Ca' Foscari
University
of Venice

Bachelor's Degree
in Computer Science

Final Thesis

CODING A TIME TRAVEL

Analyzing a time travel based videogame development:
concept, design and implementation

Supervisor
Ch. Prof. Alvise Spanò

Graduand
Melania Gottardo
Matriculation no.
874240

Academic Year
2019 / 2020

Ca' Foscari University of Venice

Bachelor's Degree in Computer Science

Coding a Time Travel

Analyzing a time travel based videogame development: concept, design and implementation

Final Thesis

Melania Gottardo

Matriculation no. 874240

Supervisor
Ch. Prof. Alvise Spanò

A.Y. 2019 / 2020

Table of Contents

Table of Figures

Table of Graphs

Table of Code Snippets

Chapter 0
Abstract

For the last fifteen years, Unity, a free cross-platform game engine, has been giving non-professionals the opportunity to jump into the gaming market by allowing independent and amateur developers to deal with each stage of videogame production. An essential ingredient is a solid game structure, which basically consists of a versatile code infrastructure and an accurate level design. These can further be enhanced by a captivating storyline, attractive graphics and visual effects, intriguing soundscapes and music. The code infrastructure in a game pairs with the business logic in a general software, encapsulating data structures and their behaviour. On the other hand, level design determines how the pieces blend together and integrate with the environment, aiming at either pleasing, constraining or challenging the player. Videogame development is a work of science, technique and art combined together in order to accomplish a primary goal: setting up an enjoyable experience. The purpose of this thesis is to report the crucial aspects in the development of a real-world videogame: a time travel based puzzle game. Among the main challenges, arranging levels that have to be entertaining in order to be successful, stuffed with different parallel timelines and space-time paradoxes altering the level structure itself by preventing or allowing the player to perform certain actions.

Chapter 1
Overview

The purpose of this Chapter is to give a general overview of what the production of a videogame covers. We will start from the necessary foundation for the development of any kind of game. Then we will discuss the idea's origin behind *Between*, the specific videogame on which this thesis is about. Finally, we will see various steps needed to transform a simple abstract concept into something concrete, a finished product.

1.1 - What is Unity?

1.1.1 - Overall introduction

Unity[1] is the most deployed game engine worldwide, since its launch in 2005. It is uniquely designed to embrace a massive amount of game typologies and produce them for cross-platform distribution. Developing for every device with various input controls at once has been crucial in growth of Unity. This led both non-professional developers and AAA[2] studios to rely on it for 2D, 3D, VR and AR games.

Due to its popularity, Unity has solid documentation, tutorials and a huge quantity of followers that create successful libraries daily to share with the entire community. Recently, Unity is becoming even more appealing thanks to an increasing improvement in the team collaboration control.

Unity is also an IDE[3]: its interface provides the user with all the tools needed, including a cloud-based service for multiplayer games and monetizing. Usually, the easiest way to operate into scenes is to drag and drop or transform the components into position. Unity projects are also organized in customizable folders where the developer can sort all the elements he uses, according to his preferences.

[1] Unity, Unity Real-Time Development Platform | 3D, 2D VR & AR Engine.
[2] A triple-A video game (AAA) is developed and financed by a large company with a solid budget.
[3] IDE stands for "Integrated Development Environment".

The most common programming language for scripts in Unity is C#. These scripts can be coded in Visual Studio tanks to a built-in integration. As an option, Unity gives the possibility to use MonoDevelop, instead of Visual Studio, and to code the scripts in JavaScript.

There is double access for standard and premium users. The former has free access to everything mentioned above, while the latter, in addition, has access to Unity's source code in C++, developer support and a flexible licensing plan, as well as being able to distribute their products on a wide scale.

1.1.2 - How it works

Since the very beginning in the Unity Hub, Unity let the user choose in which type of scenario[4] the development will be. Once the user opens his new project (see Figure 1), he can immediately adjust and customize the interface and its panels.

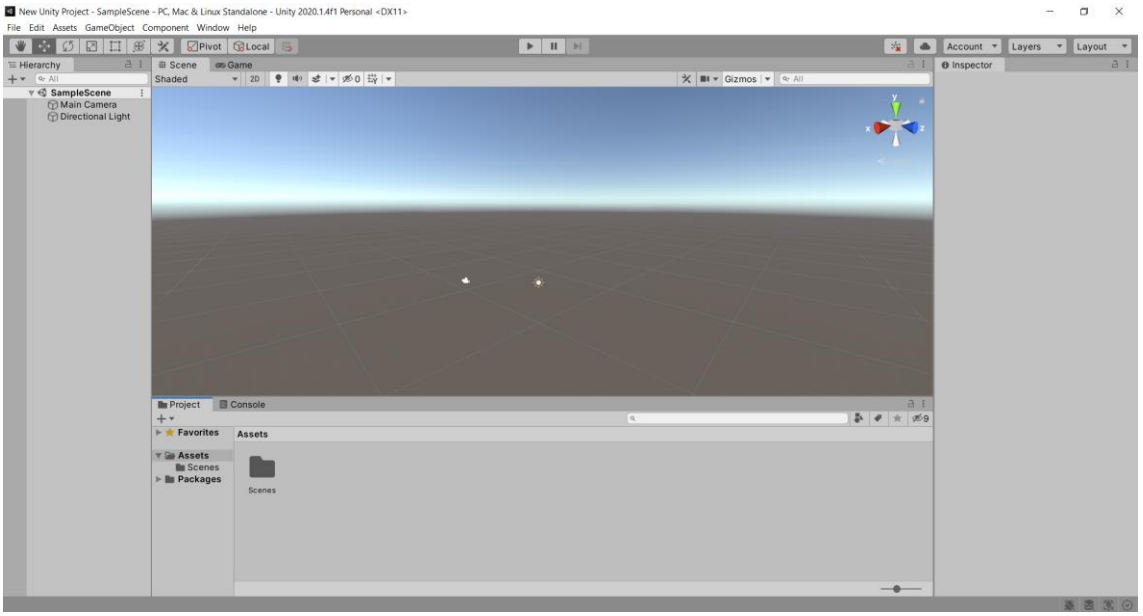


Figure 1. Unity new 3D project's workspace

Each user can move, resize and add or remove tabs, too, in every moment, based on his needs. For instance, an artist probably will add the *Animation* and *Animator* tabs which are not present in the initial setup. By here no code is required to start creating items, setting their values and attributes and putting them into the scene. Every single window has its own purpose and functionalities. Now, we will consider the main panels and their properties.

[4] E.g. 2D, 3D, mobile, etc.

Figure 2. Hierarchy tab

The *Hierarchy* tab (see Figure 2) contains a real-time list of all the GameObjects that stand in the current scene. In order to statically add a new element, it must be created here as a GameObject, perhaps with a more specific identification. It can then be inserted into the hierarchy as a child of another element with its own children. When a GameObject is here, it can be selected and modified.

Figure 3. Inspector tab

The object's properties and attributes can be manipulated in the *Inspector* tab (see Figure 3). For every GameObject, in the Inspector tab there is a list of the object's components and scripts that are attached to it. They are modular and completely customizable in the order as in the state, number or values.

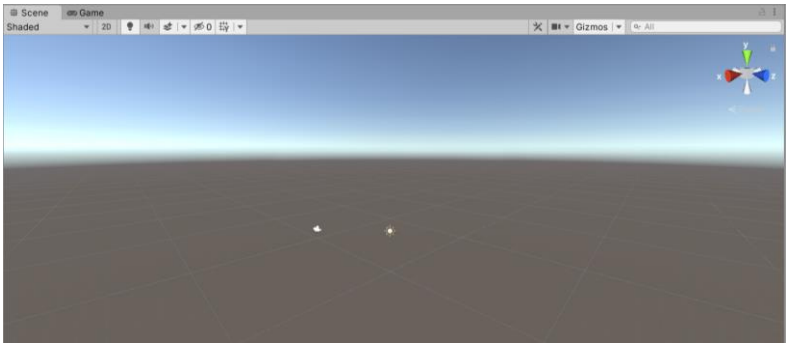


Figure 4. Scene tab

The *Scene* tab (see Figure 4) is the largest and most important panel in Unity. The tab's name suggests that this is the view of the entire current scene. It also shows a semitransparent grid guide to precisely collocate each item. All the GameObjects appear here in the right spot once the user drags and drops them into position. They can constantly be transformed and interacted with in many ways.

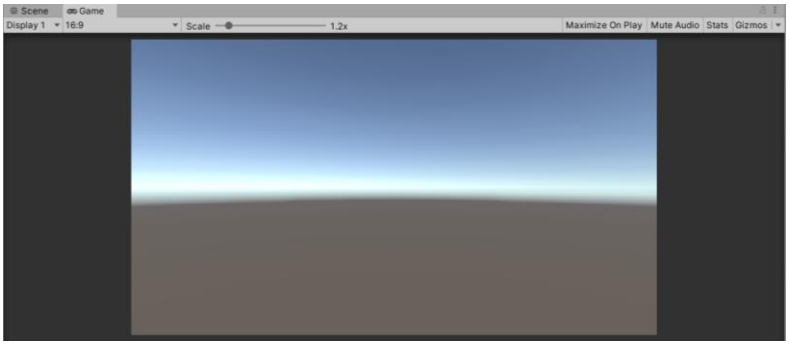


Figure 5. Game tab

While in the Scene tab the camera can be seen and transformed, the *Game* tab (see Figure 5) has the camera perspective, without the possibility to adjust anything. This is the real player view and it is very useful for testing the game.

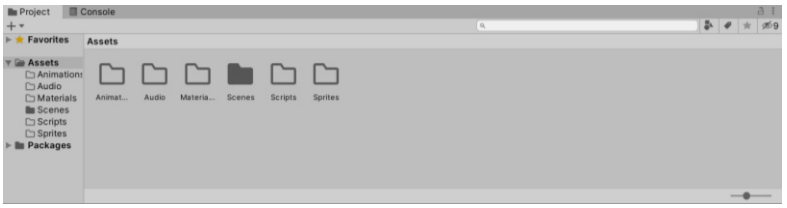


Figure 6. Project tab

The *Project* tab (see Figure 6) shows the entire project structure and how it is organized. The user can create many folders and files he wants by dropping the

sources[5] there or by creating them right there (e.g. C# scripts). Whenever a resource is needed, it has to be added in this panel first.

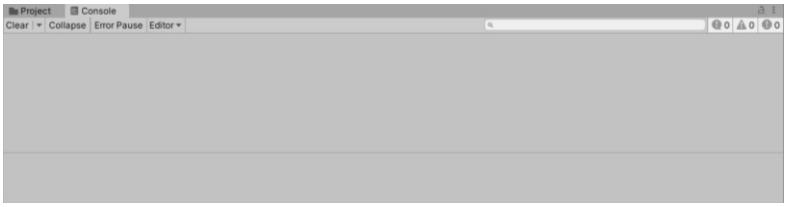


Figure 7. Console tab

The *Console* tab (see Figure 7) is where the developer will know if there are issues with the software or if the scripts contain any warnings or errors. It could also be used on purpose: a programmer could take advantage of it by dynamically printing debug messages.

An essential feature of Unity is the prefab mechanic. Unity allows the user to create the prefab of an item and store it in order to be reusable, editable and make the designers' life a little bit easier. There are also many other aspects that give credibility to Unity's software such as textures, materials, real world physics simulation, a powerful rendering and post-processing engine, etc. In view of this, Unity is a complete machine for both coders and artists who desire to develop incredible videogames.

1.2 - The idea behind the videogame

1.2.1 - Where the idea comes from

The idea of this videogame came out of the desire to divide a single space into multiple visions. Taking a level with precise start and end points, the path will be linear, and even if there were more paths to reach the goal, the logic remains one-dimensional (see Figure 8). Facing a level in a linear way could be boring in the long run, considering it an approach already seen and revised in many games since the 1980s.

[5] E.g. images, videos, audios, 3D models etc.

