2D hitUp = Physics2D.Raycast(leftBox.position, Vector2.up,
                                                     rayLength * 2, buildLayerMask);
              RaycastHit2D hitL = Physics2D.Raycast(transform.position, Vector2.left,
                                                     rayLength * 1.5f, buildLayerMask);

              if (hitL.collider == null && hitUp.collider == null)
              {
                  knowledgeBox=Instantiate(characterPlatform,leftBox.position,
                                           Quaternion.identity);
                  boxPlaced = true;
                  knowledge = 0f;
                  knowledgeBar.SetKnowledge(knowledge);
               }

               else
               {
                  Instantiate(boxFeedback, leftBox.position, Quaternion.identity);
               }
          }

          else
          {
             […]
```

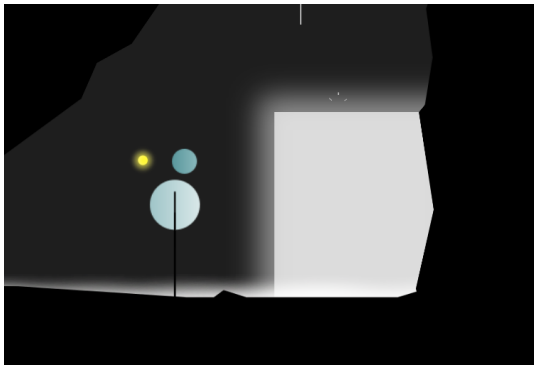

Figure 6.26. Character using the Knowledge in-game

`changeInBuildingZone` `(Void/Public)`
Arguments: `Boolean`
Sets the character to be in a building zone or not, depending on the boolean argument's value.

`resetBox` `(Void/Public)`
Sets the character to not have an auxiliary platform currently placed.

`LightInteraction` `(Void/Private)`
Called when the Light input action's button is pressed. If the Light is present in the current scene, the Light's interact function is executed.

`playerHasLight` `(Boolean/Public)`
Returns true if the Light is present in the current scene.

`changeIsInteractingWithLight` `(Void/Public)`
Arguments: `Boolean`
Sets the character to be interacting with the Light or not, depending on the boolean argument's value.

`Protect` `(Void/Private)`
Called when the Protect input action's button is pressed. If the character is currently able to protect itself and the Light is present, the Light's protect function is executed. On the other hand, if the character can protect itself but the Light is not in the scene, the Halo's protect function is called (see Figure 6.27).

```
private void Protect()
{
    if (canProtect && !isProtecting && isCurrentlyActive && !isClimbing)
    {
        if(hasLight)
           theLight.GetComponent<Light>().Protect();
        else
           theHalo.GetComponent<Halo>().Protect(false);
     }
}
```

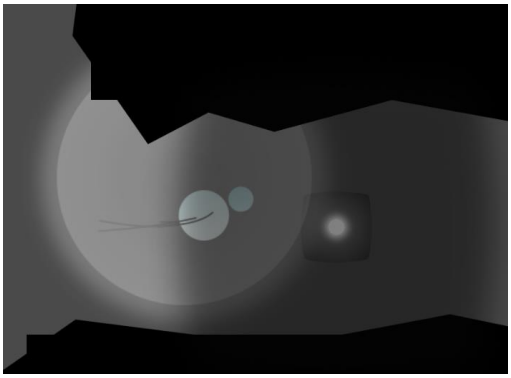

Figure 6.27. Character protecting itself in-game

`CinematicProtect` `(Void/Public)`
Called when the Protect input action's button is pressed and the character is in a cinematic. The Halo's protect function is called.

`changeCanProtect` `(Void/Public)`
Arguments: `Boolean`
Sets the character to be able to protect or not, depending on the boolean argument's value.

`changeIsProtecting` `(Void/Public)`
Arguments: `Boolean`
Sets the character to be currently protecting or not, depending on the boolean argument's value.

`Protecting` `(Boolean/Public)`
Returns true if the character is currently protecting itself.

`GetHit` `(Void/Public)`
Arguments: `Integer, String`
Takes out the number of lives defined by the integer argument and updates the HUD. If the character's lives are larger than zero, the Hurting co-routine is called. Alternatively, the Death co-routine is executed.
The string argument defines the cause of damage, which can be: "light", if the light has entered a body of water; "spikes", if the character has fallen onto a spike trap; and "left" or "right", depending on the direction from which the Setback has attacked the player (see Figure 6.28).

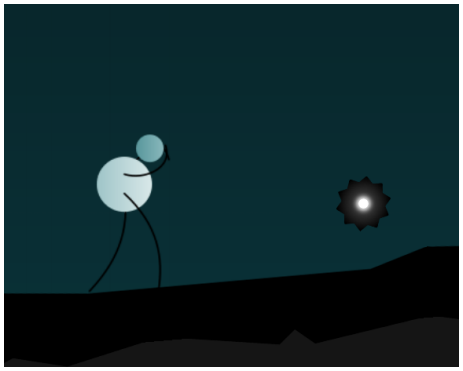

Figure 6.28. Character getting hit in-game

Death (IEnumerator/Private)
Arguments: String
Makes the player inactive and, if the string argument is “light”, stalls for two seconds. Plays the death animation and, after a three second pause, respawns the player (see Figure 6.29).

Figure 6.29. Spritesheet of the character's death animation

```
IEnumerator Death(string from)
{
    MakeInactive();
    changeIsInCinematic(true);

    sr.flipY = false;

    isDying = true;

    if (from == "light")
    {
        theLight.GetComponent<Light>().Die();
        yield return new WaitForSeconds(2f);
    }

    if (!isInWater) anim.Play("Death");
    else anim.Play("Water Death");

    yield return new WaitForSeconds(3f);

    Respawn();
}
```

Hurting (IEnumerator/Private)
Arguments: String
Makes the player inactive and plays the hurting animation. If the string argument is “left” or “right”, applies a force on the character towards the opposite direction. Stalls for one second, plays the character’s idle animation and reactivates it (see Figure 6.30).

Figure 6.30. Spritesheet of the character's hurting animation

```
IEnumerator Hurting(string from)
    {
        MakeInactive();
        changeIsInCinematic(true);

        sr.flipY = false;

        if (isCrouching && !isInWater) anim.Play("Crouching Hurting");
        else if (!isCrouching && !isInWater) anim.Play("Hurting");
        else anim.Play("Water Hurting");

        if (from == "left") rb.AddForce(Vector3.right * 250f);
        else if (from == "right") rb.AddForce(Vector3.left * 250f);

        yield return new WaitForSeconds(1f);

        if(!isFalling) rb.velocity = Vector2.zero;

        if (!isDying)
        {
            if (isCrouching && !isInWater) anim.Play("Crouching Idle");
            else if (!isCrouching && !isInWater) anim.Play("Idle");
            else anim.Play("Water Idle");

            MakeActive();
            changeIsInCinematic(false);
        }
    }
```

`GetLives` `(Integer/Public)`
Returns the current number of lives.

`GetIsProtecting` `(Boolean/Public)`
Returns true if the character is currently protecting.

`UpdateCheckpoint` `(Void/Public)`
Arguments: `Vector3`
Sets the Vector3 argument's value as the position of the latest checkpoint reached by the player.

`Respawn` `(Void/Private)`
Resets the player to its idle state, refills its lives, moves it to the nearest checkpoint and updates the heads up display. If the Light is present, it is also respawned.

```
private void Respawn()
{
    […]

    if (hasLight)
    {
        theLight.transform.position = lastCheckpoint;
        theLight.GetComponent<Light>().Respawn();
    }

    transform.position = lastCheckpoint;

    lives = 5;
    livesSlots.SetLives(lives);

    anim.Play("Idle");
}
```

changeIsInWater (Void/Public)
Arguments: Boolean
Sets the character to be underwater or not, depending on the boolean argument's value (see Figure 6.31).

Figure 6.31. Spritesheet of the character's swimming animation

InWater (Boolean/Public)
Returns true if the character is currently underwater.

changeIsInCinematic (Void/Public)
Arguments: Boolean
Sets the character to be in a cinematic or not, depending on the boolean argument's value. If this boolean variable is set to true, it prevents the character from being wrongfully reactivated when the pause menu is closed by the player.

InCinematic (Boolean/Public)
Returns true if the character is currently in a cinematic.

OnDrawGizmos (Void/Public)
Draws the different rays emitted by the character in Unity's editor.

6.4.2. Light

The non-controllable main character, the Light, is implemented in the script *Light.cs* (see Figure 6.32 and Figure 6.34). The prefab only has one child object, a Halo prefab (see section 6.4.4).

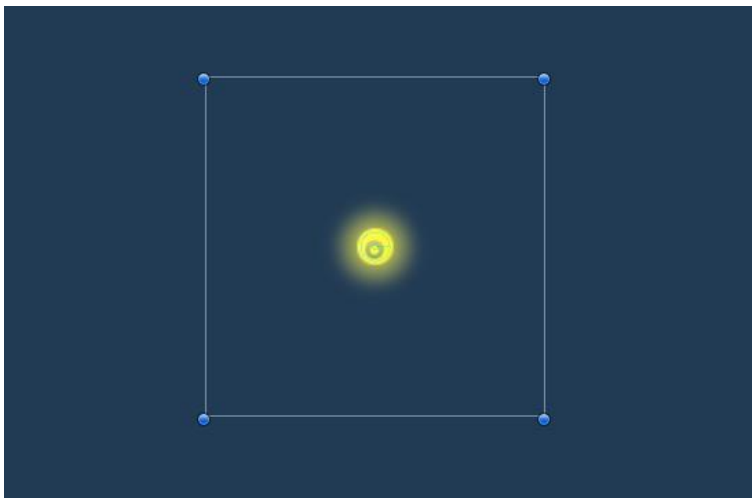

Figure 6.32. Light prefab in Unity's editor

The principal object is tagged and layered as Light and contains the following components:

- SpriteRenderer, which renders the current animation's frame.
- CircleCollider2D, which is a trigger that detects other GameObjects.
- Rigidbody2D (Kinematic), which is also needed to detect collisions.
- Animator, which controls the different character animations. There are three animations or states that transition from one to the other based on the values of two parameters. Apart from these, one additional separate state is also included (see Figure 6.33).

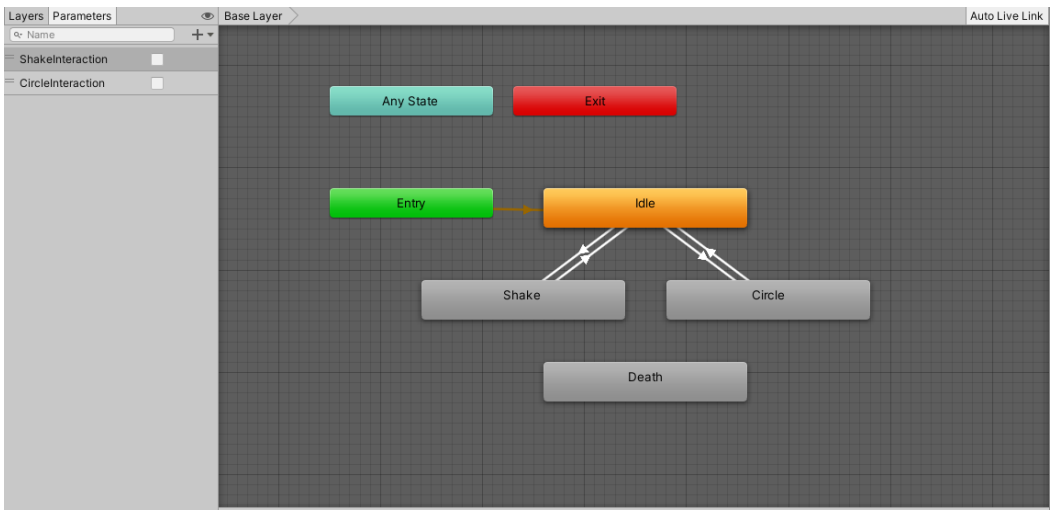

Figure 6.33. Light's animator controller states, transitions and parameters

- **Variables**
  - **`target`** (`GameObject/Public`): Reference to the object that the Light will follow around; in our case, the Character.
  - **`offset`** (`Vector3/Public`): Position offset in relation to the target.
  - **`smoothFactor`** (`Float/Public`): Multiplier to make Light's movement less abrupt.
  - **`xOffset`** (`Float/Private`): The horizontal value of the Light's offset.
  - **`anim`** (`Animator/ Private`): The Light's animator component.
  - **`sr`** (`SpriteRenderer/Private`): The Light's sprite renderer component.
  - **`isCurrentlyActive`** (`Boolean/Private`): Is the Light active and following the target?
  - **`isInteracting`** (`Boolean/Private`): Is the Light being interacted with?
  - **`isClimbing`** (`Boolean/Private`): Is the Light waiting for the character to climb?
  - **`timesInteracted`** (`Integer/Private`): Amount of times the character has interacted with the Light.
  - **`theHalo`** (`GameObject/Public`): Reference to the Light's Halo child object.

Figure 6.34. Light prefab in-game

- **Functions**

  `Awake` (`Void/Private`)
  Initializes the object's components.

  `Start` (`Void/Private`)
  Called before the first frame update. Finds the character by its tag and saves the horizontal offset.

  `Update` (`Void/Private`)
  Called once per frame. If the Light is currently active and the character is not climbing, executes the Follow function.

Follow (Void/Private)

Changes the horizontal position offset depending on the direction the character is facing. Then, moves smoothly to its new position based on the position of the target.