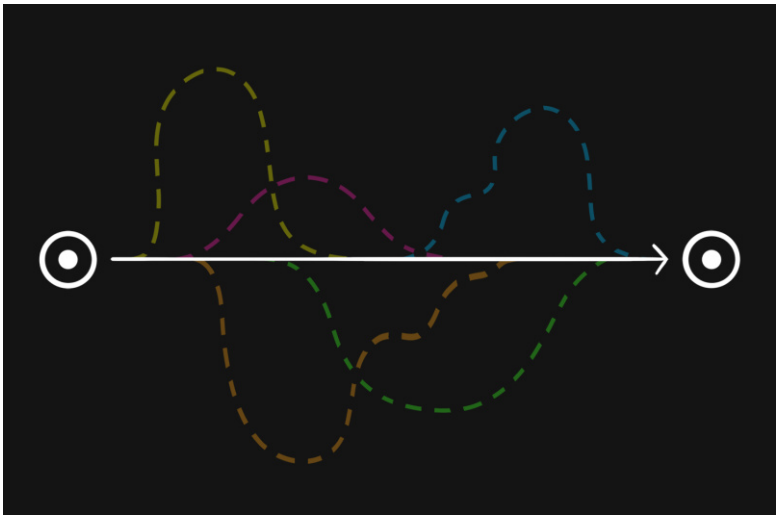


Figure 8. One-dimensional multi-path level

At first, the idea was to approach the levels forward and backward. The challenge for the player would be seeing the obstacles in two opposite ways, such as using them to climb instead of descending, or blocking a dangerous area not accessible before etc. The concept was interesting, but rearranging the same level over and over again can be easily tedious.

A second idea was to provide the player with an instrument, maybe glasses, with multiple lenses which glimpse different elements of the scene depending on their color. Certainly the nonlinearity was there, but the absence of certain objects only depending on which lens was used was a bit falling. In addition, it did not reused the same items with different behaviors, which was a successful point of the first idea.

Taking back the first idea with a new perspective, led the player to get to the end and, after twists and turns, finding himself again at the beginning of the level, but into another time set, with scene elements shifted, absent or broken. The idea was not yet convincing, but it was closer to the definitive one.

Blending the second and the third concepts together, the path of multidimensionality brightened: time travel was the key.

1.2.2 - The core of the videogame

The point was to create a portable device that allows the player to time travel. The player does not have it at first, but once he reaches and obtains it, he should be able to go back and forth between present and past.

Depending on the time in which he is, the elements would be arranged differently, could be broken, disabled or enabled. Every item is both helping and blocking the player, according to the time set. The challenge for the player is to find the right way to deal with the environment, because it is not possible to beat the level in a single timeline. Actually, the goal can be opened only at a certain time period and to unlock it is required to go back in time when another device was still active and get close to it.

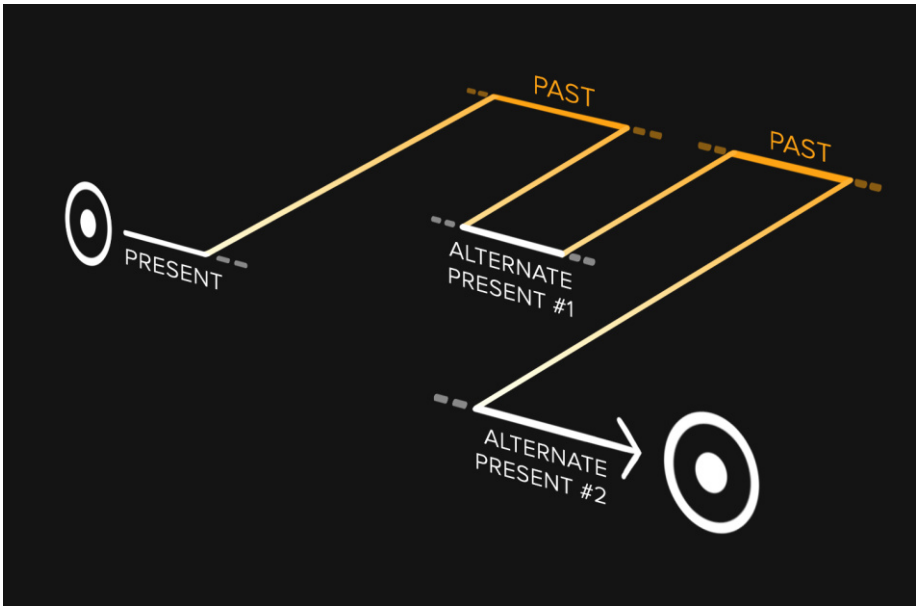


Figure 9. Multidimensional-path level

Doing so, there is a single way out, but the path to exit the level is divided into pieces around time sets (see Figure 9). The player can not simply visualize the puzzle solution as it is not in front of him, especially if he occurs in space-time paradoxes that create multiple timelines. Indeed, the solving flow shows itself through the time changes and may constraints the player to restart a level if he traps himself.

1.2.3 - Exploring the concept details

As the game evolved, we added and improved more and more details, here are some examples.

Starting with three basic and essential objects, present from the beginning: entry and exit portals, access keys and an activating transmitter. The behaviors of these objects have always been clear:

- Portals act like portals. The peculiarity is that the entry portal disappears as soon as the player enters the level, while the exit one is blocked at any time, but it can only be turned on in present time.
- The transmitter is a fixed machine that activates the keys when the player nears it. However, it is broken in present time, so to get it working the player has to travel to the past.
- Keys open doors and portals, but they can be active or not; only if they are active, they unlock the door or portal that corresponds to their color. When the player uses a key, it will no longer be available. The keys also possess the time travel mechanism, so the player is allowed to switch between past and present only when he has at least one key in his inventory.

Talking about the rest of the level design, consider as an example the first item designed: a simple box that aided as a portable step to reach higher spots in different parts of the level. Towards the later stages of development, we have attribute the box a weight and a specific friction. Furthermore, it was established the maximum height within which it is possible to drop the box without breakings. Beyond this, we have designed and improved other items over time, such as iron pipes or electrified balls that can be switched on or off, depending on the time set.

1.3 - Building a videogame from scratch

1.3.1 - Where you want to develop the videogame

First of all, you need to know where you want to develop the videogame. You can totally start from scratch and create anything yourself, or you can develop the game through an existing game engine. If the developer is a big studio that desires independence, or the main purpose is not making the game, but finding an alternative way to achieve it, then creating your own game engine is better. In other cases it is recommended to use an existing and well tested software.

Over the years, several game engines have been developed, such as Unreal Engine, Amazon Lumberyard, Godot or Cryengine, but Unity was used for the realization of this videogame. Why choose Unity? Among reasons, surely because it is free and there is plenty of written and video material besides a powerful community, suitable for newers in the world of videogame development. Moreover, Unity is an excellent engine that provides an intuitive interface and a

solid structure of physics and rendering with a proper animation setup. Nonetheless, the scripts' programming language is beginner-friendly.

1.3.2 - Learning, composing and testing

If you have chosen a software, as in this case, you need to learn how to handle it, before jumping headlong into the project. The game defined in Section 1.2 is a 2D platformer, so we will discuss how to approach this type of scenario.

Before making the game, it was necessary to execute some tests to figure out how to deal with the software and the various tools that Unity offers. Particularly, first we had to understand the interface and its tabs (see Subsection 1.1.2). Next, we wanted to manipulate something, so it was necessary to import and organise the basic assets. Then, from them we started creating GameObjects and placed them into the scene. Once we had the character and something to interact with, we created and attached a script to it, ready for the first tests of movement and interaction with the environment.

When the elements started to work together, it was time to fix the scene trivially and add some animations, too, with the aim of working on something that began to make sense. A fundamental point was being able to use the collision detection mechanism that allows the interaction between two entities. This is possible thanks to the addition of GameObjects' components such as *Rigid Body* and *Collision Box*, but there are many more to explore and add to each item.

At this point, we could proceed with something more advanced with the code development. Another key element was the understanding of how much code was needed and what was the smartest way to exploit it. In the first place, it was useful to vary the values assumed in the scripts, but then we no longer needed to change them, so it was better to remove them from the modifiable code during editing or testing.

Speaking of testing, if it is so important in any possible area, in this field it is so much more crucial, as if a programmer writes thousands of lines of code, but then the character does not even move: it makes zero sense. Tests must be continuous and follow any slight modification of the project, whether it is code, composition, graphics, animations, etc. Not only pieces had to act correctly, but they had to fit well with each other. After finishing with the basic elements, we added the other framing ones and created scripts for them, too.

Afterward that we were quite satisfied with the scene, it was the turn for temporary things to become finished and permanent. Then, it was necessary to start using

other Unity mechanics, such as switching between scenes, utilizing prefabs to modularly set scenes and including a graphical interface via Canvas. When the game has begun to take form, we added other elements like sounds, other animations, special effects (e.g. particle systems) and fixed the user interface. At this point, it was possible to improve what was done by cleaning pieces of code, adding levels, making menus more stylish, and finally exporting the project for the desired platforms.

In the light of the above, having in mind the general process of how the project was accomplished, it is about time we linger on the main and most fascinating facets. We will examine in detail how the game has been shaped to better wrap the time travel concept. Similarly, we will explain how this has been meanwhile accompanied by practical encoding work of the structure and the behavior of each component.

Chapter 2
Game Design

In this Chapter we will explore the concept of game design. We will make considerations about its different disciplines and how they give a sense of cohesion, when combined. More specifically, we will study how to involve the player on different levels, whom are narration, gameplay and environment. Then, we will emphasize the game designer's role in relation to game feel and levels' nature. Next, we will pass on its numerous skills to see more closely what they offer to the user. Meanwhile, we will take a look at the implementation of these techniques in our game, such as emotional involvement, rules design, content design, world design and level design.

2.1 - User experience

Addressing the development of a videogame can take many ways, most of them are *market-driven*, because of the dependence on the fundings invested in their publishing. Others rely on the hardware and systems behind the game, such as an ad hoc software or new technological breakthroughs to improve game mechanics, hence the term *technology-driven*. Story-driven perspective is one more manner of conceiving work, highlighting the narrative of the outcoming product. In another few cases, focusing on visual arts and how the game can highlight them more is the key, thus defining the category of *art-driven* games.

Those are all good approaches, but, in turn, the videogame we are considering brings attention to the user and his gaming experience. Each choice is made to get the player more and more involved into the game, learning its mechanisms and scrambling to achieve the goal and continue. Thereby, this whole thing leads the developer's point of view apparently towards a *design-driven* videogame. Important big studios with the means and fundings use this method in order to be able to choose a qualitative product. They create innovative teams always in communication with each other, exchanging opinions and taking full advantage of every department that contributes to the success of the final product.

2.1.1 - Narrative structure

A good slice of videogames features narrative elements that support gameplay, with diverse extensions from game to game. When an event is triggered, for example, giving the reason why it was triggered involves the user at a higher level, making the story touchable. Interactive story games focus on the narration, obtaining an almost non-existent gameplay from time to time. This technique simulates watching movies, with the major difference that one or more characters are acted by the player. Part of videogames, often mobile games, emphasize entirely on gameplay, ignoring the existence of a story. Unfortunately for them, they are minor games, often repetitive and obscured by more popular games with a defined story. This one can be very basic or really complex, the important thing is that it provides the best user experience.

Between was born from the idea of time travel, not a time travel story. Giving the context is essential to make the player understand what he is going to play, otherwise he would be shocked to see scene elements move on command. To supply a context, you need a story to tell. Telling a backstory from top to bottom at the very beginning and then letting the user play would solve the problem, but it would not be interesting and surely much less intriguing. The optimum strategy was defining a narration not too complex and providing the player little information at a time.

The logic required for level-solving added to learning game mechanics and gaining knowledge about story pieces can not be excessive. Balance between the various factors is the key. In fact, at the beginning, when levels are easier, it is a good time to explain more techniques and offer information about the backstory. Contrariwise, in the midst of a difficult level set, it is not proper to introduce many new procedures or to revolutionize the storytelling.

The story of this game starts with the character who is catapulted out of a portal into a sealed room made out of stones, bumps his head and does not remember anything. Suddenly, he sees the portal where he came from disappearing; he is disorientated and needs to look for a way out. From here, he will find a key to open the exit portal, but it only leads to another room. Slowly, he will understand how time travel works (see Section 2.1.2) and will continue his journey, waiting to find a real escape. Observing the setting and the engravings on the stones, the character will realize that he is in an unknown spooky world. At the end of his journey, he will find himself in front of a broken portal in the present, which is fixed in the past. He knows that keys only activate portals in the present, so there are five possible endings at that point:

- The player discovers that with the collected dice he can open the portal in the past, but he does not have enough. Being the room completely closed and having neither food nor water, he knows that inevitably he will die. So, the player chooses to commit suicide and end it there.
- The player discovers that with the collected dice he can open the portal in the past, but he does not have enough. Being the room completely closed and having neither food nor water, he knows that inevitably he will die. Then the player chooses to wait and let himself die.
- The player discovers that with the collected dice he can open the portal in the past, he has enough to do it and gets into it. Before entering, the player does not change the time set, thereby he finally manages to exit, but he is not in his time. Once out of the stone room, the time travel mechanism no longer works, so he is free, but he finds himself locked forever in a time that does not belong to him.
- The player discovers that with the collected dice he can open the portal in the past, he has enough of them to do it and gets into it. Before entering, the player changes the time set and enters the fixed portal in present time. Having created many alternate timelines while traveling through time, he does not know what is on the other side. This time he was unlucky: the portal sent him back to the first portal, where he bumps his head and has to start all over again.
- The player discovers that with the collected dice he can open the portal in the past, he has enough of them to do it and gets into it. Before entering, the player changes the time set and enters the fixed portal in present time. Having created many alternate timelines while traveling through time, he does not know what is on the other side. This time he was lucky: the portal takes him out of there, in his time, and finally he finds out how he first entered the portal.

2.1.2 - Gameplay

Gameplay is the reason why a game is called that. Without it, a game just can not exist; interactivity is the only thing you just can no do without. Imagine playing chess without being able to move the pieces: there would be only a chess board with pieces in position to observe. Engagement can take place in numerous dissimilar ways, departing with all possible controllers on the market[6]. Depending

[6] E.g. keyboard, mouse, joystick, gamepad, touchscreen, remote control, etc.

on the kind of controller, the type of interaction changes: in a strategy game you have to give well-targeted input, in a fighting game you have to press a combination of keys, in a simulation game you have to physically imitate the character's movements, in a game attended by a voice assistant you will have to talk, etc.

The purpose of the gameplay varies in kind. First of all, entertainment: videogames were borned with this aim and the vast majority of them point to this. Then, education: more and more studies have shown that learning while having fun facilitates it; for this reason, the market is interested in this kind of audience as well. In addition, athletes will know that the exploitation of videogames has increased training opportunities, when actual conditions are unfavorable (e.g. ski simulator when there is no snow).

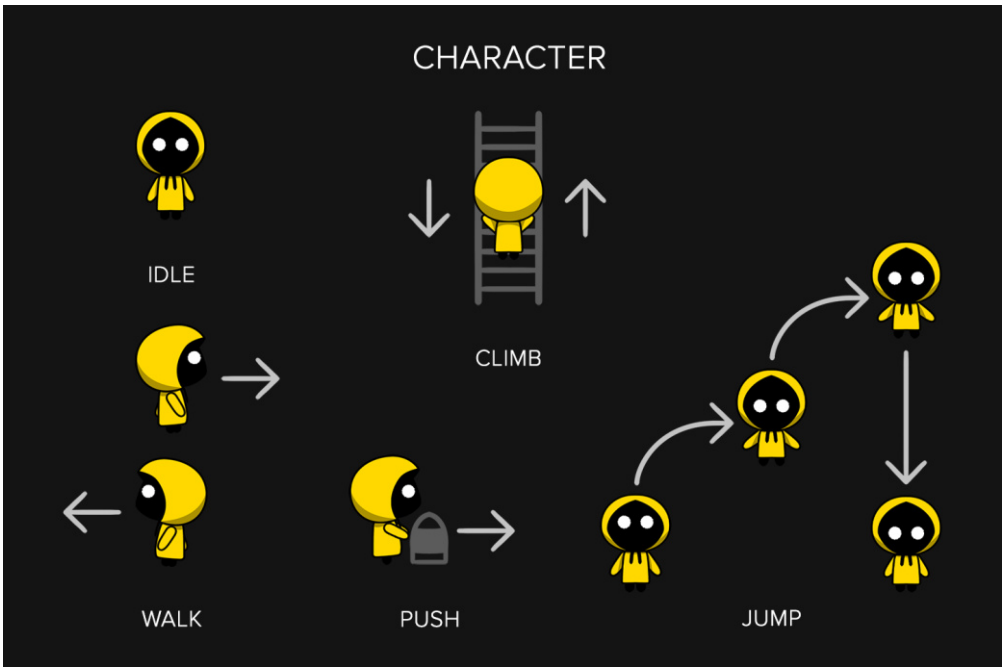


Figure 10. Basic character's actions

In *Between*, the gameplay begins with a very well known and intuitive base, the platformers. The genre of platforming games was born in the 1980s with Donkey Kong[7], and it still remains a very popular genre. Being very recognizable, it was easy to insert some particular dynamics, useful for the purpose of the game. The character of our game can walk left and right, go up and down stairs, make a single or double jump and push objects (see Figure 10). In addition to these basic actions, the player can switch between present and past by pressing the "T" key, if he owns at least one key, and when he gets close enough to a collectible object, he picks

[7] *Donkey Kong* (first version), July 9, 1981, Arcade, Nintendo, Japan.

it up. Also, when he stands in front of a running transmitter, he activates his inactive keys, while, when he approaches the exit portal with an active corresponding key, he opens it (see Figure 11).

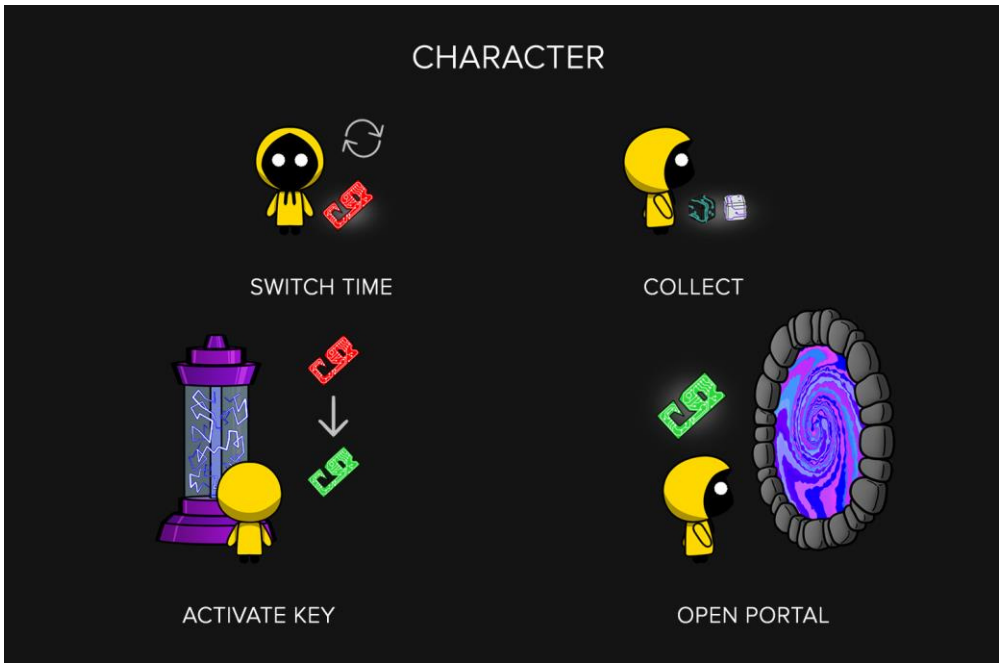


Figure 11. Advanced character's actions

The instructions are simple and limited, in order to be absorbed quickly and become natural or instinctive. The gameplay, in this case, leaves much space to the user's logic, allowing him limited maneuvers. It will be the player's talent and experience to make him move forward and improve more and more, giving him a personalized, constructive and educational gaming experience.

2.2 - The game designer

The role of the game designer can start in two ways: designing a concept or receiving a concept to work on. From this stage, the road splits into a series of actions that also include gameplay, levels, characters and controls management and designing the features of each aspect. Multiple decisions are in the hands of the game designer who has the responsibility for the game feel[8]. He cares as much for the general framework as for the details.

[8] The "game feel" is the overall experience the player perceives interfacing videogames. *Swink, S. (2008), Game Feel: A Game Designer's Guide to Virtual Sensation, CRC Press.*

During the process, the main goal is to create a game appealing to the imagination of the user. Beginning from a draft, he must arrive at a detailed list of elements to implement. As an architect, the game designer must study every independent element on its own and in relation with the rest to obtain a robust and well-designed structure. The general composition has to be clear, including knowing which factors are required for a certain component to be able to play its full role. He must be collaboratively with other departments[9] to create every single version of the game, starting from the prototype.

The tasks of a game designer do not end here, because he must constantly return to the concept to refine and smooth the corners at each step, as the game evolves. He must follow the development and test each version of the game to verify that the pieces of the puzzle fit well and observe the rules he has established. Meanwhile, he must also keep an eye on the quality, playability and easiness of learning the gameplay mechanics to guarantee the optimal user experience possible.

2.3 - Branches of the game design

As we have just said, a game designer divides his role into many different areas. We will go into details of some of these areas to explain what has been done in our specific case with the videogame *Between*.

2.3.1 - Rules and content design

Like in any game, there are rules to be respected, starting from the game's boundaries and the number of players. The game manual provides to the player is nothing but a reduced version of all the rules established by the game designer. The rules design pairs with the content design that opens the characterization sphere for each individual element or action. This last facet is important to give depth to the game, to increase its value and its overall complexity. So, this intertwines the overall functioning and the peculiarities that transform a game from anonymous to recognizable.

What rules and content have been developed in this case? The 2D context for a single player had already been chosen, so, from here, we opted for a development on a horizontal plane limited at left and right ends with the possibility of jumping

[9] If the project is a one-man job, like in this case, it is evident the collaboration between diverse departments, as all of them are just one person.

on platforms. The player's goal is to exit the level, but, while doing that, he may die or find himself forced to restart the level because of a dead end.

Once the basic rules were clarified, it followed the building of the skeleton that every level would have. Five elements were absolutely indispensable for the completion of the levels: the player, the entry portal and the exit one, at least one key to unlock the exit portal and the transmitter for the key activation (see Figure 12). At that stage, drafting these objects' tasks and characteristics (see Section 1.2.3) was necessary, in order to build the rest of the surroundings around them.

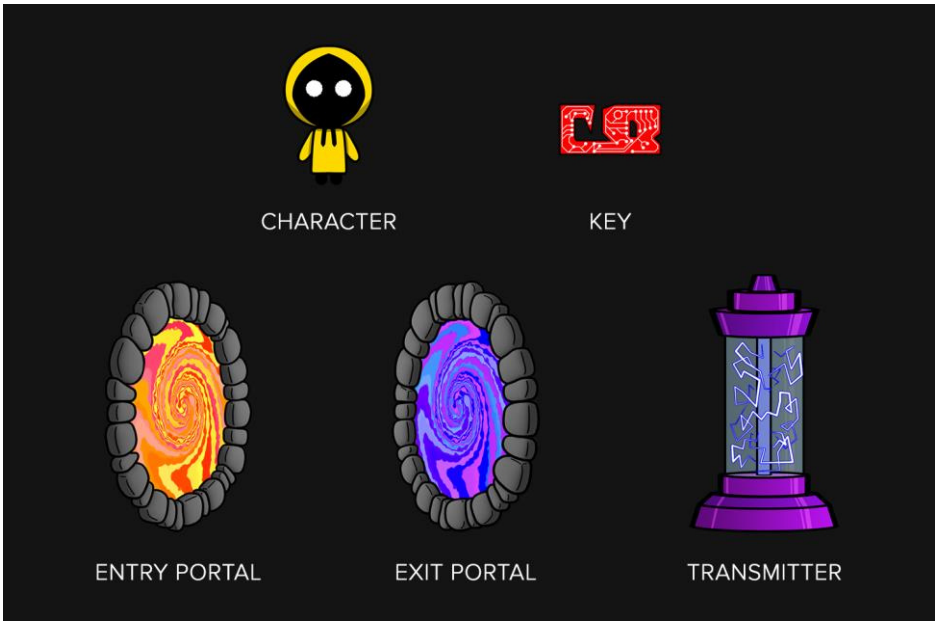


Figure 12. Fundamental levels' elements

Finally knowing what we were dealing with, it was possible to expand the details. First, we established some further character's features, such as his humanoid physiognomy, his walking speed, the jump power, the presence and amount of multiple jumps and possible directions of movement. Simultaneously, sizing and proportioning the standard level and its main elements with their relative locations were the priority.