```
void Follow()
{
    if (target.GetComponent<PlayerController>().isFacingLeft())
    {
       offset.x = xOffset;
       if(!isInteracting) sr.flipX = true;
    }

    else
    {
        offset.x = -xOffset;
        if (!isInteracting) sr.flipX = false;
    }

    Vector3 targetPosition = target.transform.position + offset;
    Vector3 smoothPosition = Vector3.Lerp(transform.position, targetPosition,
                                          smoothFactor * Time.deltaTime);
    transform.position = smoothPosition;
}
```

Protect (Void/Public)

If the Light is currently active, calls its Halo's protect function (see Figure 6.35).

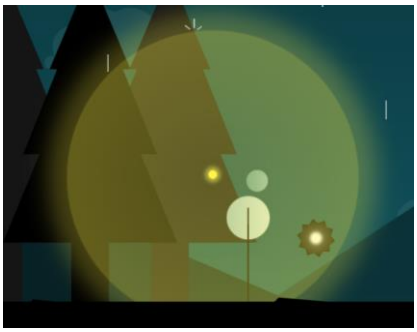

Figure 6.35. The Light protecting in-game

Interact (Void/ Public)

If the Light is currently active, sets itself as being interacted with and starts the DoInteraction co-routine.

DoInteraction (IEnumerator/Private)

Depending on the parity of the timesInteracted variable's value, does a shake animation or a circle animation. After a short delay, the Light goes back to its idle state and sets itself as interactable again.

```
IEnumerator DoInteraction()
{
        if(timesInteracted % 2 == 0)
            anim.SetBool("ShakeInteraction", true);
        else
            anim.SetBool("CircleInteraction", true);

        yield return new WaitForSeconds(1.5f);

        […]

        target.GetComponent<PlayerController>().changeIsInteractingWithLight(false);
        isInteracting = false;
 }
```

Die (Void/Public)
Calls the Death co-routine.

Death (IEnumerator/Private)
Makes itself inactive, plays its fading out animation and waits for a few seconds.

Respawn (Void/Public)
Sets itself as active and plays its idle state.

toggleIsCrouching (Void/Public)
Arguments: Boolean
Changes its vertical position offset depending on the boolean argument's value.

toggleIsClimbing (Void/Public)
Arguments: Boolean
Sets the itself as waiting, or not, for the character to climb depending on the boolean argument's value.

MakeInactive (Void/Public)
Sets the Light as inactive.

MakeActive (Void/Public)
Sets the Light as active.

6.4.3. Setbacks

The main enemies of the game, the Setbacks, are implemented in the script *Setback.cs*. There are two types of Setback, the kamikaze and the bomber. Due to their similarities, they share the same script but are saved as two different prefabs (see Figure 6.36). This prefab only has one child object, called Sprite, which stores the SpriteRenderer component.

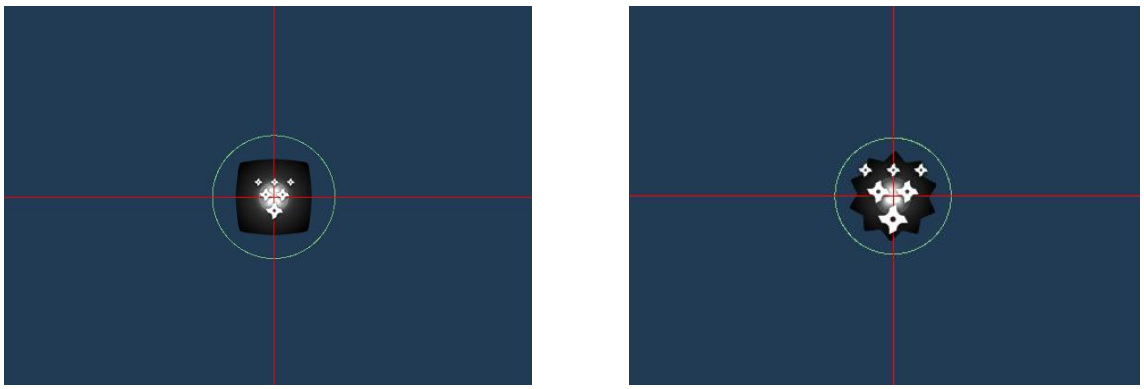

Figure 6.36. Setback (bomber and kamikaze) prefabs in Unity's editor

The principal object is tagged and layered as Setback and contains the following components:

- CircleCollider2D, which is a trigger that detects other GameObjects.
- ParticleSystem, which contains a particle animation that is played when the Setback is defeated.

- **Variables**
  - **`Character`** (`GameObject/Public`): Reference to the Character prefab in the scene.
  - **`attackLayerMask`** (`LayerMask/Private`): GameObjects with the layers included in this mask will be attacked.
  - **`ps`** (`ParticleSystem/Private`): The Setback's particle system component.
  - **`sr`** (`SpriteRenderer/Private`): The Setback's sprite renderer component.
  - **`Type`** (`Integer/Public`): The type of Setback; zero for bomber, one for kamikaze.
  - **`Sprite`** (`GameObject/Public`): The Sprite child object.
  - **`Knowledge`** (`GameObject/Public`): The Knowledge's prefab.
  - **`Explosion`** (`GameObject/Public`): The SetbackExplosion's prefab.
  - **`speed`** (`Float/Public`): The speed of the kamikaze Setback.
  - **`rayLength`** (`Float/Public`): The default length of the ray when ray casting.
  - **`rotation`** (`Float/Private`): The rotation of the kamikaze Setback.
  - **`isActive`** (`Boolean/Private`): Is the Setback still not defeated?

- **isAttacking** (Boolean/Private): Is the Setback currently attacking?
- **isReturningHome** (Boolean/Private): Is the Setback going back to its initial position?
- **shake** (Boolean/Private): True if the Setback has to shake when a player is detected; false if it should attack.
- **initialPosition** (Vector3/Private): The home position.

- **Functions**

  Start (Void/Private)

  Finds the Character in the scene by its tag, gets the SpriteRenderer and ParticleSystem components, saves its initial position and adds the layer "Character" to the attack layer mask.

  Update (Void/Private)

  If the character and setback are currently active and the latter is not attacking or returning home, the DetectPlayer function is called.

  DetectPlayer (Void/Private)

  Emits four rays that detect collisions on the attackLayerMask. If the character is detected on the left or right and shake is true, the Shake co-routine is called; alternatively, the Attack function is called. If the character is in water, it has been detected on the vertical axis and the Setback is a bomber, the Shake co-routine is called. However, if the player is not detected, the Setback returns home.

```
private void DetectPlayer()
{
    RaycastHit2D hitL = Physics2D.Raycast(transform.position, Vector2.left,
                                          rayLength, attackLayerMask);
    RaycastHit2D hitR = Physics2D.Raycast(transform.position, Vector2.right,
                                          rayLength, attackLayerMask);
    RaycastHit2D hitUp = Physics2D.Raycast(transform.position, Vector2.up,
                                           rayLength, attackLayerMask);
    RaycastHit2D hitDown = Physics2D.Raycast(transform.position, Vector2.down,
                                             rayLength, attackLayerMask);

    if (hitL.collider != null)
    {
        rotation = 600f;
        isAttacking = true;

        if(shake)
            StartCoroutine(Shake());
        else
            Attack(hitL.collider.transform.position);
    }

    else if (hitR.collider != null)
    {

        […]
    }
```

```
        else if (hitUp.collider != null || hitDown.collider != null)
        {
             if(Type == 0 && Character.GetComponent<PlayerController>().InWater())
             {
                 isAttacking = true;

                 if (shake)
                      StartCoroutine(Shake());
              }
        }

        else
        {
              if (!shake)
              {
                   isAttacking = false;
                   isReturningHome = true;
                   StartCoroutine(GoBackHome());
        }
     }
}
```

Shake (IEnumerator/Private)

Shakes the Setback for a short time by using the sinus function in Mathf. The increments or decrements of position are only applied as long as the new position is not too far from the initial position. Once the shaking animation is over, it is checked if the player is still nearby.

```
IEnumerator Shake()
{
    […]

    float shakingSpeed = 0.1f;
    while (isActive && time < duration)
    {
        shakinessX = Random.Range(-0.05f, 0.05f);
        shakinessY = Random.Range(-0.05f, 0.05f);

        float shakeChangeX = Mathf.Sin(shakingSpeed * Time.time) * shakinessX;
        float shakeChangeY = Mathf.Sin(shakingSpeed * Time.time) * shakinessY;

        if (Mathf.Abs((shakingPosition.x + shakeChangeX) - initialPosition.x) < 0.1f)
            shakingPosition.x += shakeChangeX;
        if (Mathf.Abs((shakingPosition.y + shakeChangeY) - initialPosition.y) < 0.1f)
            shakingPosition.y += shakeChangeY;

        transform.position = shakingPosition;

        time += Time.deltaTime;

        yield return null;
     }

     yield return new WaitForSeconds(0.25f);

     shake = false;

     if (Character.GetComponent<PlayerController>().InWater() && Type == 0)
         StartCoroutine(Explode());
     else
         DetectPlayer();
}
```

`Attack` `(Void/Private)`

Arguments: `Vector3`

If the setback is a bomber, explodes. Alternatively, if it is a kamikaze, chases the target towards the position entered as an argument.

`Explode` `(IEnumerator/Private)`

Instantiates an explosion and waits for a few seconds before being able to attack again (see Figure 6.37).

```
IEnumerator Explode()
{
    yield return new WaitForSeconds(0.25f);
    if (isActive)
       Instantiate(Explosion, transform.position, Quaternion.identity);

    shake = true;

    yield return new WaitForSeconds(2f);

    isAttacking = false;
}
```

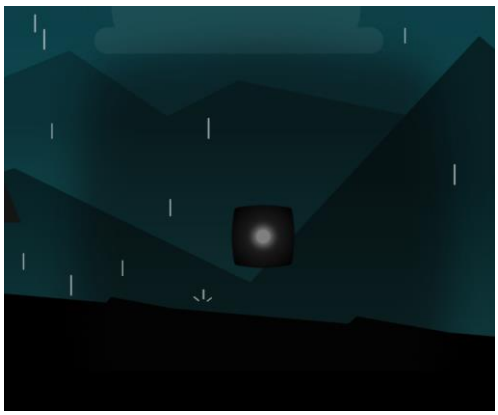

Figure 6.37. Bomber Setback emitting an explosion in-game

`Chase` `(IEnumerator/Private)`

Arguments: `Vector3`

Moves to the target position entered as an argument. In addition, rotates the object while moving in order to generate more dynamism (see Figure 6.38).

```
IEnumerator Chase(Vector3 target)
{
    while (isActive && transform.position != target && isAttacking)
    {
       Vector3 currentEulerAngles = transform.eulerAngles;
       currentEulerAngles.z += rotation * Time.deltaTime;
       transform.eulerAngles = currentEulerAngles;

       transform.position = Vector2.MoveTowards(transform.position, target,
                                                speed * Time.deltaTime);
       yield return null;
    }
    isAttacking = false;
}
```

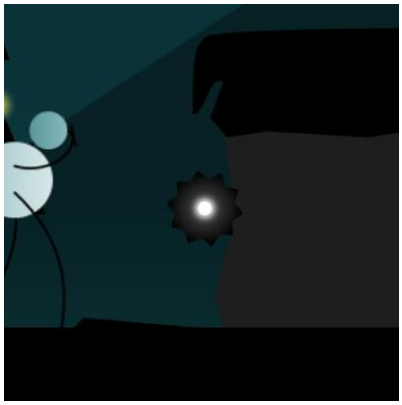

Figure 6.38. Kamikaze Setback attacking the player in-game

`GoBackHome` `(IEnumerator/Private)`
Returns to its initial position after having launched an attack.

`Die` `(IEnumerator/Private)`
Plays the particle system animation, instantiates a Knowledge orb and reduces its Sprite child object's size. Once the Sprite's size is zero, the Setback GameObject is destroyed.

```
IEnumerator Die()
{
    float duration = ps.main.duration + 3f;
    float time = 0;

    ps.Play();

    Instantiate(Knowledge, transform.position, Quaternion.identity);

    while (time < duration)
    {
        Sprite.transform.localScale = Vector3.Lerp(Sprite.transform.localScale,
                                        new Vector3(0, 0, 0), time / duration);
        time += Time.deltaTime;
        yield return null;
    }

    isActive = false;
    Destroy(this.gameObject);
}
```

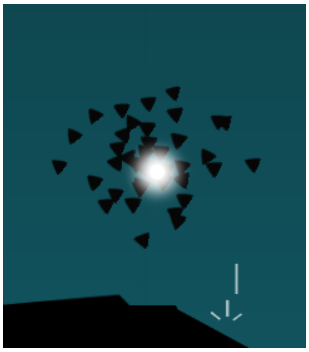

Figure 6.39. The Setback's death animation in-game

OnTriggerEnter2D (Void/Private)
Arguments: Collider2D
If the collider that has entered is tagged as Player, the character is hit and, if the Setback is a kamikaze, it returns to its initial position. On the other hand, if the collider is tagged as Halo, the Die co-routine is called. Finally, if the kamikaze Setback collides with a platform, it returns to its home.