The next step was to define new objects or actions in relation to what had already been created. The first item was a box: the player could climb on it, push it and drop it from an elevated platform. Being able to play with time demanded a movable object at will for reaching different useful spots in diverse moments. As an alternative, then we created another item, with different physical features. Indeed, an iron pipe can only roll and it has no edges, as opposed to a box (see Figure 13). Afterward, these two items were characterized by real world properties. The box has a lighter weight than the iron pipe, conversely, the friction of an iron

pipe is less than that of the box. Ultimately, a wooden box is much more fragile than an iron pipe, so we established a maximum height from which the box can fall without breaking.

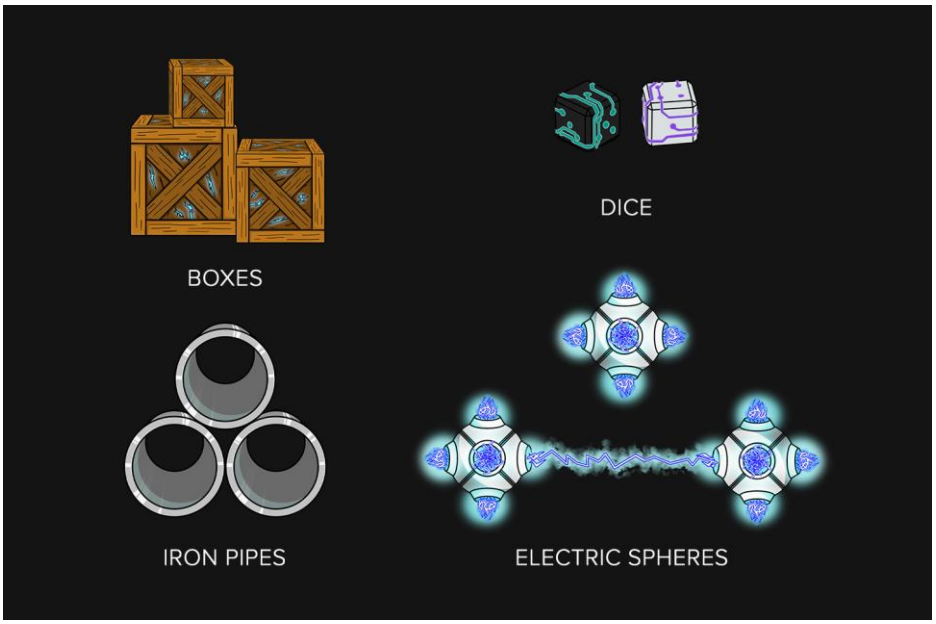


Figure 13. Additional levels' elements

Previously, we saw that the player can die within a level. Actually, it was interesting to integrate situations that allow that. Counteracting regular objects with dangerous objects required a well distinguishing impact on the player, such as static objects versus alive and interactive objects. To maintain the theme, spheres charged with electricity would be ideal and they would be lethal to the touch if lit. In fact, unlike the transmitter, where electricity flows into the glass, here the player is not protected. Depending on the time set, these dangerous balls can be active or not, allowing the player to stem some obstacles. For variety, it was thought to give a double scope to these electric spheres, thus they could exist alone or create an electric beam if aligned (see Figure 13).

A final added element were dice (see Figure 13). Having an inventory built just to collect keys, we could use it for gathering something else as well. Dice are collectible objects that are located within the levels. Their count remains saved, and if they hit a certain amount, the player can utilize them to unlock portals at the end of the game.

2.3.2 - World and level design

When you construct a world in which to set the video game, you have to think about two views that concern it: the world and levels' composition, if they exist,

and the appearance that the world will exhibit. Speaking of world design, the player needs to see a coherent and uniform game's theme. This field also covers the decision to include or not a world map, whether to make it fully or partially visible, whether to show missions, areas or levels. In short, decide the measure in which the player knows where he is. In case there were levels, these have to be deepened at a level design phase. Therefore, the designer must understand how they are arranged and how the player can access them, beyond studying the inner structure of each level. This task is fundamental, in the first place, to give the player well-assembled levels to play and, in the second place, to understand how to direct or deflect him from his destination.

Regarding *Between*, the world was clearly made up of individual levels. Since the background story talked about time travel, the levels had to be linked by the same continuous common story. In light of this, we thought levels as a sequence with incremental difficulty (see Subsection 4.3.2). The sequence is linear, hence we established the player could never replay a level already faced (unless a game restart), avoiding linearity interruptions. This choice was also dictated by the internal structure's complexity of individual levels: it was better not letting the player focus on which level to deal with. Always keeping the player's focus on solving the level in front of him, we decided not to provide any progress suggestions, leaving the player to follow the flow of events. The only information given to him is the status of his inventory and the actual time set.

According to the progress of the scene elements design (see Section 2.3.1), the game spirit was increasingly directed towards an alien world with unknown technologies of which you do not have full vision, but just a few hints (see Figure 14). The atmosphere evolved in this sense: starting from time travel, portals and electricity, we needed something to carry the player into a mysterious environment, full of both magic and technology. Levels have thus become rooms built with slabs and engraved stones, where crystals and powerful machinery coexist. Props also contributed enhancing the general mood, adding fluorescent substances and circuits to emphasize the alien context. Everything is adorned with magical dust and a slightly dark atmosphere (see Figure 17). In addition, when traveling in the past, the level presents a sepia-colored patina, to induce a sense of antiquity inside the player.

Figure 14. Atmosphere concept

Having found the mood, we moved on to find a good design for the levels themselves. The first factor to determine was the length of the levels: a level does not appear entirely in the player's screen, the camera will follow him where he will move left or right. Being rooms, the player can jump to higher places, but will not have to climb anything; in fact, the camera will not follow him in height. For the placement of scene components, criteria were established: the portals would always remain at the same height, one on the far left and the other on the far right. However, everything else could have been placed anywhere, on the ground or above raised platforms. To learn more about the level design process and their generation through multiple timelines, please refer to Section 4.3, as this topic requires more in-depth observation.

2.3.3 - Other branches

There are many other ramifications of game designer's role, but for the purpose of this thesis, they will not be handled. Anyway, we quickly see a couple of them, whom are audio design and user interface design.

The audio design incorporates many processes, from soundtracks, to background music, to special effects, but it also includes dialogue voices, interaction sounds and anything that can produce a sound. In this videogame there are no dialogues, but background music and sounds generated by interactions with the interface or

the environment. The background music had to reflect the atmosphere allowing the player to further immerse himself in the game. It had to be enigmatic, slightly gloomy, but also a bit magical. On the other hand, sound effects had to respect the mood, but they had to stand out, to give the player sound feedbacks. Moreover, they had to reflect what had triggered them, for example, collecting a key is a joyful event, in contrast to breaking a box.

The last thing we are going to talk about is UI[10] design. The UI is essential in almost all videogames, as it is unavoidable to create menus, start screen, final screen, transitions, to show inventory, character's health and stamina, game maps, etc. The UI graphic has to be consistent and uniform with the style of the game, as well as well readable and it must not cover important information. In *Between*, the UI has been used for two main reasons: creating main and pause menus and, within levels, showing the inventory and the time set.

Figure 15. Main menu or "Home"

[10] UI stands for "User Interface".

Figure 16. Pause menu

With regards to menus (see Figures 15 - 16), the graphic shows a strong magic mold, with an obvious reference to portals, especially in colors. Menus' content changes by offering different features. In the main menu, you can continue or start a new game, you can access the options and customize your character. Instead, in the pause menu, you can restart or continue the level, you can return to the home or access the options. The style is continuous between the two menus, but the contents change. For its part, the level UI (see Figure 17) is very minimalist, it does not cover any element and does not draw attention. Even the font used throughout the UI is the same to give recognizability and cohesion. Eventually, we have added a last small detail to the UI, to leave nothing to chance: we have customized the mouse cursor (see Figure 17), providing a graphic consistency with the rest of the scenario.

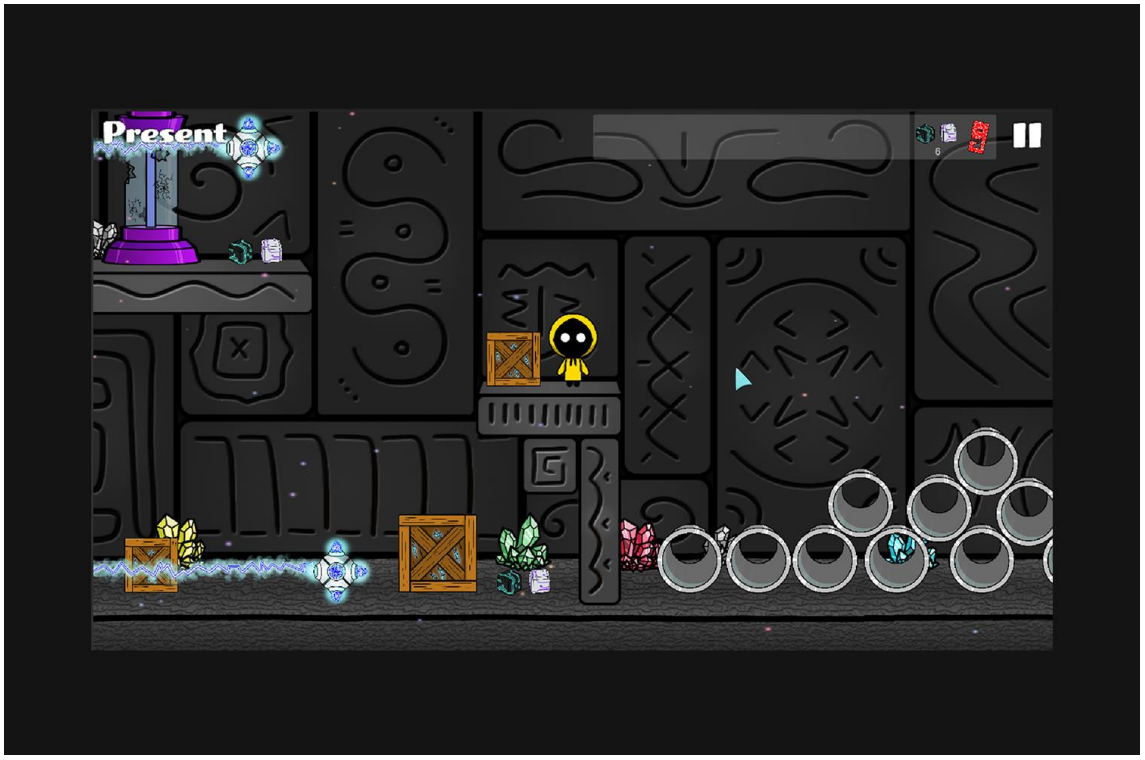


Figure 17. Regular level with UI and custom cursor

Basing on what we said, we can grasp the flow which allows the game to be fluent and functioning smoothly at the players' eyes. Now, having understood the behavior of the various pieces, we will see why we are able to elicit these behaviors and how they are generated. Finally, it is time to take over the game programmer's work, which means the real code that allows the character and the objects created to operate.

Chapter 3
Implementation

The intention of the third Chapter is to provide a clarification on the backbone of game mechanisms, namely its implementation. First, we will illustrate what an object-oriented language is and why we used it. Later, we will observe the structure's branches in detail. Finally, we will talk about the most attractive topic in this area, the code: we will touch only the most remarkable points, commenting on targeted Code Snippets.

The project implementation material is available at the following link: https://github.com/Mel-ania/Between-TimeTravelVideogame.git. Please refer to the v1.0 release.

3.1 - Object-oriented class system

Implementing a videogame requires using a programming language to encode the state and behavior of the various components. Using Unity, the recommended language is C#, but you can also program in JavaScript or other languages. For *Between* we followed the suggestion and chose to write in C# with the built-in integration for Visual Studio. «C# is a modern, object-oriented, and type-safe programming language. C# enables developers to build many types of secure and robust applications that run in the .NET ecosystem. [...] C# is an object-oriented, component-oriented programming language. C# provides language constructs to directly support these concepts, making C# a natural language in which to create and use software components.»[11]

Programming in an object-oriented language, it is fundamental to capture its purpose and peculiarities. Such as a functional language would be programmed with functions, in an object-oriented language, it is necessary to use objects. Each class has its own fields, methods and properties that can be inherited by its children or recalled and used by other objects. Thanks to *inheritance*, classes can and must be organized as a hierarchy, before assimilating, recalling, modifying and

[11] Microsoft Docs, C# docs.

adding new specialties to themselves. Two very powerful mechanisms, in this sense, are *overriding* and *overloading*:

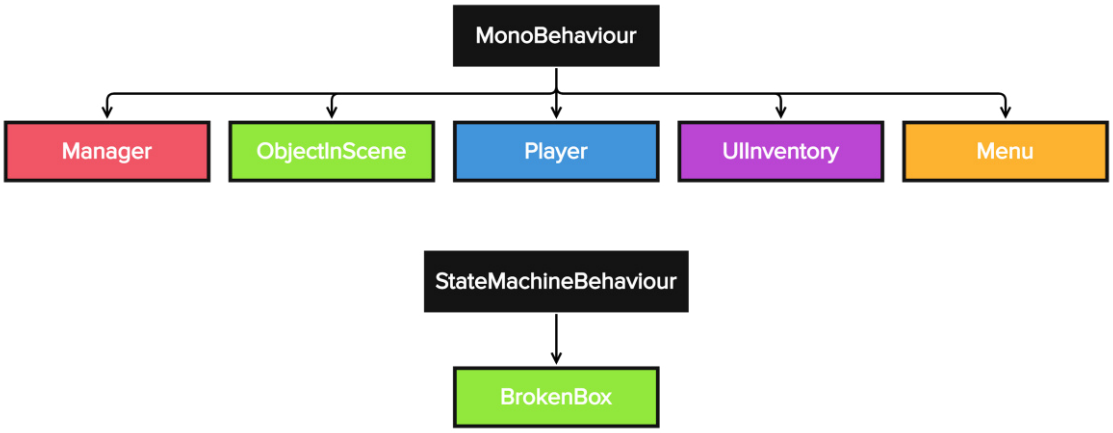
- *Overriding* is a type of polymorphism that allows you to modify the content of an inherited method. A child class will inherit all the parent class methods, therefore, it is often necessary specifying the behavior of certain methods. The method's name and signature have to be the same (except for the return type), in order to override it. If desired, in the new method the programmer can also call the superclass' one.

- *Overloading* is another type of polymorphism by which you can call multiple different methods with the same name. In order to do this, the methods' signature must be different in the type and/or in the number of parameters. It is not possible to exploit this strategy for methods that only differ in the return type. Overloading is useful for obtaining the same result from diverse parameters, without thinking about multiple names.

Since this paradigm, it is very natural for the game programmer to assign objects attributes comparable to their characteristics in the game. The character walks and jumps, a door is closed and a key can be collected and opens something closed. It is easy to understand why this system is worldwide diffused for creating videogames and more.

3.1.1 - Hierarchy Tree

We said that, working with an object-oriented language program, it is indispensable to know how to generate and employ a hierarchy tree. In this kind of tree, each class has only one parent class, but can have numerous children classes. The idea is organizing each component in relation to others with similar features. For example, "Fruit" class will have "FreshFruit" and "DriedFruit" classes as children; the "FreshFruit" class, for its part, will be parent of four classes corresponding to the seasonal fruit categories and each of them will have all their seasonal fruits as children. We shall now consider the structure of our video game's hierarchy tree at a high level and to what extent it prepares the subsequent work.



Graph 1. High level hierarchy tree of *Between*'s root classes

From Graph 1, we find out the direct children classes of *MonoBehaviour* and *StateMachineBehaviour*. *MonoBehaviour* is the base class that all scripts generated in Unity extend. It contains a multitude of functions that Unity automatically calls, some of which are:

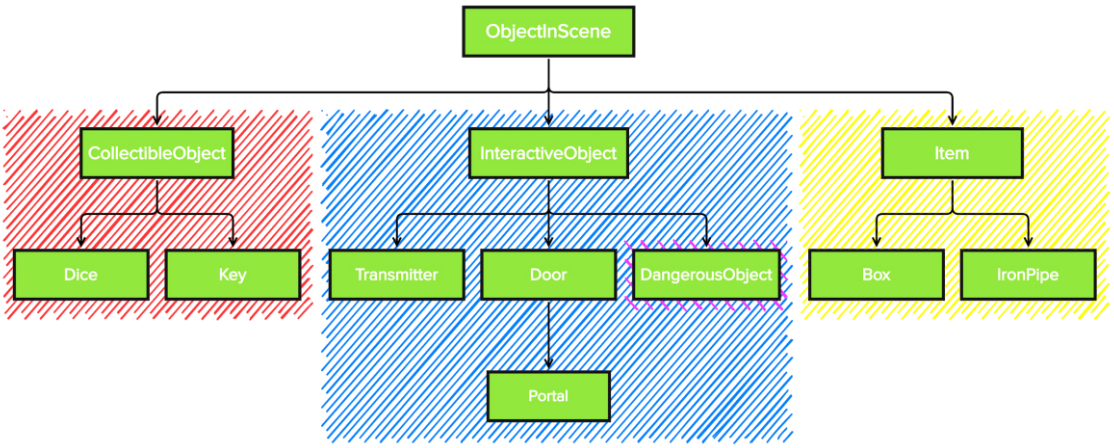
- Awake();
- Start();
- Update();
- FixedUpdate();
- LateUpdate();
- OnTriggerEnter();
- etc.

Each of these functions is part of the GameObject's life cycle, whose features its own script, and the scene's life cycle, in which the GameObject under consideration lies. Unity will call the *Awake()* function as soon as it loads the scene and immediately after it will call the *Start()* function, while cyclically it will call the *Update()* function. There are also two very similar functions to the last one, which operate in the same way, but at different times. Each processor has different performance from device to device, so Unity will call the Update() function at different intervals depending on the device. Instead, Unity will call the *Fixedupdate()* function with a fixed frequency regardless of processor. In addition, the *Lateupdate()* function, as the name suggests, will be called just after the Update() one. There are many other functions provided by the MonoBehaviour class and Unity documentation lists and details all of them[12].

[12] Unity Documentation, Scripting API: MonoBehaviour.

StateMachineBehaviour class, similar to the previous one, is a base class that classes which need to get their hands on the machine state extend. Normally, it is only used to handle certain behaviors during animations or transitions between them. Unity's official documentation also contains more information about this class[13]. Quickly observing the only class that extends the StateMachineBehaviour class, we can say that we have used it exclusively for a simple operation after an animation. This class has no children.

As regards to *Player* and *UIInventory* classes, they have no children with the current version of the game. *Player* class clearly encodes attributes and actions of the playable character (see Subsection 3.2.1), as well as the inventory status. Instead, the *UIInventory* class has the sole aim to manage the inventory representation in the UI. The two classes will be discussed and explored in Subsection 3.2.2.



Graph 2. High level *ObjectInScene* hierarchy subtree

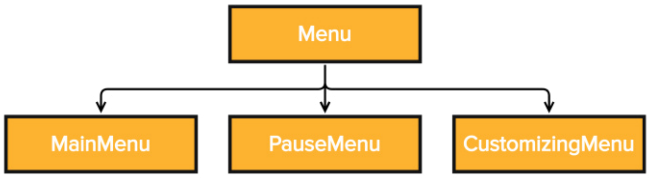
Now, moving on to talk about the subtree rooted in *ObjectInScene*. This class is a direct child of MonoBehaviour and contains all the scene objects. Graph 2 shows three branches involving its direct children: *CollectibleObject*, *InteractiveObject* and *Item*. These classes, respectively, represent the class of collectible objects, meaning the ones the player can take and store in his inventory; the class of objects which the player can interact with and the items present within the scene that have their own physics. This first subdivision is well highlighted in the graph and denotes a very marked view of classes:

- *CollectibleObject* class has two leaf children, as they are not specified further. We have identified no other useful or interesting subcategories.

[13] Unity Documentation, Scripting API: StateMachineBehaviour.

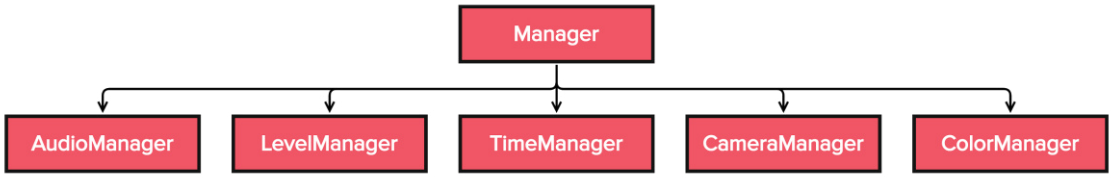
Therefore, we thought to proceed only with classes representing precise elements, such as *Dice* and *Key*.

- *InteractiveObject* class has three children: two specific interactive component classes (one of which is a superclass of a more specialized one) and a class that opens another whole category, *DangerousObject*. Even now this class has no children, as there was no need to differentiate different dangerous objects, if in the future we will desire various objects that inflict diversified damages to the player, the game programmer simply will create DangerousObject subclasses.
- *Item* class, once again, has only two leaf children that constitute specific objects. We do not deny that adding many new items, it would be useful to create subcategories by materials or sizes. In this version, the only two items are *Box* and *IronPipe*.



Graph 3. High level *Menu* hierarchy subtree

Using Graph 3, we examine another subtree rooted in one of the subclasses in Graph 1: *Menu*. Having different game menus to access, you had to organize them, like all the other elements. There are two major menus: the *MainMenu*, which is the "Home", and the *PauseMenu*, which appears when the game is paused. In addition, from the MainMenu, you can open a dedicated menu for customizing the character, a *CustomizingMenu*. We did not create the latter as a MainMenu's child, because, at code level, they have no common attributes, although logically they are within each other. These three menus have no children.



Graph 4. High level *Manager* hierarchy subtree

The last MonoBehaviour's child to consider is *Manager*, which is also the root of a subtree (see Graph 4). The Manager class collects all the game managers at various levels. Its children deal with both a more structural and a more conceptual

part. Particularly, *AudioManager*, *CameraManager* and *LevelManager* classes control the audio, the camera movements and the passage between different scenes. Meanwhile, the other classes, *TimeManager* and *ColorManager*, direct scene features, such as time travel and character's color.

3.2 - Player and inventory

If we have only dealt with the program at a high level so far, it is time we entered into the code. In order to draw up this thesis, only the most interesting code fragments will be shown, omitting banalities and exclusive references of graphics, animations or special effects.

As the initial argument, we want to deepen the character with which the player will enter into symbiosis. We want to make it clear how he acts and what he is allowed to do. In addition, we will also focus on the inventory, both from a concrete and visual point of view.