```
void OnTriggerEnter2D(Collider2D col)
{
    if (isActive)
    {
        if (col.gameObject.tag == "Player")
        {
            string from;
            if (col.transform.position.x > transform.position.x) from = "left";
            else from = "right";

            Character.GetComponent<PlayerController>().GetHit(1, from);

            isAttacking = false;

            if (Type == 1)
            {
               isReturningHome = true;
               StartCoroutine(GoBackHome());
            }
          }

          else if (col.gameObject.tag=="Halo" &&
                    Character.GetComponent<PlayerController>().Protecting())
          {
              isAttacking = false;
              isActive = false;
              rotation = 0f;
              StartCoroutine(Die());
          }

          else if (col.gameObject.tag == "Platform" && Type == 1)
          {
              isAttacking = false;
              isReturningHome = true;
              StartCoroutine(GoBackHome());
          }
     }
}
```

OnDrawGizmos (Void/Public)
Draws the different rays emitted by the Setback in Unity's editor.

Explosion

The explosion emitted by the bomber Setback is implemented in the script *Explosion.cs*. The principal object is tagged and layered as Setback and contains the following components (see Figure 6.40):

- SpriteRenderer, which stores and displays the object's sprite.
- CircleCollider2D, which is a trigger that detects other GameObjects.

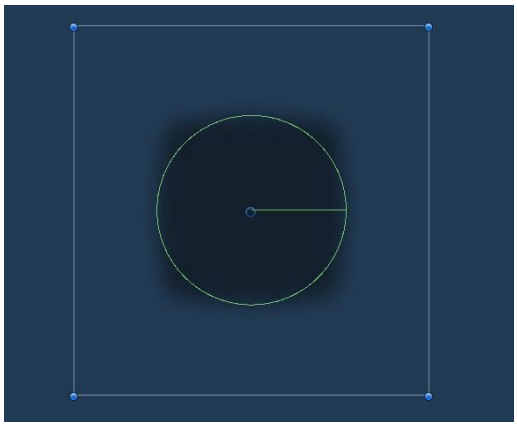

Figure 6.40. SetbackExplosion prefab in Unity's editor

- **Variables**
  - **`Character`** (`GameObject/Public`): Reference to the Character prefab in the scene.
  - **`sr`** (`SpriteRenderer/Private`): The object's sprite renderer component.
  - **`isExploding`** (`Boolean/Private`): Is the Setback currently attacking?
  - **`affectsPlayer`** (`Boolean/Private`): Is the explosion currently lethal to the player?

- **Functions**

  `Start` (`Void/Private`)

  Finds the Character in the scene by its tag, gets the SpriteRenderer component and calls the Exploding co-routine.

  `Exploding` (`IEnumerator/Private`)

  Increases itself until reaching its target size. Afterwards, fades itself out by decreasing its sprite's alpha value. As soon as the sprite is at 95% opacity, it stops affecting the player and, when it is completely transparent, the explosion object is destroyed.

```
IEnumerator Exploding()
{
   […]

   while (time < duration && isExploding)
   {
       transform.localScale = Vector3.Lerp(transform.localScale,
                              new Vector3(25f, 25f, 1f), Time.deltaTime / duration);
       time += Time.deltaTime;
       yield return null;
   }

   […]

     while (time < duration && isExploding)
     {
         sr.color = Color.Lerp(sr.color, new Color(1f, 1f, 1f, 0f),
                               (Time.deltaTime * 4f) / duration);
         time += Time.deltaTime;

         if(affectsPlayer && sr.color.a < 0.95f)
         {
             affectsPlayer = false;
         }

         yield return null;
     }

     […]
```

OnTriggerEnter2D (Void/Private)

Arguments: Collider2D

If the collider that has entered is tagged as Player, the character is not protected and the explosion currently affects the character, the character is hit.

6.4.4. Halo

The player protects itself from Setbacks with a semi-transparent bubble named Halo, which is implemented in a script with the same name. The Halo object is tagged and layered as Halo and its components include a SpriteRenderer and two CircleCollider2Ds, one of which is the trigger that detects the Setbacks.
The non-trigger collider is necessary in order to detect the collision of the rain drops with the bubble. However, this makes the Halo collide with the environment and influence the player's movement. In order to solve this problem, the layer Halo has been set to only react with objects layered as Setback (see Figure 6.41 and Figure 6.42).

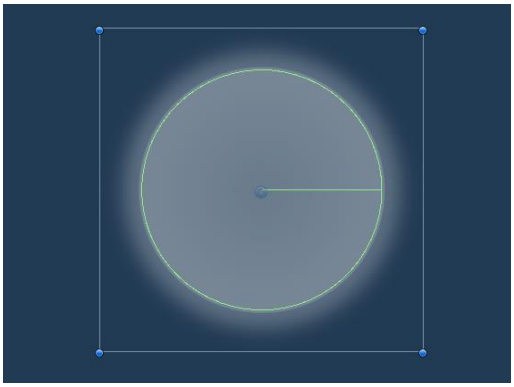

Figure 6.41. Halo prefab in Unity's editor

- **Variables**
  - **`Character`** (`GameObject/Public`): Character's prefab in the scene.
  - **`sr`** (`SpriteRenderer/Private`): The sprite renderer component.
  - **`col`** (`CircleCollider2D/Private`): The trigger collider component.
  - **`isProtecting`** (`Boolean/Private`): Is the Halo currently active?

- **Functions**

  `Start` (`Void/Private`)
  Finds the Character object in the scene, gets the components SpriteRenderer and CircleCollider2D (trigger) and disables both of them.

  `Protect` (`Void/Public`)
  Arguments: `Boolean`
  Enables both the SpriteRenderer and the CircleCollider2D, changes the Character variables canProtect and isProtecting and starts the Protecting co-routine. The boolean argument is specifically used for the fourth chapter and dictates if the player is in a cinematic or not.

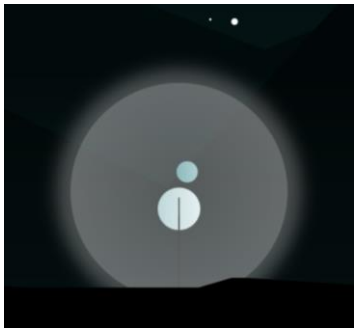

Figure 6.42. Halo prefab in-game

HideHalo (Void/Public)
Arguments: Boolean
The boolean argument dictates if the Halo sizing down animation should be played or not. If the Halo must be hidden immediately, the SpriteRenderer is disabled. Afterwards, the DestroyHalo co-routine is called.

Protecting (IEnumerator/Private)
Arguments: Boolean
Sets the variable isProtecting to true and initializes the variables time and duration. Then, sets a target size for the Halo, which depends on whether it is instantiated by the Mind or by the Light. During the previously defined duration, the Halo increases its current size until it reaches the desired target size. This is done with the following expression:

```
while (time < duration && isProtecting)
{
    transform.localScale = Vector3.Lerp(transform.localScale, targetSize,
                                        time / duration);
    time += Time.deltaTime;
    yield return null;
}
```

Once the target size has been reached, the HideHalo function is called in order to play the sizing down animation. The boolean argument is the one that dictates if the player is in a cinematic; in this case, the script will wait three seconds before calling HideHalo, so as to not destroy the Halo so rapidly.

DestroyHalo (IEnumerator/Private)
This co-routine works very similarly to the Protecting function, but in the opposite way. Instead of increasing the Halo's size, its size is decreased until a very minimal value is reached. Afterwards, the SpriteRenderer and CircleCollider2D are disabled and the isProtecting variables are set to false for both the Halo and the Character. The script then does a short timeout before enabling the Character to protect itself again.

6.4.5. Knowledge

The Knowledge pick-up prefab is implemented in the script *Knowledge.cs*. It consists of a main object, tagged as Knowledge and layered as Default that includes the SpriteRenderer, a trigger BoxCollider2D in charge of detecting the player and an Animator that plays the looping Knowledge orb's flicker animation. The main object has a child GameObject named Block, which contains a vertical CapsuleCollider2D that prevents players to go through the orb if their Knowledge is not full yet. This Block object is untagged, but layered as Knowledge (see Figure 6.43 and Figure 6.44).

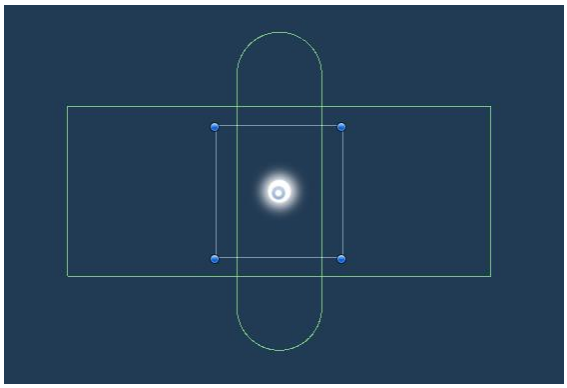

Figure 6.43. Knowledge prefab in Unity's editor

- **Variables**
  - **`Character`** (`GameObject/Public`): Character's prefab in the scene.
  - **`Block`** (`GameObject/Public`): The Block child object.

- **Functions**

  `Start` (`Void/Private`)
  Finds the Character object by its tag in the scene.

  `PickUp` (`Void/Public`)
  Arguments: `Boolean`
  If the Character does not have full Knowledge, adds another unit and the Knowledge object destroys itself.

  `OnTriggerEnter2D` (`Void/Private`)
  Arguments: `Collider2D`
  If the Collider2D that has entered is tagged as "Player", the Knowledge object sets itself as pickable for the Character. Beforehand, if the Character already has full Knowledge, deactivates the Block so as to not force the player to pick up the Knowledge.

OnTriggerExit2D (Void/Private)
Arguments: Collider2D
If the Collider2D that has exited is tagged as “Player”, the Knowledge object unsets itself as pickable for the Character.

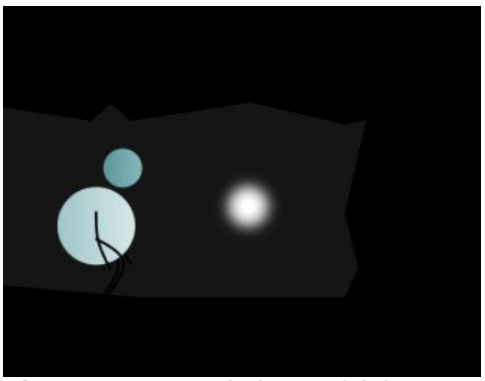

Figure 6.44. Knowledge prefab in-game

6.4.6. Level building

There are different objects that are used to build the different levels which have been saved as prefabs in order to ease the process of layering the different elements out.

Platform

The platforms which are climbed onto are implemented in the script *Platform.cs*. They are tagged and layered as Platform and include a trigger BoxCollider2D and a kinematic RigidBody2D component. In the prefab there are five children objects (see Figure 6.45):

- Sprite: Includes the SpriteRenderer and a non-trigger BoxCollider2D. This child object is layered as Environment, in order to prevent the two colliders of the prefab to overlap.
- The other four children only have a transform component and correspond to the two positions where the player is moved to before starting the animation, the initial position where the character is placed in order to smoothly adapt to the increment of size of the climbing spritesheet and, lastly, the position where the player finishes the animation.

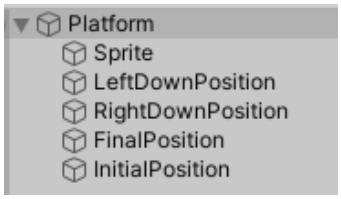

Figure 6.45. Hierarchy of the Platform prefab

Apart from regular platforms, there is also a tunnelled platform object that can either be climbed onto or crouched through. All of its components are the same to the regular platform, except the size of the non-trigger collider and its layer name, which is TunnelledPlatform, not Platform (see Figure 6.46).

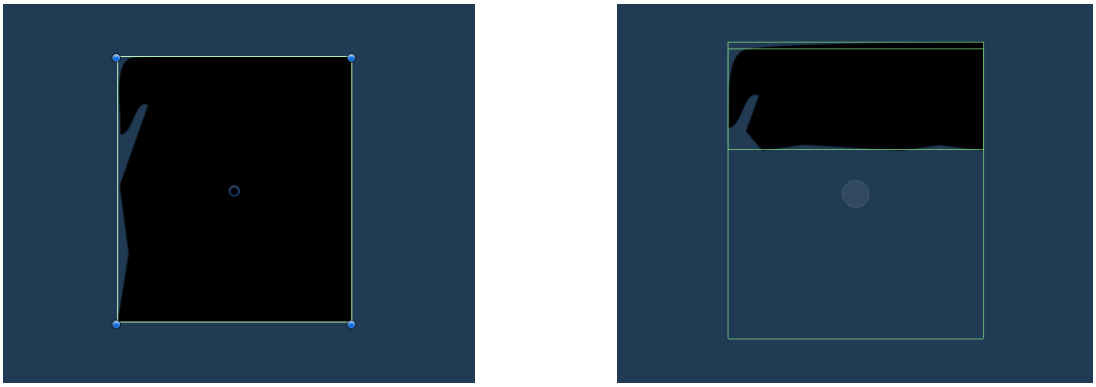

Figure 6.46. Platform and TunnelledPlatform prefabs in Unity's editor

- **Variables**

The script *Platform.cs* includes four public variables that store the four Transform components of each different position: LeftDownPosition, RightDownPosition, FinalPosition and InitialPosition.

- **Functions**

Each variable has its own getter, which returns a Vector3 object that corresponds to the position stored in the Transform.

Tunnel

The tunnel prefab is layered with the same name and is composed of two children objects, which represent the two entries of the tunnel. Both children are also layered as Tunnel and include a trigger BoxCollider2D (see Figure 6.47).

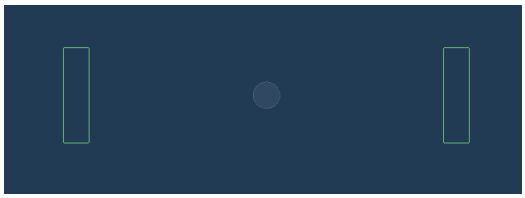

Figure 6.47. Tunnel prefab in Unity's editor

Water

The water object in the game is tagged and layered as Water and is implemented in the script *Water.cs*. The main object includes two colliders, one of which is a trigger and the other is not. Much like the Halo, the non-trigger collider is necessary in order to detect the collision of the rain drops with the water (see Figure 6.49).

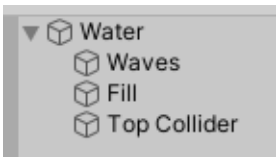

Figure 6.48. Hierarchy of the Water prefab

The prefab has three children objects; the waves, which include a SpriteRenderer and an animator that plays the looping wave animation; the fill, which corresponds to the transparent water sprite that can be sized up or down in order to fit different spaces; the top collider, which is layered as Environment and its purpose is to block the Character from exiting the water, since its gravity scale is set to zero when underwater (see Figure 6.48).

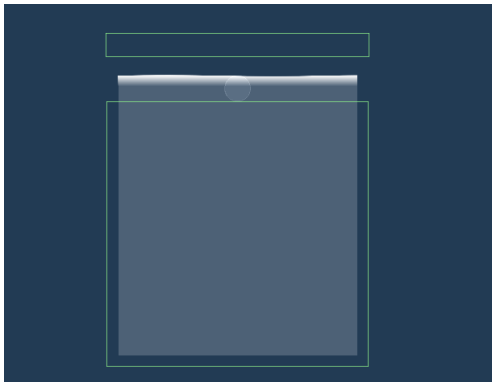

Figure 6.49. Water prefab in Unity's editor

- **Variables**
  - **`Character`** (`GameObject/Public`): Character's prefab in the scene.
  - **`TopCollider`** (`GameObject/Public`): Water's top block.
  - **`swimmable`** (`Boolean/Public`): Can the player swim in this water?

- **Functions**

  `Start` (`Void/Private`)
  Finds the Character object by its tag in the scene.

  `OnTriggerEnter2D` (`Void/Private`)
  Arguments: `Collider2D`
  If the collider that has entered is tagged as Light, all the remaining Character's lives are taken out. However, if the collider is tagged as Player and the water is swimmable, the Character is set to be in water and the TopCollider is enabled.

  `OnTriggerExit2D` (`Void/Private`)
  Arguments: `Collider2D`
  If the collider that has exited is tagged as Player and the water is swimmable, the Character is set to be on ground and the TopCollider is disabled.

Environment limiters

There are several other world-building elements that, although not having any specific components apart from basic colliders, are saved as prefabs in order to facilitate the creation of levels. These are layered as Ground or Environment, and some prime examples are floors, edges and rocks (see Figure 6.50).

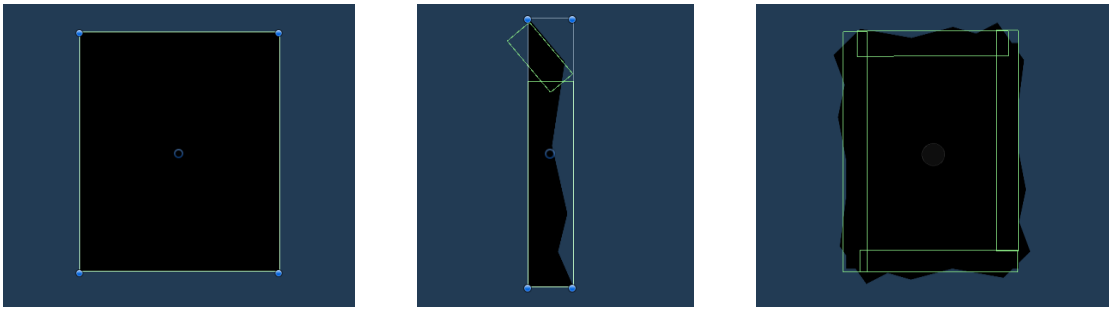

Figure 6.50. Floor, Edge and Rock prefabs in Unity's editor

Other important world elements include trees, which are are stored in singles, groups of two or three, in order to easily place them by hand. However, these do not include a collider since they are used solely in the background for aesthetic purposes (see Figure 6.51).

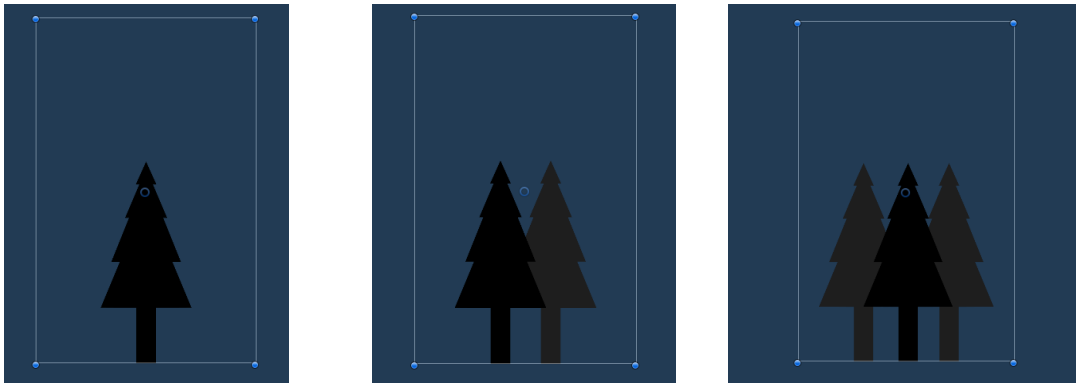

Figure 6.51. Tree prefabs in Unity's editor

Building zone

The zones where the player is able to build and recover the auxiliary platform are implemented in the script *BuildingZone.cs*. These prefabs are composed by a trigger BoxCollider2D and a looping Particle System which, by constantly instantiating and deleting particles, mimics a light white mist that emerges from the ground (see Figure 6.52 and Figure 6.53).

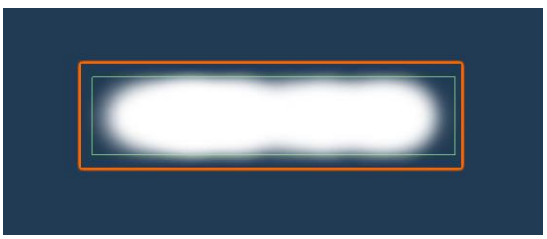

Figure 6.52. BuildingZone prefab in Unity's editor

- **Variables**
  - **`Character`** (`GameObject/Public`): Character's prefab in the scene.

- **Functions**

`Start` (`Void/Private`)
Finds the Character object by its tag in the scene.

`OnTriggerEnter2D/OnTriggerStay2D` (`Void/Private`)
Arguments: `Collider2D`
If the collider that is inside the building zone is tagged as Player, sets the Character to being currently in a building zone.

`OnTriggerExit2D` (`Void/Private`)
Arguments: `Collider2D`
If the collider that has exited the building zone is tagged as Player, sets the Character to not being currently in a building zone.

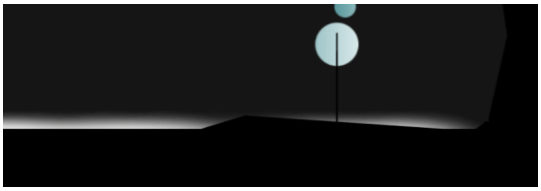

Figure 6.53. BuildingZone prefab in-game

Lever
Levers that can be interacted with are implemented in the script *Lever.cs*. The main object includes only a trigger BoxCollider2D (see Figure 6.55).

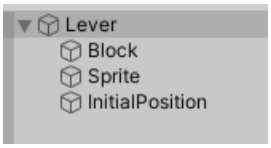

Figure 6.54. Hierarchy of the Lever prefab

In the prefab there are three children objects (see Figure 6.54):

- Block, which includes a PolygonCollider2D that prevents the player from going through the lever.
- Sprite, which includes the components SpriteRenderer and Animator.
- InitialPosition, which stores a Transform component with the position where the player is moved to when interacting with the lever.

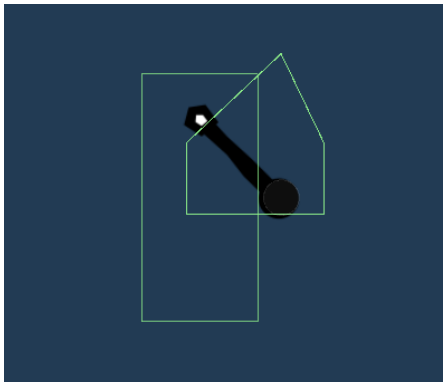

Figure 6.55. Lever prefab in Unity's editor

- **Variables**
  - **`Character`** (`GameObject/Public`): Character's prefab in the scene.
  - **`AssociatedObject`** (`GameObject/Public`): The GameObject that is moved when the lever is activated.
  - **`Sprite`** (`GameObject/Public`): The Sprite child object.
  - **`InitialPosition`** (`Transform/Public`): The Transform component of the InitialPosition child object.
  - **`startPosition/finalPosition`** (`Vector3/Public`): The world position where the AssociatedObject starts from and the position where it ends.
  - **`stopsPlayer`** (`Boolean/Public`): Should the player be inactive while the AssociatedObject is moving?
  - **`isMoving`** (`Boolean/Private`): Is the AssociatedObject moving?
  - **`currentlyInteractable`** (`Boolean/Private`): Is the lever active?
  - **`timesUsed`** (`Integer/Private`): The number of times the player has interacted with the lever.
  - **`anim`** (`Animator/Private`): The Animator component of the Sprite child object. It includes three states: Still, which does not play any animation; Up Down, which plays the animation of the lever being pulled; and Down Up, which plays an animation that rotates the lever back to its original position (see Figure 6.56).

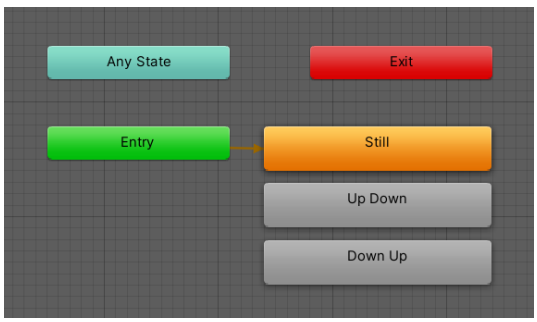

Figure 6.56. Animator controller for the Lever

- **Functions**

`Start` `(Void/Private)`
Gets the animator component from the Sprite and finds the character object in the scene by its tag.

`Activate` `(Void/Public)`
If the lever is active, the co-routine RotateLever is called and, if the AssociatedObject is not moving, increments the variable timesUsed and calls the co-routine MoveAssociatedObject (see Figure 6.57).

Figure 6.57. Lever being activated in-game

`RotateLever` `(IEnumerator/Private)`
Sets the lever to not currently interactable and plays the animation Up Down (see Figure 6.58), so as to represent the lever being pulled.

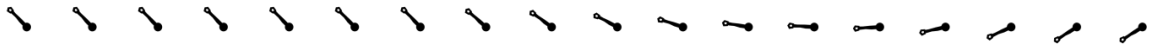

Figure 6.58. Spritesheet of the lever's Up Down animation

After a two second delay, the animation Down Up is played and the lever goes back to its original rotation. There is a one second delay and, if the AssociatedMoving is not moving anymore, the lever is set back to being interactable. As in the rest of the game's implementation, these delays are implemented with the function WaitForSeconds, which suspends the co-routine execution for the given amount of seconds.

`MoveAssociatedObject` `(IEnumerator/Private)`
Starts by setting a default speed and delays the function for three seconds. Afterwards, sets the target position based on the times the lever has been used. Therefore, if the lever has been interacted with an even amount of times, the AssociatedObject has to be moved to its start position. When the AssociatedObject has reached its destination, by using the Vector2's function MoveTowards, the lever is set back to being interactable and, if it stops the player, the Character is made active again.

`isCurrentlyInteractable/doesItStopPlayer` `(Boolean/Public)`
Getters for the variables currentlyInteractable and stopsPlayer.

`getInitialPosition` `(Vector3/Public)`
Getter for the InitialPosition transform's position value.

`OnTriggerEnter2D/OnTriggerStay2D` `(Void/Private)`
Arguments: `Collider2D`
If the Collider2D inside the collider is tagged as "Player", the Lever object sets itself as interactable for the Character.

`OnTriggerExit2D` `(Void/Private)`
Arguments: `Collider2D`
If the Collider2D that has exited the collider is tagged as "Player", the Lever object unsets itself as interactable for the Character.

Spikes

The spike traps that are lethal to the player are implemented in the script *Spikes.cs*. These prefabs are composed by a trigger BoxCollider2D, a SpriteRenderer and a looping Particle System which, by constantly instantiating and deleting particles, mimics a black mist that emerges from the spikes. There is one child object named Block, which includes a non-trigger BoxCollider2D that acts as a foundation (see Figure 6.59).

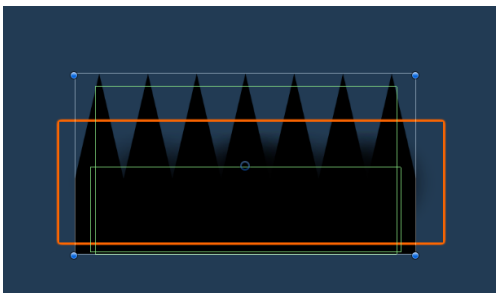

Figure 6.59. Spikes prefab in Unity's editor

- **Variables**
  - **`Character`** `(GameObject/Public)`: Character's prefab in the scene.

- **Functions**

`Start` `(Void/Private)`
Finds the Character object by its tag in the scene.

`OnTriggerEnter2D` `(Void/Private)`
Arguments: `Collider2D`
If the collider that has entered is tagged as Player, all the remaining Character's lives are taken out (see Figure 6.60).

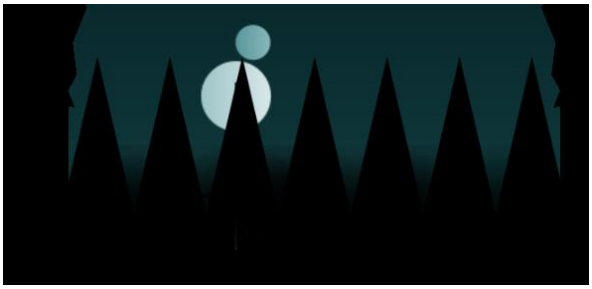

Figure 6.60. Spikes prefab in-game

6.4.7. Auxiliary platform

The auxiliary platform that players can build is saved as a prefab named CharacterPlatform. Its components and structure are nearly identical to the regular Platform, including also the *Platform.cs* script (see Figure 6.61).

Figure 6.61. Hierarchy of the CharacterPlatform prefab

However, this box is special since it also includes a child BuildingZone object on top of it, so as to allow the player to recover the platform when on top. This BuildingZone is formed by a trigger BoxCollider2D and the script *BuildingZone.cs*. In addition, in order to make a visual connection between the auxiliary platform and the Knowledge, the platform has a white glow that surrounds it (see Figure 6.62 and Figure 6.63). Please, note that both scripts mentioned in this section have a more detailed description in section 6.4.6.

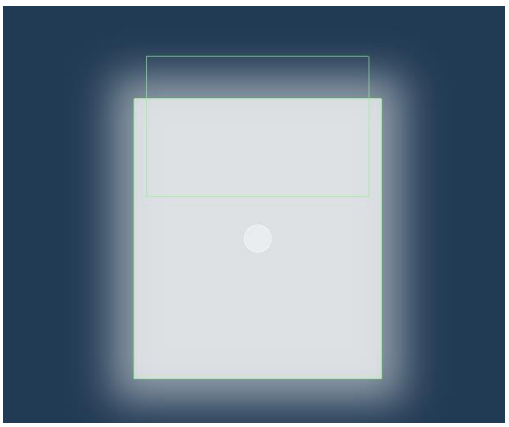

Figure 6.62. CharacterPlatform prefab in Unity's editor

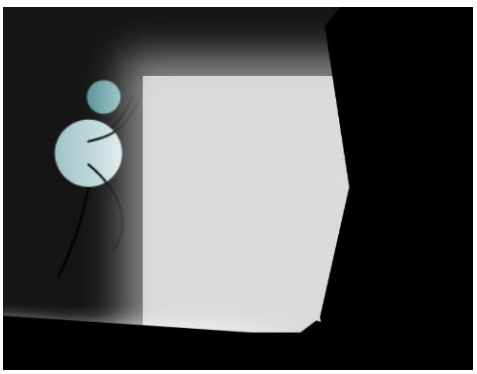

Figure 6.63. CharacterPlatform prefab in-game

Feedback

When the player cannot build an auxiliary platform due to an environment element being in the way, in order to give the player visual feedback, a semi-transparent black box is spawned and then faded out (see Figure 6.64 and Figure 6.65).

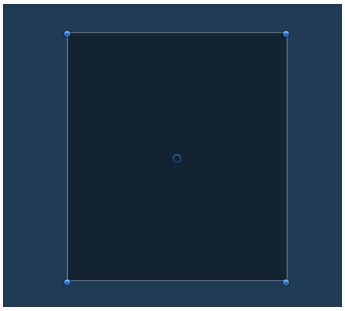

Figure 6.64. CharacterPlatformFeedback prefab in Unity's editor

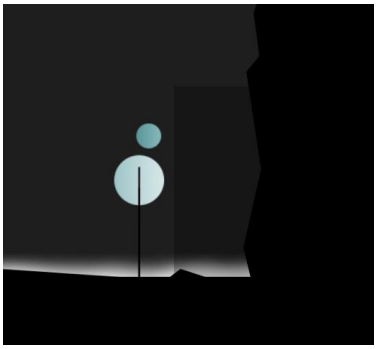

Figure 6.65. CharacterPlatformFeedback prefab in-game

The components that form this prefab include a SpriteRenderer and an Animation. The latter consists of two key frames that interpolate the sprite's transparency value (alpha) from eighty to zero. When the last key frame is reached, an event is triggered. This event calls the Destroy function, which is implemented in the script *BoxFeedback.cs*.

- **Functions**

  `Destroy` `(Void/Private)`

  Is called when the animation is over and destroys the prefab.

6.4.8. Checkpoints

Checkpoints are mainly used to update the player's respawn position according to its progress through the level. In addition, some of the checkpoints also restart the character's auxiliary platform so as to prevent the player from recovering a platform that is too far behind. The prefab is implemented in the script *Checkpoint.cs* and consists of one trigger CapsuleCollider2D identical to the Character's own collider (see Figure 6.66).

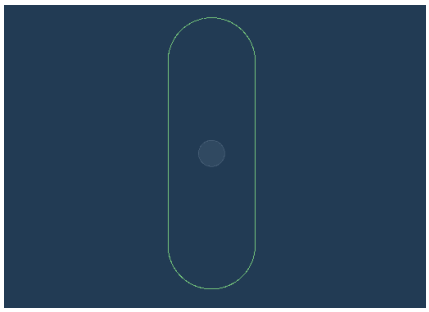

Figure 6.66. Checkpoint prefab in Unity's editor

- **Variables**
  - **`Character`** (`GameObject/Public`): Character's prefab in the scene.
  - **`resetsBox`** (`Boolean/Public`): Will the player be able to recover the auxiliary platform previously placed along the level, or not?
  - **`triggered`** (`Boolean/Private`): Has the player already entered the checkpoint?

- **Functions**

  `Start` (`Void/Private`)

  Finds the Character object by its tag in the scene.

  `OnTriggerEnter2D` (`Void/Private`)

  Arguments: `Collider2D`

  If the collider that has entered is tagged as Player and the checkpoint has not been triggered yet, the player's last position is updated and the checkpoint is set as triggered. In addition, if the checkpoint is set to reset the box, the character is set not to have a box currently placed.

6.4.9. Tutorial Zone

Tutorial zones display on screen basic game information and are implemented in the script *Tutorial.cs*. The main object includes only a trigger BoxCollider2D (see Figure 6.68 and Figure 6.69).

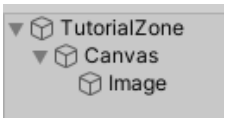

Figure 6.67. Hierarchy of the TutorialZone prefab

The prefab has a Canvas child object that includes an Image that is rendered in world space (see Figure 6.67).

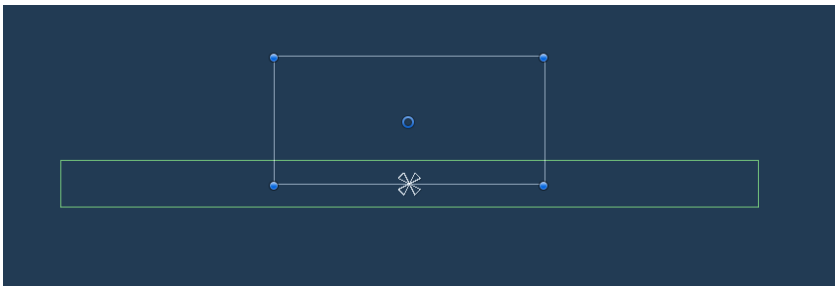

Figure 6.68. TutorialZone prefab in Unity's editor

Figure 6.69. TutorialZone prefab in-game

- **Variables**
  - **`Character`** (`GameObject/Public`): Character's prefab in the scene.
  - **`TutorialImage`** (`Image/Public`): The Image object from the Canvas.
  - **`Options`** (`Sprite array/Public`): An array with the three variations of the image prompt. Each variation displays a different button to press depending on the current input type: keyboard, PlayStation controller or XBOX controller.
  - **`startDelay`** (`Float/Public`): The delay before the tutorial prompt is first shown.
  - **`firstTime`** (`Boolean/Private`): Has the prompt been shown before?
  - **`fadingIn`** (`Boolean/Private`): Is the prompt fading in?
  - **`fadingOut`** (`Boolean/Private`): Is the prompt fading out?
  - **`lastInputType`** (`Integer/Private`): The latest input type detected.

- **Functions**

  `Start` `(Void/Private)`
  Sets the Image's sprite according to the current input type and finds the Character object by its tag in the scene.

  `Update` `(Void/Private)`
  If the input type has changed, the Image's sprite is updated.

  `OnTriggerEnter2D` `(Void/Private)`
  Arguments: `Collider2D`
  If the collider that has entered is tagged as Player, the prompt is set to fade in and, if it is not currently fading out, the co-routine to change the image's alpha is started. If it is the first time the character has entered the tutorial zone, sets firstTime to false.

  `OnTriggerExit2D` `(Void/Private)`
  Arguments: `Collider2D`
  If the collider that has exited is tagged as Player, the prompt is set to fade out and, if it is not currently fading in, the co-routine to change the image's alpha is started.

  `ChangeAlpha` `(IEnumerator/Private)`
  Arguments: `Boolean`
  Fades in or out the TutorialImage depending on the boolean argument, which dictates if the prompt is to be hidden or not. If the player has exited the zone, keeps decreasing the image's alpha value while the player is outside the collider. If the player enters the tutorial zone while the image is fading out, it starts fading it in. On the other hand, if the player has entered the zone, keeps increasing the image's alpha value while the player is inside the collider. If the player exits the tutorial zone while the image is fading in, it starts fading it out.

  `HidePrompt` `(Void/Private)`
  Sets the image to fade out without taking into account whether the player is outside the tutorial zone or not. Mainly used after the prompt button is pressed in chapters four and six.

6.5. Cinematics

In order to give the game a proper cinematic feel, the game transitions smoothly between chapters and uses mechanics that implement the technique commonly known as *“Show, don't tell”.* The game’s cinematics can be divided in those that happen while in-game and those that do not.

6.5.1. In-game

There are triggers along the levels that smoothly displace the camera in order to present different elements in a more visual way. These object, named CinematicCameraTrigger, include a trigger BoxCollider2D and are implemented in the script *CinematicCamera.cs* (see Figure 6.70).

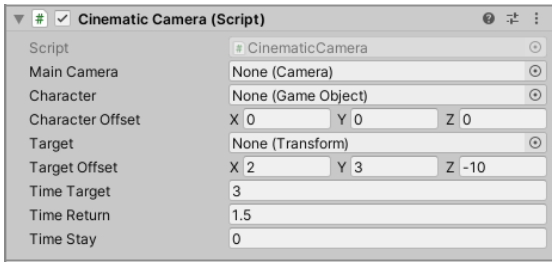

Figure 6.70. CinematicCamera script's public variables

- **Variables**
  - **`mainCamera`** (`Camera/Public`): The game’s rendering camera.
  - **`character`** (`GameObject/Public`): Character’s prefab in the scene.
  - **`characterOffset`** (`Vector3/Public`): The camera’s position offset to the player.
  - **`target`** (`Transform/Public`): The transform component of the target object in the scene.
  - **`targetOffset`** (`Vector3/Public`): The camera’s position offset to the target.
  - **`timeTarget/timeReturn/timeStay`** (`Float/Public`): The seconds the camera takes to reach the target, return to the character and the time it stays on the target, respectively.
  - **`triggered`** (`Boolean/Public`): Has the animation been triggered yet?

- **Functions**

  `OnTriggerEnter2D` (`Void/Private`)

  Arguments: `Collider2D`

  If the collider that has entered is tagged as Player and the cinematic has not been triggered yet, starts doing the cinematic.

doCinematic (Void/Private)
Sets the cinematic to be triggered, makes the character inactive, sets the camera to stop following the player and starts the keepDoingCinematic co-routine, which takes as arguments a target position, duration of the movement and order.

keepDoingCinematic (IEnumerator/Private)
Arguments: Vector3, Float, Integer
Moves the camera to the target position in the specified duration. Depending on the order of the animation, the camera goes to the target object's position or the character's position.

6.5.2. Cutscenes

There are cinematics that are not integrated directly into the gameplay, except for occasional button prompts. These cinematics are often found in transitions between chapters, or in cinematic chapters, and share many aspects, such as variables or functions.

- **Common variables**
  - **TitleCover** (Image/Public): An image with the chapter's number and name.
  - **BlackScreen** (Image/Public): A small black image used for transitions between scenes.
  - **HUDBlock** (Image/Public): A large black image that hides the heads-up display.
  - **order** (Integer/Private): The current order of the cinematic.

- **Common functions**
  doCinematic (Void/Private)
  Calls a different function or co-routine depending on the current order. All other functions include a boolean argument that determines if, once executed, the order should be increased.

  FadeIn (IEnumerator/Private)
  Arguments: Image, Boolean
  Fades in the image by increasing its colour alpha value (transparency) over time. If the boolean is set to true, increments the current order and calls the function doCinematic.