3.2.1 - Player

Our character must first be able to move and jump. Let us therefore start from this premise to begin showing how the program works.

```
private void FixedUpdate()
{
    moveInput = Input.GetAxis("Horizontal");
    rb.velocity = new Vector2(moveInput * speed, rb.velocity.y);

    if ((isFacingRight == false && moveInput > 0) ||
        (isFacingRight == true  && moveInput < 0))
    {
        Flip();
    }
    ...
}
```

Code Snippet 1. *Player* Class, *FixedUpdate()* Method, player's movement

In Code Snippet 1, we see the method *Fixedupdate()* reported, the same that we had introduced in Subsection 3.1.1. We chose this method instead of *Update()*, because it is important that you do not lose responsive performance and feedback from the character that you command. In this extrapolation, we save the horizontal

axis input[14] and we calculate the speed at which the character 's rigid body will move, creating a Vector2. After that, we check if the character is facing the correct direction while there is an input. In case the character is facing right with left input or vice versa, we call the *Flip()* method, which will turn the character, scaling the entire GameObject.

```
private void Update()
{
    if (isGrounded)
    {
        extraJumps = extraJumpsValue;
    }

    if (Input.GetKeyDown(KeyCode.Space) && extraJumps > 0)
    {
        rb.velocity = Vector2.up * jumpForce;
        extraJumps--;
    }
    else if (Input.GetKeyDown(KeyCode.Space) && extraJumps == 0
                                             && isGrounded)
    {
        rb.velocity = Vector2.up * jumpForce;
    }
    ...
}
```

Code Snippet 2. *Player* Class, *Update()* Method, player's jumps

In the case of Code Snippet 2, to manage the jumps, we opted for the *Update()* method. This is because within the *Fixedupdate()* method we update a checker (placed at character's feet) by writing if he is or is not touching the ground. Here, the "isGrounded" checker is used to determine if it is possible to reset the extra jumps counter. Subsequently, if the player presses the space input and the number of jumps is greater than zero, they are decremented and the character will jump. If the player presses space and there are no extra jumps, while the player is on the ground, he will jump, otherwise he will not.

This method also handles time travel input, but we will specify it in Section 4.1.

[14] Unity provides an automatic assumption for certain motion input, which the user can customize. For the horizontal axis, the input value would only be in [-1, +1] range. 0 represents neutral input, -1 is left input and +1 is right input. The accuracy of this value depends on the device: from a keyboard, the input will always be only -1, 0 or 1; instead, from a joystick, it will cover every slight change, allowing float numbers.

Within the *OnTriggerEnter2D(Collider2D other)* method in the *Player* Class, the interaction between the character and *other* when they collide with each other is encoded. In order for the collision to occur, both the character and the potential collided object must possess a *Collider* among their components. Moreover, inside the Collider there is a "Is Trigger" checkbox that must be checked only in the potential collided object. With this in mind, *other* is compared to a multitude of tags[15], in an attempt to return the right behavior. Now, we are going to examine some examples.

```
if (other.CompareTag("Transmitter"))
{
    Transmitter transmitter = other.GetComponent<Transmitter>();
    if (transmitter.IsActive)
    {
        if (ContainsKeyType(Key.KeyType.Red) && isGrounded)
        {
            ColorListRedToGreen();
        }
    }
}
```

Code Snippet 3. *Player* Class, *OnTriggerEnter2D(Collider2D other)* Method, transmitter

In Code Snippet 3, *other* is recognized as "Transmitter". At this point, we can extract the "Transmitter" component and check if it is active. If so, then we check if the player is on the ground and if there are keys with KeyType red (see Subsection 3.3.1) in his inventory by the *ContainsKeyType(Key.Keytype keytype)* method. If the answer is yes again, we call the *ColorListRedToGreen()* method, which turns all the red keys in the inventory into green keys.

```
if (other.CompareTag("Portal"))
{
    if (ContainsKeyType(Key.KeyType.Green) && time.IsPresent)
    {
        Portal portal = other.GetComponent<Portal>();
        RemoveOneKey(Key.KeyType.Green);
        portal.OpenPassage();
    }
}
```

Code Snippet 4. *Player* Class, *OnTriggerEnter2D(Collider2D other)* Method, portal

[15] A *tag* is a label that the developer attaches to a GameObjects to identify it within a category.

In Code Snippet 4, *other* is recognized as "Portal". This if branch is very similar to the case where *other* is "Door", but with some small changes. In this instance, we check if there are green keys available in the inventory with the *ContainsKeyType(Key.Keytype keytype)* method and verify that the time is present through the *TimeManager* "time". If yes, we get the "Portal" component and use it to open it thanks to its *OpenPassage()* method, while we remove a green key from the inventory with *RemoveOneKey(Key.Keytype keytype)*. Otherwise, we do nothing.

```
if (other.CompareTag("Enemy"))
{
    pm.DeadPlayer();
    Destroy(gameObject);
}
```

Code Snippet 5. *Player* Class, *OnTriggerEnter2D(Collider2D other)* Method, enemy

In Code Snippet 5, *other* is recognized as "Enemy". This case means that the player is dead, therefore we recall the *PauseMenu* and its method *DeadPlayer()*, deepened in Section 3.4. Finally, we destroy the character's GameObject, making it no longer playable.

3.2.2 - Inventory

Moving on to analyze the inventory management more specifically. It contains both keys and dice, but the latter are managed only by a counter. Hence, we are just going to talk about the key management.

```
if (other.CompareTag("Key"))
{
    Key key = other.GetComponent<Key>();
    AddKey(key);
    Destroy(other.gameObject);
    OnInventoryChanged?.Invoke(this, EventArgs.Empty);
}
```

Code Snippet 6. *Player* Class, *OnTriggerEnter2D(Collider2D other)* Method, key

As you see in Code Snippet 6, always within the same method, *other* is recognized as "Key". Being a key, and so a collectible object, the player will have to collect it and place it in his inventory. First, we extract the "Key" component, next we call the *AddKey(Key key)* method which adds the key to the real inventory. After storing it, we destroy its GameObject to remove it from the scene and we indicate to the UI that something has changed in the inventory.

```
List<Key> keyList = player.KeyList;
int i = 0;
for (i = 0; i < keyList.Count; i++)
{
    Key key = keyList[i];
    Key.KeyType keyType = key.IsKeyType;
    Transform keyTransform = Instantiate(keyTemplate, container);
    keyTransform.gameObject.SetActive(true);
    keyTransform.GetComponent<RectTransform>().anchoredPosition =
            new Vector2(-50 * i, 0);
    Image keyImage = keyTransform.Find("Base").GetComponent<Image>();
    switch (keyType) {...}
}
```

Code Snippet 7. *UIInventory* Class, *UpdateVisual()* Method, set key inventory

Changing the inventory composition, the UI does not update automatically, in fact, it is necessary to invoke a notification, as we have just said. When the *UIInventory* class receives the signal, first, it deletes the old key inventory. Immediately after, as Code Snippet 7 shows, it recreates the key list and for each of them it builds its graphics. Now, we can look at one cycle: we extract a key and its KeyType, then we instantiate a "Transform" component from a template, activate it and place it next to the previous key. Finally, we target and assign the right color to the key through a switch on its KeyType.

3.3 - Objects in scene

After the main character, we proceed with all the other scene objects, studying their most interesting aspects. In the structure of this Section, we will reference the *ObjectInScene* subtree hierarchy (see Graph 2).

3.3.1 - Collectible objects

A CollectibleObject does not stand out for its actions or performances, but rather for the fact that they are collectible and can generate a particular interaction. Talking of keys, the way the player uses them, from one side, is obvious, but from the other it relates to time travel, which is unusual. However, this aspect is more related to owning the keys than to the keys themselves. We will better study time travel in Section 4.1.

```
private KeyType keyType;

public enum KeyType
{
    Red, Green, Blue, ...
}

public KeyType IsKeyType
{
    get { return keyType; }
    set { keyType = value; }
}
```

Code Snippet 8. *Key* Class, defining *KeyType* type

Concerning the key itself, we want to focus on its *KeyType* (see Code Snippet 8), which we have already mentioned several times. This type specially built for keys is actually an *enum*, or an enumeration of things that form that type. The *KeyType* contains all possible key types. Each key has a default *keytype* and it may change, depending on whether it is or is not a key to be activated. In order not to directly allow another class to access the *keytype* field, we used a property. «A property is a member that provides a flexible mechanism to read, write, or compute the value of a private field. Properties can be used as if they are public data members, but they are actually special methods called *accessors*. This enables data to be accessed easily and still helps promote the safety and flexibility of methods.»[16] This way, we have preserved the private access of the field and ensured its safety.

3.3.2 - Interactive objects

InteractiveObject are those the player has an exchange with. They are animated objects that change their status and act preventing or allowing the player to perform certain actions. Everyone of them has its own methods that the *Player* class can invoke.

```
public new void OpenPassage()
{
    if (time.IsPresent)
    {
        portal.gameObject.SetActive(true);
    }
}
```

Code Snippet 9. *Portal* Class, *OpenPassage()* Method

[16] Microsoft Docs, Properties - C#.

Observing the *Portal* class, there is one method we saw the Player class invoked in Code Snippet 4: *OpenPassage()*. From Code Snippet 9, we can notice that this method overrides (thanks to the "new" keyword) the homonymous method that the *Portal* class has inherited from the *Door* class. The method checks by the *TimeManager* that the time is present. If so, we set the portal's GameObject active, thus opening the passage.

```
public void TurnOnOff()
{
    if (isTurning)
    {
        off.SetActive(isActive);
        isActive = !isActive;
        on.SetActive(isActive);
    }
}
```

Code Snippet 10. *DangerousObject* Class, *TurnOnOff()* Method

Changing class, we can also study an example from the class *DangerousObject*, beginning from Code Snippet 10. A dangerous object can vary status (on or off) depending on the time set. The *TurnOnOff()* method takes care of that. First, we check if the object in question can change its status. If so, we swap the status of its on and off version through the value of the "isActive" boolean. If it was on, its on version is hidden and its off version is shown, if it was off, the opposite happens.

3.3.3 - Items

With regard to *Item* class, we have previously explained how items were attributed to physical characteristics from the real world. With this assumption, we have assigned most of these attributes directly in the Inspector (see Figure 3). However, for custom behaviors, it was necessary to create a special encoding.

```
private void LateUpdate()
{
    if (!isGrounded && isLanded)
    {
        isLanded = false;
        yPosition = transform.position.y;
    }
    else if (isGrounded && !isLanded)
    {
        isLanded = true;
        if (yPosition - transform.position.y > maxHeight)
        {
            animator.SetTrigger("broken");
        }
    }
}
```

Code Snippet 11. *Box* Class, *LateUpdate()* Method

With the *Box* class, we had established there would be a maximum height from which a box could fall without breaking. Code Snippet 11 highlights how the *LateUpdate()* method of the *Box* class ensures that. In the *Fixedupdate()* method, we wrote a check that records if the object is on the ground. With a first check, we set, if the object is not on the ground, but has landed, that it has not landed and reset its altitude. If it does not get into that if branch, we check if it is on the ground and it has not landed: in case, we confirm that it is landed and we calculate the height distance it traveled. If it is too high, we then tell the Animator it has to start the broken box animation.

At this juncture, it was decided to open a small parenthesis on animation to explain the *BrokenBox* class role. Once the animation started, it was necessary to deactivate his GameObject, considering the unusable box. To do this, we created a special class, which would do it at the end of the animation.

3.4 - Menus

Briefly entering into the world of menus, they serve the player to move through various interfaces and to make him feel not abandoned in the game.

```
public void NewGame()
{
    PlayerPrefs.SetInt("SavedLevel", 1);
    PlayerPrefs.SetInt("Dices", 0);
    SceneManager.LoadScene("Level 1");
}
```

Code Snippet 12. *MainMenu* Class, *NewGame()* Method

From the "Home" or *MainMenu* (see Figure 15), you can do two primary actions and two secondary actions. The two main and most important actions are starting a new game and continuing the previous game. From Code Snippet 12, we see one of the two principal possibilities. In the *NewGame()* method we want the player to start from the beginning, so we use the *PlayerPrefs* to set level 1 as a checkpoint and the dice number to 0. If there was a score to reset, you should do it here. Ultimately, charging a new game starting with the first level scene.

```
public void Continue()
{
    SceneManager.LoadScene("Level " + PlayerPrefs.GetInt("SavedLevel"));
}
```

Code Snippet 13. *MainMenu* Class, *Continue()* Method

In parallel, the *Continue()* method (see Code Snippet 13) simply loads the last saved level. This means that through the *SceneManager* (provided by Unity), we load the scene with a name corresponding to "Level" plus the level number we extract from the *PlayerPrefs*. *PlayerPrefs* are small information that the programmer can save; in fact, they locally save data such as scores or checkpoints.

```
public void OnPause(bool isPaused)
{
    if (isPaused)
    {
        Time.timeScale = 0f;
    }
    else
    {
        Time.timeScale = 1f;
    }
}
```

Code Snippet 14. *PauseMenu* Class, *OnPause(bool isPaused)* Method

Similarly, from the *PauseMenu* (see Figure 16), you can perform two main and two secondary actions. The peculiarity of this specific menu is that when it is activated,

the background game has to stop. Alternatively, the player may accidentally press keys and move the character without knowing, possibly dying. To freeze time, we wrote the *OnPause(bool isPaused)* method (see Code Snippet 14). Simply, we check if the game is paused when we invoke the method: if yes, we scale the game time to zero, i.e. freezing the game; otherwise, we scale the game time to one, i.e. unfreezing the game and resetting it at normal speed.

```
public void DeadPlayer()
{
    gameObject.SetActive(true);
    OnPause(true);
    continue.gameObject.SetActive(false);
}
```

Code Snippet 15. *PauseMenu* Class, *DeadPlayer()* Method

In Subsection 3.2.1, we noted that, when the player dies due to an enemy, we call the *DeadPlayer()* method (see Code Snippet 15) of the *PauseMenu* class. This method activates the menu and tells its *OnPause(bool isPaused)* method that time has to stop. Then, it hides the possibility to resume the level and forces the player to either restart the level, or just go back to the Home.

3.5 - Managers

Now, we move to the *Manager* hierarchy subtree to consider a final couple of intriguing aspects, which might not be considered by a beginner, but are really useful.

```
private void Awake()
{
    DontDestroyOnLoad(transform.gameObject);
    ...
}
```

Code Snippet 16. *AudioManager* Class, *Awake()* Method, preserving background music

First, the *DontDestroyOnLoad(Object target)* method that the *Awake()* method of the *AudioManager* class invoked in Code Snippet 16. This single method allows a GameObject not to be destroyed when changing scene, which is normal. It is very useful, especially to keep a continuous background music within multiple scenes.

```
private void LateUpdate()
{
    if (player)
    {
        Vector3 temp = transform.position;
        temp.x = player.position.x;
        if (temp.x > min && temp.x < max)
        {
            transform.position = temp;
        }
    }
}
```

Code Snippet 17. *CameraManager* Class, *LateUpdate()* Method

As the last method we are going to analyze, we take a look at Code Snippet 17, where we find the method *LateUpdate()* of the *CameraManager()* class. In it, we encoded the camera to follow the character. Please note that, in Unity, to identify a GameObject's location, it uses a vector where the coordinates are stored and updated. So, we make sure the player is not dead. With this condition verified, we create a temporary vector where saving the camera location. We then replace the temporary vector x-coordinate with the character's position vector x-coordinate. Next, we check this coordinate does not exceed the level limits and, consequently, we update the camera's position with the temporary vector. By this way, the camera will always follow the character (within the level limits) on the horizontal axis, but never on the vertical one.

As a result of code fragments we just saw more the remaining code not shown here, the videogame appears very solid and well-functioning. Every line of the program, from the simplest to the most complex, is necessary to make it a bearing structure that supports design, story, gameplay, graphics and everything else. Thus, after understanding its mechanism, we can jump into the project's highlight and study how our player can travel through time.

Chapter 4
Time Travelling

This fourth Chapter discusses time travel, this videogame's soul. First of all, we will explain in detail how time travel works in the space-time continuum of *Between*. Next, we will conclude the code analysis with the last Code Snippets regarding time travels and space-time paradoxes. Finally, we will talk about how we have materially created levels on multiple timelines and how the player can find them compelling as the game progresses.

4.1 - How time travel works

In the world specifically created for this videogame, time travel exists. In every fictional world where time travel happens, the authors interpret it in a myriad ways. In some scenarios, the characters live in a single timeline and everything is already predetermined. In others, even if the characters perform different actions, they always arrive at the same conclusion, because destiny can not be changed. Another way of thinking is travel on parallel timelines. Other ones can change their timeline, or even the whole world one, traveling at any time in the past and in the future, erasing historical memories from people's minds.

For *Between*, we chose to allow our character to travel between two specific time points, hence the eponymous name of the game. The player can switch only between these two moments: he can not decide what era he will end in and he can not go into a future he has not yet lived. These two time points correspond to the present time moment which the character belongs to and to a specific past moment occurred much before the character's birth. When he comes back to the present, he returns to the precise moment when he has gone back in time. While, when he travels to the past, he will always return to the exact same moment every time (see Figure 18).

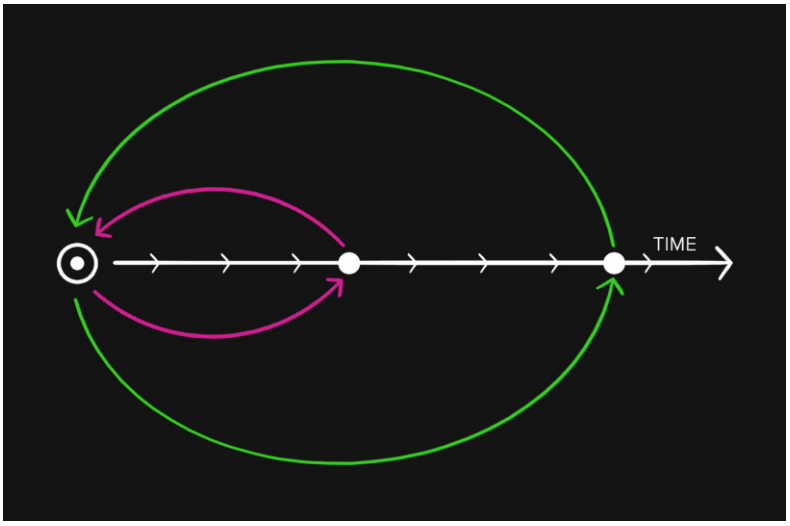


Figure 18. Time travel in *Between*

The fixed point in the past is an interchange, from which potentially infinite timelines with alternate present depart. During the journey, the player will need to switch time to overcome certain obstacles, otherwise insurmountable. However, whenever the player goes back in time to change the scene in front of him, it will generate a space-time paradox.

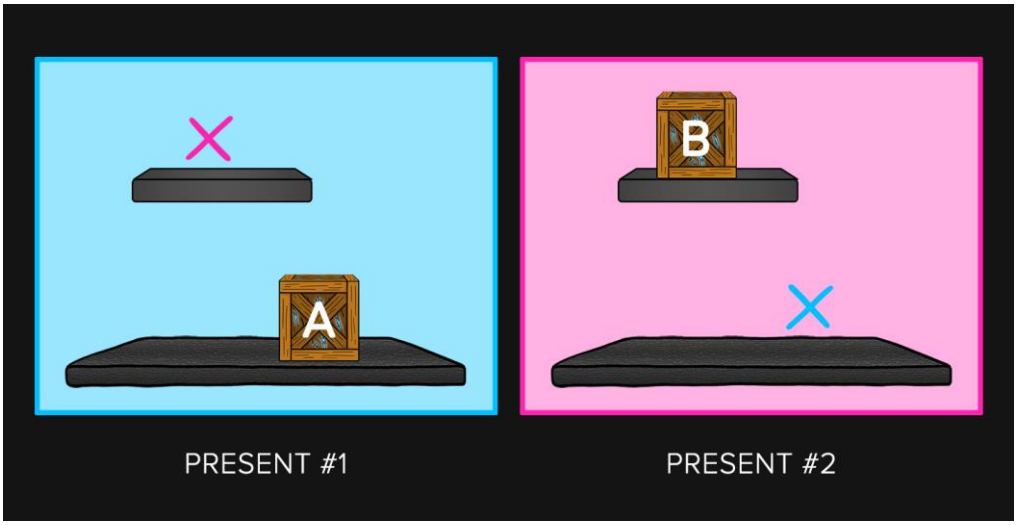


Figure 19. Space-time paradox

How do space-time paradoxes work? The concept is simple: if a box is at point A in present #1, but it is at point B in present #2, it creates a paradox (see Figure 19). Reality does not accept paradoxes, so it splits the two presents. When the player goes back in time because he needs to change the position of some objects, he creates a paradox. Therefore, reality will create a new timeline with an alternate present (see Figure 20). Note that the player can no longer go back to the first

present, unless he repositions all objects in the same place, but this would stop him from advancing. However, there is a small shrewdness, whose allows the player to move between the past and the specific present from which he comes. If in the past he does not move anything, then he has not altered the space-time continuum, being able to return to the present without repercussions. *CollectibleObjects* are atemporal, so they do not belong to any specific timeline.

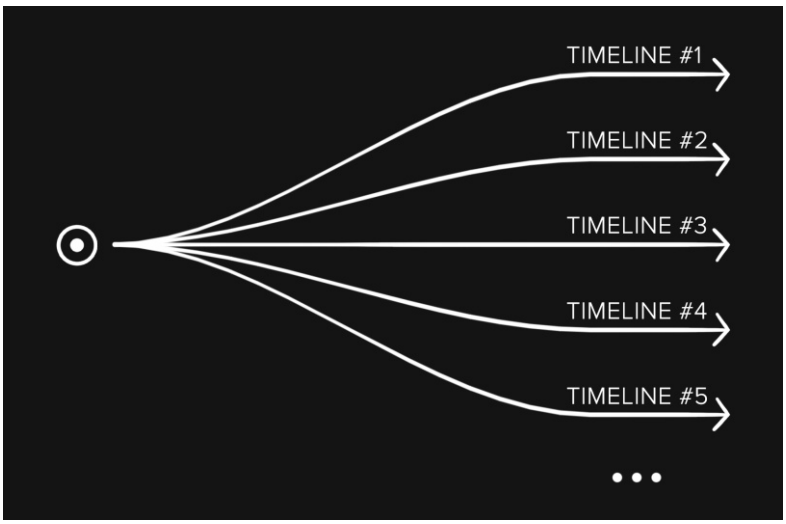


Figure 20. Time splitting into multiple timelines

4.2 - Exploring time travel techniques

We thus end the explanation of time travel at a high level, to give space to the code that bears them. This Section explores the most interesting code aspects we began to see in Sections 3.2 - 3.3 - 3.4 - 3.5. Following the testing of some techniques, we have reached the current version we propose in this Section.

4.2.1 - Different parallel timelines

In Section 4.1 we set out how time travel works in the world of our video game. So, we talked about two exact time points between which the player can travel. Now, we will see how we managed these time travels at a code level and how it allows multiple parallel timelines.

```
private void Update()
{
    ...
    if (ContainsAtLeastOne() && Input.GetKeyUp(KeyCode.T))
    {
        time.ChangeTime();
    }
}
```

Code Snippet 18. *Player* Class, *Update()* Method, changing time

The first thing to know, especially for the player, is how he can practically activate time travel. From Code Snippet 18, we see the *Player* class's *Update()* method. Here we check if the player is providing the "T" input and if the inventory has at least one key through the *ContainsAtLeastOne()* method. If so, we use the *TimeManager* to call his *ChangeTime()* method (see Code Snippet 19). By this way, it will automatically calculate in which time direction the player will travel and bring him to it.

Knowing what to do to move back and forth in time, we are actually going to see how time travel works within the *TimeManager* class.

```
public void ChangeTime()
{
    isPresent = !isPresent;
    foreach (Item item in itemList)
    {
        if (!isPresent)
        {
            item.gameObject.SetActive(true);
        }
        item.ResetPosition();
    }
    foreach (DangerousObject danger in dangerList){
        danger.TurnOnOff();
    }
}
```

Code Snippet 19. *TimeManager* Class, *ChangeTime()* Method

Looking at the Code Snippet 19, we examine the content of the *ChangeTime()* method of the *TimeManager* class. The first thing we do is toggling the "isPresent" boolean value to change the time set according to the current time. Next, we take the entire item list and for each of them, if the player is traveling to the past, we activate his GameObject. Anyways, we invoke the *ResetPosition()* method (see Subsection 4.2.2) of each one of them to set their right position. Concerning

DangerousObject, we similarly take their whole list and for each object we call its method *TurnOnOff()* (see Subsection 3.3.2).