```
IEnumerator FadeIn (Image i, bool changesOrder)
{
    float t = 1f;

    while (i.color.a < 1f)
    {
        i.color = new Color(i.color.r,i.color.g,i.color.b,i.color.a+(Time.deltaTime/t));
        yield return null;
    }

    if (changesOrder)
    {
        order += 1;
        doCinematic();
    }
 }
```

FadeOut (IEnumerator/Private)

Arguments: Image, Boolean

Fades out the image by decreasing its colour alpha value (transparency) over time. If the boolean is set to true, increments the current order and calls the function doCinematic.

```
 IEnumerator FadeOut (Image i, bool changesOrder)
{
    float t = 1f;

    while (i.color.a > 0f)
    {
        i.color = new Color(i.color.r,i.color.g,i.color.b,i.color.a-(Time.deltaTime/t));
        yield return null;
    }

    if (changesOrder)
    {
        order += 1;
        doCinematic();
    }
 }
```

MoveObject (IEnumerator/Private)

Arguments: GameObject, Vector3, Float, Boolean

Moves the GameObject to the target position in the defined duration. If the boolean is set to true, increments the current order and calls the function doCinematic.

```
IEnumerator MoveObject (GameObject obj, Vector3 target, float duration, bool changeOrder)
{
    float time = 0f;

    Vector3 start = obj.transform.position;

    while (time < duration)
    {
       Vector3 newPosition = obj.transform.position;

       newPosition.x = Mathf.Lerp(start.x, target.x, time / duration);
       newPosition.y = Mathf.Lerp(start.y, target.y, time / duration);

       obj.transform.position = newPosition;

       time += Time.deltaTime;
       yield return null;
    }

    if (changeOrder)
    {
       order += 1;
       doCinematic();
    }
}
```

MoveCamera (IEnumerator/Private)

Arguments: Vector3, Vector3, Float, Boolean

Moves the main camera to the target position in the defined duration. If the boolean is set to true, increments the current order and calls the function doCinematic.

```
IEnumerator MoveCamera (Vector3 start, Vector3 target, float duration, bool changeOrder)
{
    float time = 0f;

    while (time < duration)
    {
       Vector3 newPosition = mainCamera.transform.position;

       newPosition.x = Mathf.Lerp(start.x, target.x, time / duration);
       newPosition.y = Mathf.Lerp(start.y, target.y, time / duration);

       mainCamera.transform.position = newPosition;

       time += Time.deltaTime;
       yield return null;
    }

    if (changeOrder)
    {
       order += 1;
       doCinematic();
    }
}
```

All chapters' cinematics are separated in two scripts, one for the start and one for the ending. The start cinematic is stored inside the Controller object of the scene, while the ending is stored in a GameObject with a trigger collider at the end of the scene. Nonetheless, since chapters four and six are purely cinematic, every aspect of their animation is implemented in a single script. The scripts that implement the different cinematics and transitions are:

- *Chapter1_Start.cs* / *Chapter1_End.cs*
- *Chapter2_Start.cs* / *Chapter2_End.cs*
- *Chapter3_Start.cs* / *Chapter3_End.cs*
- *Chapter4.cs*
- *Chapter5_Start.cs* / *Chapter5_End.cs*
- *Chapter6.cs*
- *Credits.cs*

6.6. User interface

6.6.1. Main menu

The main menu of the game is stored in the scene Menu and is implemented in the script *MainMenu.cs*. The scene's canvas component consists of (see Figure 6.71):

- The object Main, which contains the principal buttons of the menu.
- The object Play, which contains the buttons for each chapter.
- The object Options, which includes a toggle to full screen the game.
- Other images, such as the WAVA logo or a black screen.

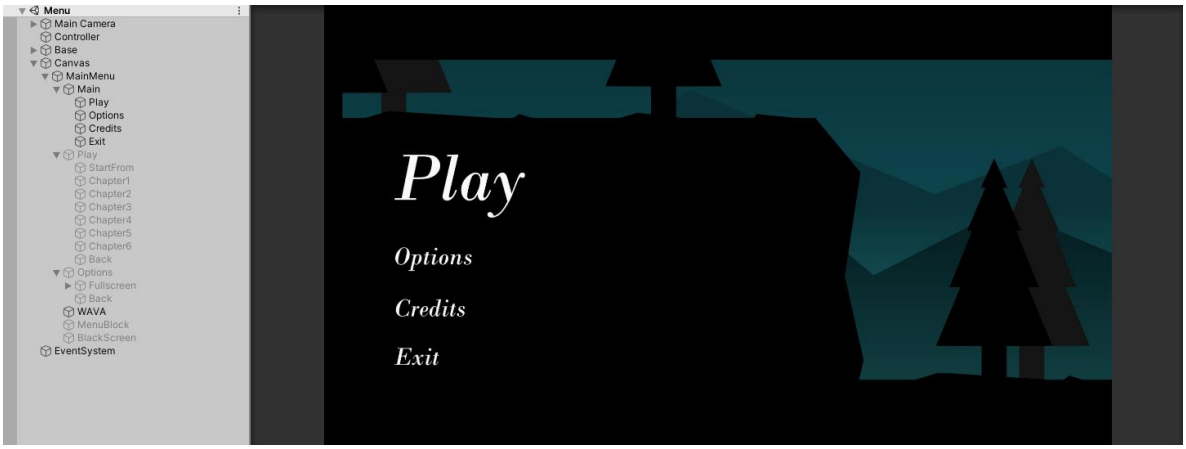

Figure 6.71. Hierarchy and view of the Menu scene in Unity's editor

- **Variables**
  - **`mainCamera`** (`Camera/Public`): The game's rendering camera.
  - **`InitialPosition`** (`Vector3/Public`): The initial camera position.
  - **`MenuPosition`** (`Vector3/Public`): The menu's camera position.
  - **`MenuBlock`** (`Image/Public`): A black image that hides the menu.
  - **`BlackScreen`** (`Image/Public`): A black screen to transition between chapters.
  - **`WAVALogo`** (`Image/Public`): The logo of the game.
  - **`order`** (`Integer/Public`): The current order of the cinematic.
  - **`Chapter`** (`Integer/Public`): The chapter selected by the player.
  - **`quitGame`** (`Boolean/Public`): Has the player pressed the exit button?

- **Functions**

  `doCinematic` (`Void/Private`)

  Starts by moving the camera to the MenuPosition, fading the MenuBlock out and fading the WAVALogo in. Then, if the player has selected a level to play, loads the scene that corresponds to that chapter.

  `ExitGame` (`Void/Public`)

  Sets quitGame to true and starts to fade in the BlackScreen. Once the image is completely visible, the function *Application.Quit()* is called.

PlayButton (Void/Public)
Arguments: Integer
Sets the chapter to load depending on the integer argument's value. Fades the BlackScreen in and, once the image is completely visible, that chapter's scene is loaded.

SetFullscreen (Void/Public)
Arguments: Boolean
Changes the game's view mode to full screen depending on the boolean argument's value.

6.6.2. Pause menu

The pause menu of the game is stored in the prefab PauseMenu and is implemented in the script *PauseMenu.cs*. The prefab consists of (see Figure 6.72):

- The object Buttons, which contains the principal buttons of the pause menu.
- The object Controls, which contains an image with the game's controls.
- The object Options, which includes toggles to full screen the game or hide the heads up display.

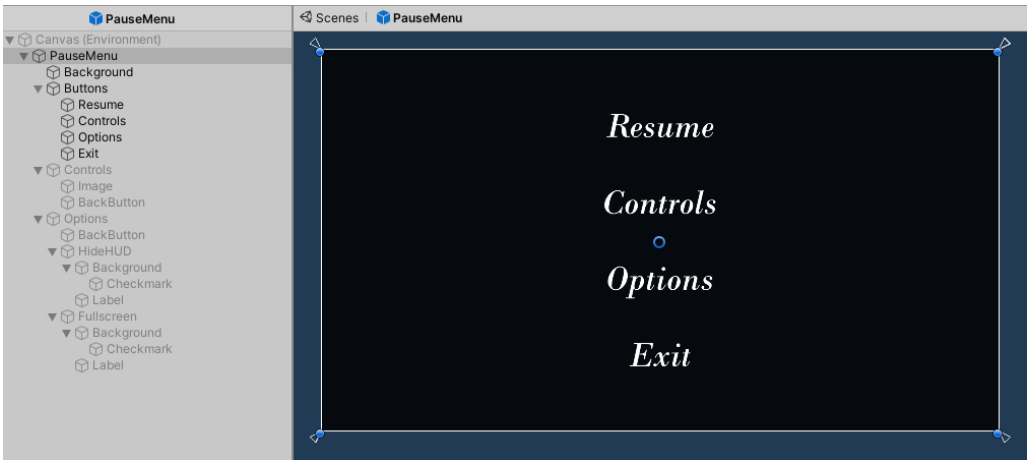

Figure 6.72. Hierarchy and view of the PauseMenu prefab in Unity's editor

- **Variables**
  - **`Character`** (`GameObject/Public`): Character's prefab in the scene.
  - **`BlackScreen`** (`Image/Public`): A black screen to transition between chapters.
  - **`HUDBlock`** (`Image/Public`): A black image that hides the HUD.
  - **`hiddenHUD`** (`Boolean/Public`): Is the hide HUD toggle active?
  - **`goToMenu`** (`Boolean/Public`): Has the player pressed the exit button?

- **Functions**

  `HideHUD` (`Void/Public`)

  Depending on whether the toggle to hide the heads up display is active, the HUDBlock's colour alpha value is set to 0 (transparent) or 1 (opaque).

  `SetFullscreen` (`Void/Public`)

  Arguments: `Boolean`

  Changes the game's view mode to full screen depending on the boolean argument's value.

  `Exit` (`Void/Public`)

  Sets goToMenu to true and starts to fade in the BlackScreen. Once the image is completely visible, the Menu scene is loaded.

6.6.3. HUD

The heads up display of the game is stored in the prefab HUD and is divided in two parts: the KnowledgeBar and the LivesSlots (see Figure 6.73).

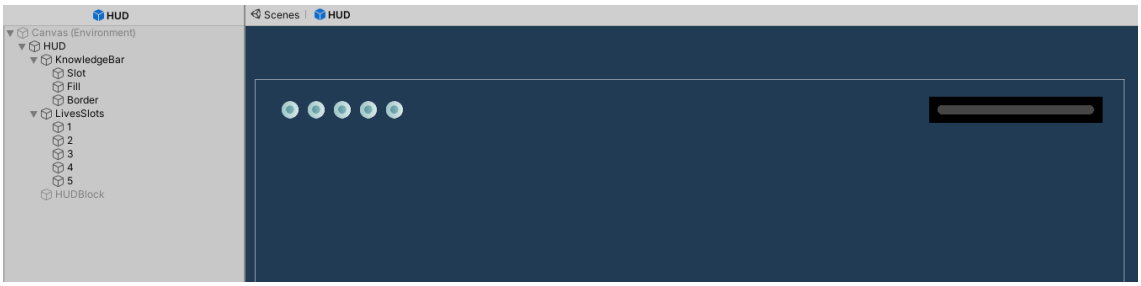

Figure 6.73. Hierarchy and view of the HUD prefab in Unity's editor

The Knowledge bar is located on the top right and consists of a slider object that represents the player's current Knowledge. Its general functioning is implemented in the script *KnowledgeBar.cs*.

- **Variables**
  - **`smoothFactor`** (`float/Private`): Value to make the increase and decrease of the bar's fill more fluid.
  - **`slider`** (`Slider/Public`): The slider object, which has a maximum value of five.

- **Functions**

  `SetKnowledge` (`Void/Public`)
  Arguments: `Float`
  Calls LerpBar with the new Knowledge value as an argument.

  `LerpBar` (`IEnumerator/Private`)
  Arguments: `Float`
  Smoothly interpolates between the current slider value and the one that has been entered as an argument.

```
slider.value = Mathf.Lerp(slider.value, Knowledge, smoothFactor * Time.deltaTime);
```

The lives slots are located on the top left and consist of a five images that represent the player's current health. Its general functioning is implemented in the script *LivesSlots*.cs.

- **Variables**
  - **Images** (Image array/Public): The five image slots.
  - **Sprites** (Sprite array/Public): The two sprite options that can be placed in each image.
  - **maxLives** (Integer/Private): Maximum of lives the player can have.

- **Functions**

  SetLives (Void/Public)

  Arguments: Integer

  Sets a specific Sprite in each image slot depending on the current number of lives entered as an argument. From zero to the current lives value, sets the full life Sprite and, for the rest of images, sets an empty slot.

```
for (int i = 0; i < Lives; i++)
{
    Images[i].sprite = Sprites[1];
}

for (int j = Lives; j < maxLives; j++)
{
    Images[j].sprite = Sprites[0];
}
```

6.7. Testing

6.7.1. Editor testing

Along all the game's development, each newly implemented component was tested in the Unity's editor. In some instances, a more complete playthrough of the game was done in order to analyse the overall game behaviour. Showing the different objects' Gizmos, such as colliders or rays, was extremely beneficial for comprehending the different problems and errors encountered (see Figure 6.74).

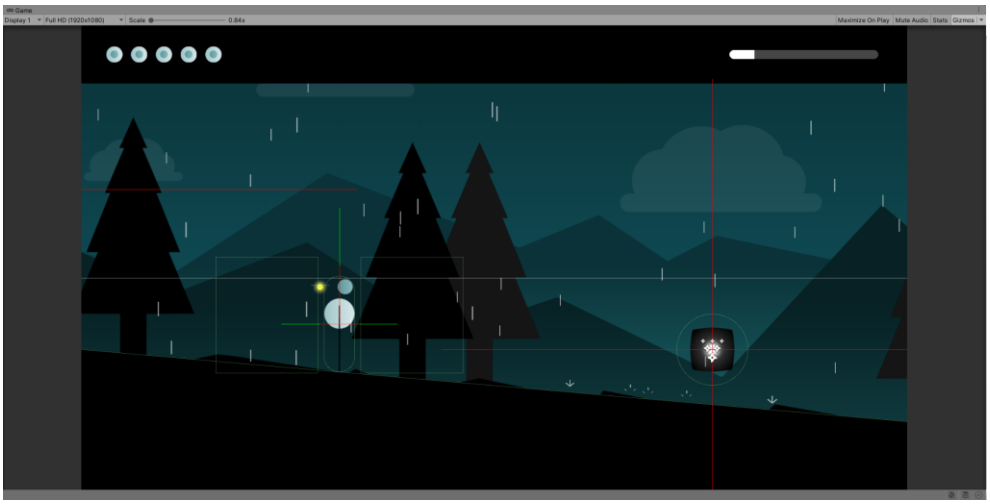

Figure 6.74. Testing a scene inside Unity's editor

6.7.2. Build testing

In order to test the project as a finished product, several versions of the game were built and played through. Apart from our own examination, other individuals also tested the game in order to help us determine if the project was accessible, enjoyable and whether the story was transmitted correctly. The testers' level of expertise was very diverse and, considering their positive feedback, we were able to conclude that the game indeed caters to a wide and varied audience.

7. Results

In this section we will analyse the final outcome of the objectives defined at the start of the project (see section 1.3).

The main purpose of this project was to develop a cinematic 2D game with puzzle and platforming elements that intended to transmit a profound message with its design and unique aesthetic. For each one of the more specific objectives previously defined, we will evaluate their level of completion.

- **Deepening the knowledge already acquired about the game engine Unity and the programming language C#.**
  After a full year of developing this project, we have become more familiarised with Unity and C# and have learnt new functions, methods and other implementation techniques.

- **Implement game mechanics that reinforce the story and the message transmitted by the game.**
  All the planned game mechanics have been implemented and the proposed game loop has been materialised. In addition, some mechanics that were originally intended to be developed in the near future, such as the underwater movement, have also been created for this version of the game.

- **Design an atmospheric environment that is captivating, unique and reinforces the game's predominant topics.**
  In spite of its simplicity, the game's environment feels unique and alive thanks to its geometry, colouring, level design and other aspects, such as the parallax background effect.

- **Create different animations for the character and other assets that give the game a certain smoothness and fluidity.**
  All the game characters have been brought to life by a wide array of spritesheet animations, as well as with the inclusion of particle systems. Additionally, the game transitions smoothly between chapters with its cinematics and other visual elements.

In conclusion, we can determine that all objectives have been reached and exceeded, since there are several mechanics and cinematics that were not expected to be developed for this final degree project.

7.1. Final in-game screenshots

Figure 7.1. In-game screenshots of the splash screen

Figure 7.2. In-game screenshot of the main menu

Figure 7.3. In-game screenshots of the chapters' title covers

Figure 7.4. In-game screenshot of the initial text cinematic

Figure 7.5. In-game screenshot of Chapter 1 (1)

Figure 7.6. In-game screenshot of Chapter 1 (2)

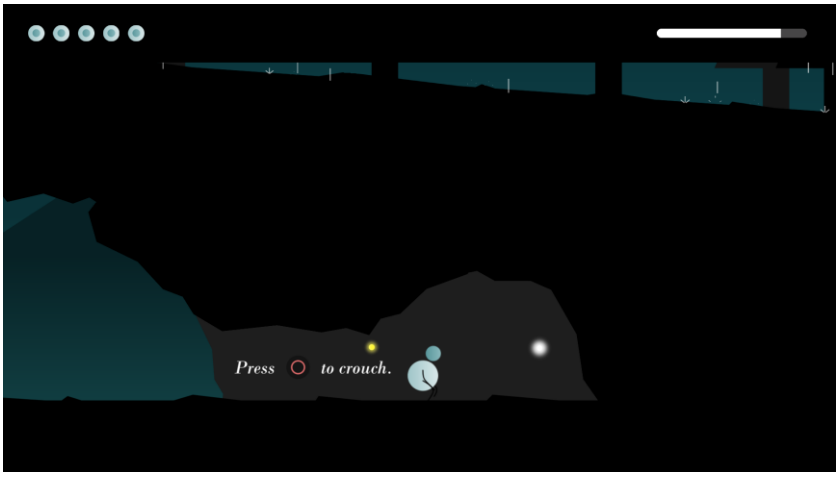

Figure 7.7. In-game screenshot of Chapter 1 (3)

Figure 7.8. In-game screenshot of Chapter 1 (4)

Figure 7.9. In-game screenshot of Chapter 1 (5)

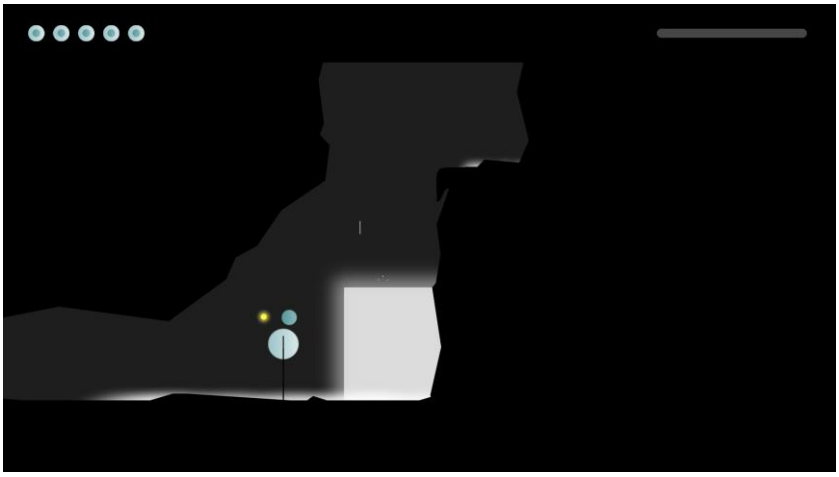

Figure 7.10. In-game screenshot of Chapter 2 (1)

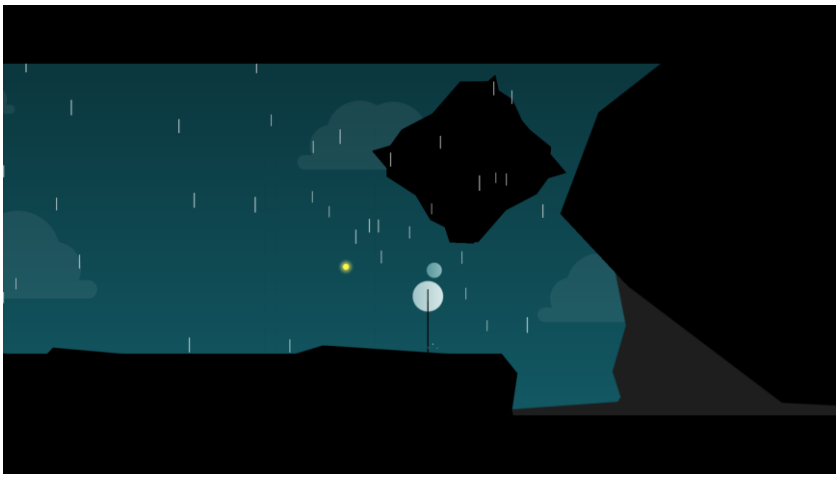

Figure 7.11. In-game screenshot of Chapter 2 (2)

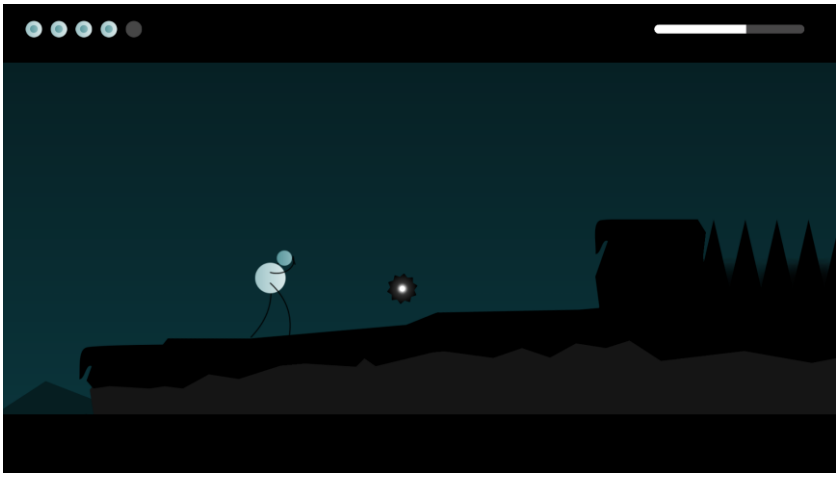

Figure 7.12. In-game screenshot of Chapter 3 (1)

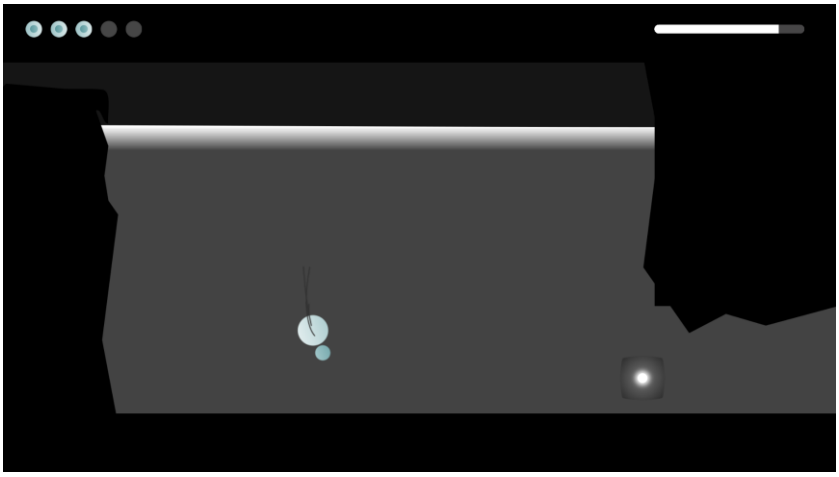

Figure 7.13. In-game screenshot of Chapter 3 (2)

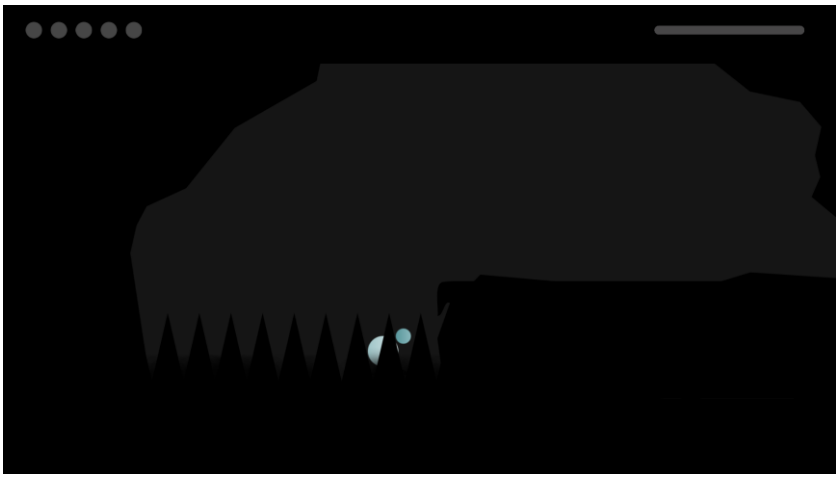

Figure 7.14. In-game screenshot of Chapter 3 (3)

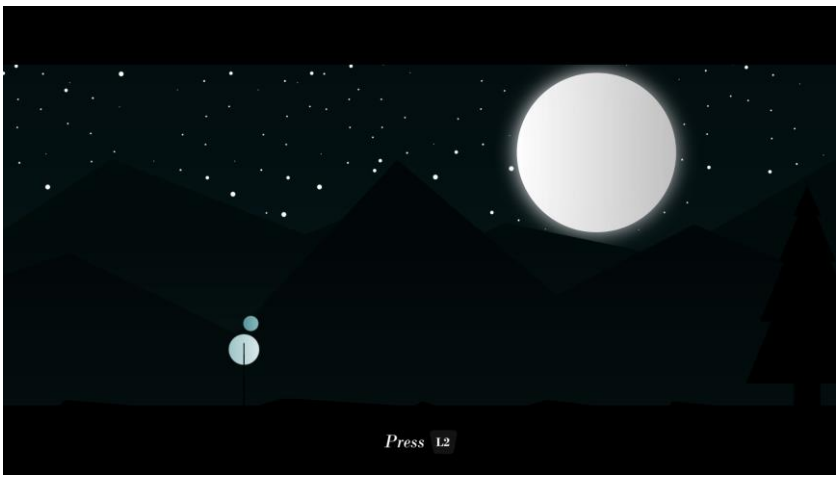

Figure 7.15. In-game screenshot of Chapter 4

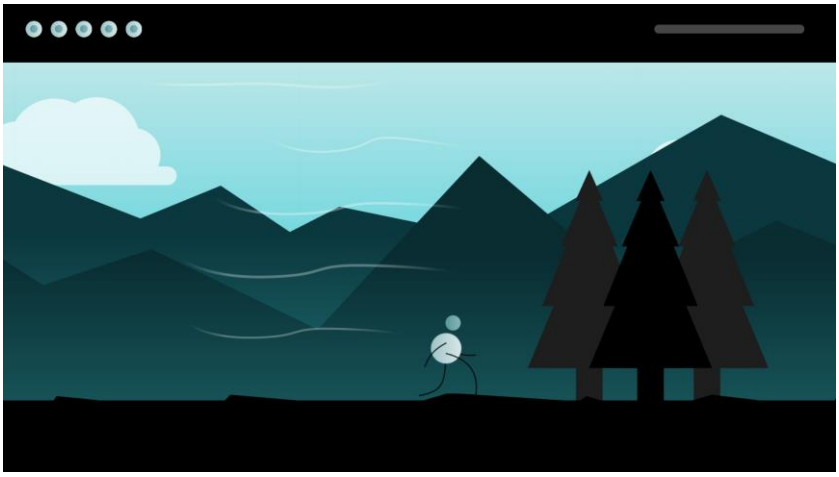

Figure 7.16. In-game screenshot of Chapter 5 (1)

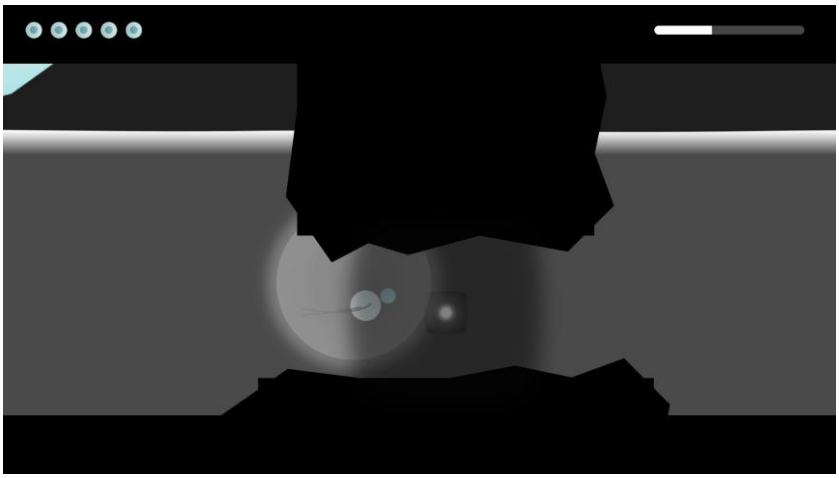

Figure 7.17. In-game screenshot of Chapter 5 (2)

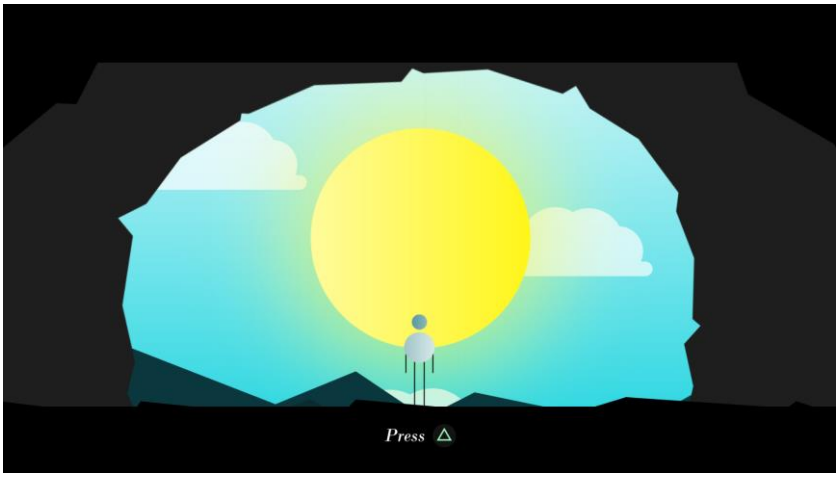

Figure 7.18. In-game screenshot of Chapter 6

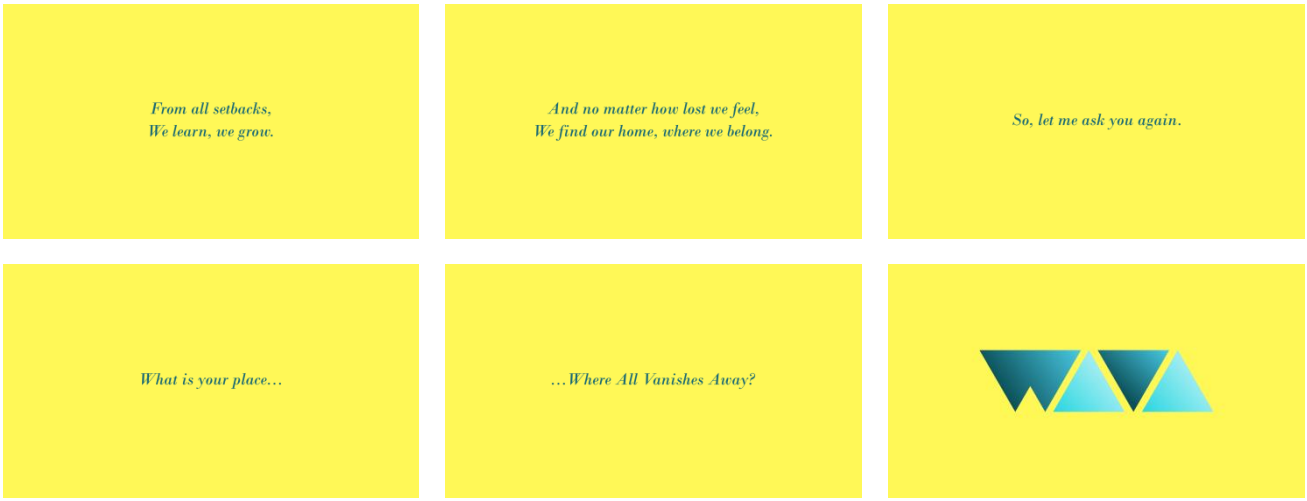

Figure 7.19. In-game screenshots of the final text cinematics

Figure 7.20. In-game screenshots of the credits

All screenshots from Figure 7.1 to Figure 7.20 have been taken while doing a playthrough of the final built game.

8. Conclusions

This final degree project has been the first major game fully designed and developed in solitary and it has allowed us to put into practice all the knowledge acquired during the Bachelor's in Design and Development of Video Games. In spite of preferring and usually focusing on the design and narrative aspects of game development, working on the artistic and implementation side of things has been greatly appreciated and enjoyed.

The idea for WAVA started in 2019, when we were assigned to make a marketing plan for an imaginary game and, at that point, the title WAVA was born. The geometry formed by the four letters and their similarity to a mountain range created an inner thought that would be revisited later on. The summer after, the four letters turned into an acronym; then, the concept visuals for the game were made and, step by step, every single aspect came together and culminated in the game that has been developed for this project.

As stated previously, all the objectives proposed at the start of this project have been reached and exceeded, as the final prototype of the game has ended up being bigger and longer that what was expected at first. Originally, only two or three levels were planned to be developed; however, in the end, all six chapters of the game have been implemented. This has been made possible thanks to good planning and management before and during the game's design and implementation.

Developing a video game from scratch is no easy task and many problems and bugs can occur. However, thanks to Unity's explicit documentation and the many forums and tutorials that can be consulted online, all the different development setbacks were overcome with very little trouble. All in all, we are very pleased with all the tools and resources used to develop this project, including software and hardware, since they all have done their job perfectly without supposing any real economic cost.

Even though we feel the final outcome of the game is very polished and nearly final, we are very passionate about this project and want to keep developing several aspects in order to improve the overall experience. All of these tasks that are to be developed in the near future will be defined in the next section.

In conclusion, we are very pleased with the final product and we believe that it has been the perfect culmination to our university tenure. Much like in WAVA, a new chapter begins and it is time to find our own place in the video game industry, which is one that requires hard work, dedication and, above all things, passion for what we create.

9. Future work

In this section we will define the current aspects of the game that can be improved, as well as those elements that, although not being proposed for this final degree project, can be added in the near future.

- **Add music and sound elements**
  Cinematic games are extremely visual per se; however, the different audio elements help enhance the player's experience. A priority for our future work is to add different sound effects along with a backing soundtrack that reinforces the narrative and visuals.

- **Upgrade the visual aspect**
  Even though we are satisfied with the visual result, we believe there are many aspects that can be improved. These aspects include more variety in the world elements, adding new animations to the characters and environment and creating smoother cinematics.

- **Optimization of the implementation**
  The scripts and overall implementation of the game can be re-organized in order to optimize the game's development and ease the understanding of the code.

- **Implement full controller support**
  As of now, the game is only available on PC and can be played with keyboard, PlayStation controller or XBOX controller. Nonetheless, the interaction with menus can only be done with the computer's mouse. Allowing players to navigate through the menus with their controller would make the game more accessible and convenient.

- **Add new game mechanics**
  In order to improve the overall player experience and create richer puzzles, new game mechanics and actions will be implemented. Some examples include pushing and pulling boxes or climbing ladders.

- **Prolong the duration of the game**
  WAVA has been designed to be a short game, currently at an approximate playtime of thirty minutes. Making the playable chapters longer would result in the plot points and cinematics being more impactful and meaningful to the player.

- **Publish and promote the game**
  Once the previous aspects have been worked on, the game will be self-published under the studio North Oriole Games. Prior to its release, the game will be promoted through our different social media accounts.

10. Bibliography

Books

- Adams, E., Dormans, J. (2012). *Game Mechanics: Advanced Game Design*. New Riders.

- Skolnick, E. (2014). *Video Game Storytelling: What Every Developer Needs to Know about Narrative Techniques*. Watson-Guptill.

Websites

- Unity Technologies. (2022). *Unity User Manual: Version 2020.3*. https://docs.unity3d.com/2020.3/Documentation/Manual/UnityManual.html

Videos

- Code Monkey. (Dec. 2019). *Parallax Infinite Scrolling Background in Unity* [Video]. https://www.youtube.com/watch?v=wBol2xzxCOU

- Antarsoft. (Aug.2020). *Unity 2D Platformer Tutorial 7 - How To Create 2D Smooth Camera Follow Script For Your Player* [Video]. https://www.youtube.com/watch?v=_QnPY6hw8pA

- Daniel Wood. (Sep. 2021). *Unity 2D Platformer Tutorial 20 - Respawn checkpoints* [Video]. https://www.youtube.com/watch?v=VGOVe_adFMc

- DevDuck. (Apr. 2019). *Unity 2D Rain Tutorial* [Video]. https://www.youtube.com/watch?v=xkB6yzCBfgw

- Sleepy Lava. (Jun. 2020). *Unity tutorial - 2D Rain with splash* [Video]. https://www.youtube.com/watch?v=QP8zj-JQgmI

- Brackeys. (Aug. 2017). *How to use GitHub with Unity* [Video]. https://www.youtube.com/watch?v=qpXxcvS-g3g

- Brackeys. (Dec. 2017). *SETTINGS MENU in Unity* [Video]. https://www.youtube.com/watch?v=YOaYQrN1oYQ

11. User manual and installation

11.1. Installation and execution

Please, note that this version only works for the operating system Windows.

- Download the ZIP file “WAVA”.
- Extract the files into any directory.
- Access into the directory and do not delete any of the existing files.
- Execute the file “WAVA.exe” (see Figure 11.1).

Figure 11.1. WAVA's executable icon

11.2. Game manual and instructions

Please, note that in this version the menus only work with the mouse.

- After the initial splash screen animation, the main menu, which includes four buttons, is shown.
  - **Play:** To go to the level selector.
  - **Options:** To change the game’s view settings.
  - **Credits:** To watch the credits’ scene.
  - **Exit:** To quit the game.

- Inside the level selector, click on any chapter to load it (see Figure 11.2).

Figure 11.2. WAVA’s level selection screen

- The different chapters will transition on to the next one starting from the level we have selected in the main menu. At any given moment, the player can access the pause menu (see Figure 11.3), which includes four buttons:
    - **Resume:** To close the pause menu and go back to the game.
    - **Controls:** To check the game's controls.
    - **Options:** To change the game's view settings.
    - **Exit:** To go back to the main menu.

Figure 11.3. WAVA's pause menu

11.3. Controls

To see the different actions that can be done in the game, as well as their respective bindings for each different type of controller, see Table 11.1.

Action	PC	PlayStation	XBOX
Horizontal movement	A-D / Horizontal arrows	Left joystick	Left joystick
Underwater vertical movement	W-S / Vertical arrows	Left joystick	Left joystick
Run	Shift *(hold)*	R2 *(hold)*	RT *(hold)*
Climb	W / Up arrow	Cross	A
Crouch	C	Circle	B
Protect	Space	L2	LT
Pick up / Interact	E	Square	X
Build and recover auxiliary platform	E *(hold)*	Square *(hold)*	X *(hold)*
Interact with the Light	Q	Triangle	Y
Pause	Escape	Options	Menu

Table 11.1. Controls and bindings for each action