4.2.2 - The mechanics of space-time paradoxes

The last important class we have to consider in order to insert the last piece of the puzzle is the *Item* class. This class is full of essential statements which handle space-time paradoxes. An *item* is a static object, thereby it should not perform actions, nonetheless within each item a lot of information flows.

```
public struct ItemState
{
    public Vector2 Position;
    public Quaternion Rotation;

    public ItemState(Vector2 position, Quaternion rotation)
    {
        Position = position;
        Rotation = rotation;
    }

    public void ChangeState(Vector2 newPosition, Quaternion newRotation)
    {
        Position = newPosition;
        Rotation = newRotation;
    }
}

private ItemState currentState;
private ItemState pastState;
private ItemState presentState;
```

Code Snippet 20. *Item* Class, *ItemState* Struct

The initial *Item* class block is a *struct* called *ItemState* (see Code Snippet 20). It will help storing different states of the items themselves. «Structs are similar to classes in that they represent data structures that can contain data members and function members. [...] They can be conveniently implemented using value semantics where assignment copies the value instead of the reference.»[17]

Inside the struct we defined two fields for position and rotation, as well as a constructor and the *ChangeState(Vector2 newPosition, Quaternion newRotation)*. From this simple struct, we generate three fields with *ItemState* type:

[17] Microsoft Docs, Structs - C# language specification.

"currentState", "pastState" and "presentState". Here the meaning of these three *ItemState*:

- "pastState" stores item's position and rotation of the past moment the player reaches traveling back in time;
- "presentState" stores item's position and rotation of the present moment in which the player leaves to go back in time;
- "currentState" stores the actual item's position and rotation at the current play moment, regardless of whether the character is in the past or in the present.

```
private void Update()
{
    currentState.ChangeState(transform.position, transform.rotation);
    if (!isPresent.IsPresent)
    {
        if(StateDifference(pastState, currentState) > 0.1)
        {
            presentState = currentState;
        }
    }
}
```

Code Snippet 21. *Item* Class, *Update()* Method

Going down, we find the method *Update()* (see Code Snippet 21). Here, whenever Unity calls this method, we update the "currentState" with the updated item's position and rotation. Then, if the player is in the past and there is a difference in position or rotation between "pastState" and "currentState", we update the "presentState" to the "currentState". Here "currentState" symbolizes the state in which the item is located in the past, but at a time subsequent to the arrival, even just a few seconds later. If there is a difference, it means that the player has moved the item in the past. So, we have to create a new timeline with an alternate present, where that item takes the new position.

```
private float StateDifference(ItemState is1, ItemState is2)
{
    float xPosition, yPosition, xRotation, yRotation;
    float diffPosition, diffRotation;

    xPosition = Math.Abs(is1.Position.x - is2.Position.x);
    yPosition = Math.Abs(is1.Position.y - is2.Position.y);
    xRotation = Math.Abs(is1.Rotation.x - is2.Rotation.x);
    yRotation = Math.Abs(is1.Rotation.y - is2.Rotation.y);

    diffPosition = xPosition + yPosition;
    diffRotation = xRotation + yRotation;

    return (diffPosition > diffRotation)? diffPosition : diffRotation;
}
```

Code Snippet 22. *Item* Class, *StateDifference(ItemState is1, ItemState is2)* Method

StateDifference(Itemstate is1, Itemstate is2) method, as in Code Snippet 22, calculates the difference in position and rotation between two *ItemStates* through their coordinates. The method's signature requires two *ItemState* from which to extract x- and y-coordinates for position and rotation. To find any difference, we use the *Abs(float value)* method from the *Math library* which returns an absolute value. Once we get all the differences among coordinate pairs, we sum the two for the position and the two for the rotation. Ultimately, we use a ternary operator to return the major difference between the two of them.

```
public void ResetPosition()
{
    if (isPresent.IsPresent)
    {
        currentState = presentState;
    }
    else
    {
        presentState = currentState;
        currentState = pastState;
    }
    transform.position = currentState.Position;
    transform.rotation = currentState.Rotation;
}
```

Code Snippet 23. *Item* Class, *ResetPosition()* Method

The last code fragment we analyze is Code Snippet 23. From the Code Snippet 19, we have seen the usage of the *ResetPosition()* method, that the *TimeManager*

invokes when the time set changes. This method is the key to dealing with space-time paradoxes. So far, we have figured out how to store the right item state information, but we have never used it. In *ResetPosition()*, if the player is in the present, the "currentState" becomes the "presentState". When he goes back to the past, the “presentState” assumes the last item state value of the present. Changing then the “currentState” to the original “pastState”, it is actually resetting its location. When we update the placeholders, we apply the position and rotation transformation directly to the object.

4.3 - Level design process

In Subsection 2.3.2, we gave a first explanation of what a level designer does. Now, we dedicate this Section to the level design process that was used in the case of our videogame. Due to multiple timelines, the level designer could not be linear, contrarily, he had to adapt.

Level designers often use placeholders and prototypes in game versions without final graphics. Previously, level designers used to do both visual arts and levels’ structure, but now studios make more and more attempts to divide the two areas looking for specialists.

4.3.1 - Level design in multiple timelines

When the player faces a level, he knows he will have to manipulate objects as he time travels. He will assume the level is specifically designed to allow him to get to the end and that he will probably have to change the scene to do it. A level designer starts with this thought in mind to conceive amusing levels.

Among the steps to design a complete level, there are concept arts and sketches. Once the level designer completes these steps, he moves on to modeling the scene, placing the various objects in it, often with the help of a level editor (see Subsection 4.3.2). This process could require several revisions and each version could be set aside, for starting the work all over again looking for a better result.

In our case, he had to take into consideration the basic scene elements (see Subsection 2.3.1) and the player’s goal: getting to the exit portal. The level always has two portals, one at the start and one at the end, and in between all the rest. Disposing everything else, we had to find a way to avoid obvious paths and build a path working on multiple timelines.

The approach embraces the following concept: an object can help or hinder you, but in different time sets it can serve different purposes. Similarly, an item may need to stay in a certain position to keep a passage open, so when traveling to the past, you must be careful not to move it. Some objects are active only in specific time periods, such as the transmitter, while some other objects may be broken in the present. Moreover, at the beginning of the level, you can not time travel until you obtain a key. Therefore, the level has to be structured in such a way that allows the passage somehow and somewhen.

The first step in the process was devising small brain teasers, assuming the player could only overcome them by time traveling. For example, if there is an electric sphere on the path to the transmitter, the player has to pass it by jumping. Whereas, if it is an electric beam too long to be surpassed, he must go back in time to when the electric beam was off. We did not do this step in Unity, because composing a scene in its editor takes a long time. Designing a level by constantly moving and adjusting the pieces, it was much easier to sketch it on paper.

After that, we tried to combine some of those little puzzles to get a more framed composition which was more difficult to visualize at first impact. Once we had some interesting enigmas, we had to create real levels, connecting several puzzles. It was necessary understanding where to insert essential level elements and how to push the player to undertake certain paths rather than others. Only at the end of this process, we transposed everything into Unity through its level editor.

To make these concepts clearer, we are going to explicate three examples, namely three whole levels. We will talk about the right paths and the level variations due to time travel and alternate presents.

Figure 21. *Between*, Level A

From Figure 21, we entirely observe level A. The player will not have this equally wide view, in fact, his screen will contain about half of it. As you can see, there are

entry and exit portals, the character in front of the entry portal, the transmitter and a key near the exit portal. Keep in mind that the entry portal will disappear immediately, while the exit one will not be active. Also observe the transmitter is broken, because we are in the present. The player's goal will be arriving at the exit portal to pick up the key, going to the transmitter in the past to activate it and going back to the portal to turn it on and get out.

There are four brain teasers in this level:

- The first one is very easy, it does not require time travelling or moving scene objects. You must avoid the electric beam and go further, so you just have to jump on the box between the two electric spheres. To cross it, all you have to do is double-jump to the second box to avoid injury.
- Even the second does not exige time travel. In fact, you do not yet have the key that gives you the ability to do it, so it is done on purpose. When you get on the first platform, if you try to reach the second one without moving anything, you will not succeed. All you have to do is fall on iron pipes and move some of them to jump off.
- Once you have taken the key, you can finally time travel. This third enigma's logic is very similar to the second one. To get back on the first platform, you will have to move iron pipes again, otherwise you will not be able to reach it. However, you may not be able to move them if they all fall down covering the entire floor. In this case, you can go back in time and jump on them.
- The final puzzle requires a little more attention. Once back on the first platform, you will have not to drop the box. If you drop it, you will just go in the past to get it back. By moving the box to the platform's edge, you will be able to use it as a trampoline to double jump. To get on the top platform, where the transmitter is, you have two options: either you are very precise with jump landing under the electric sphere, or you simply go back in time when it was off.

After that, the level resolution is trivial. You will activate the key, while fronting the working transmitter in the past, you then will retrace the same path as before to the exit portal.

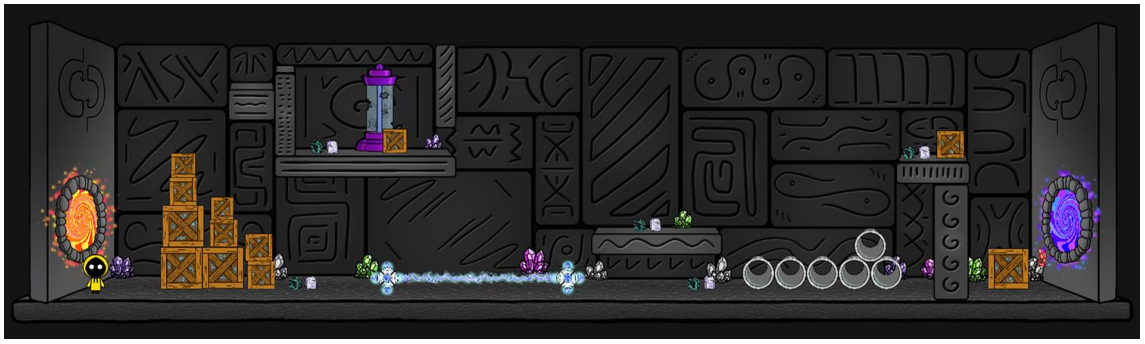


Figure 22. *Between*, Level B

From Figure 22, we entirely observe level B. As before, the player will not have this equally wide view, he will see about half of it. Again, there are the two portals, the character, the transmitter and a key (near the exit portal). The player's goal will be arriving at the exit portal to pick up the key, going to the transmitter in the past to activate it and going back to the portal to turn it on and get out.

For this level, we have prepared five brain teasers:

- Once again, the very first puzzle is easy. You will have to push the pile of boxes, first to pass and then to create a bridge which you walk on to avoid the electric beam. Note that some boxes are too small to protect you, so he will have to be careful about pushing the bigger boxes onto the beam.

- From the first platform, all you have to do nothing but jump on the iron pipes to access the second platform. Here we find the first dead end of the game, which forces the player to start again in case he makes a mistake. When you get off the platform to pick up the key, you will notice you can not jump high enough to go back on it again. So, you will have to start over. On the platform there is a box you have to drop to then be able to climb it. There are two other problems at this point: the first is that you must be careful not to drop the small box beyond the bigger one. This would be equivalent to the previous situation without an extra box. The second is more difficult to happen, but it is still plausible. Once you take the key, you must be cautious not to go back in time. If you accidentally move the bigger box, you would create an alternate present where you could no longer exploit the extra box.

- When he finally manages to climb back on the second platform, you will still have to return to the start. To do it, you can take advantage of the boxes you have already moved onto the electric beam. However, if the bigger

boxes are too far to reach them, you will have to push the iron pipes under and beyond the first platform.

- At the starting point, you need to reach the small elevated room, where the transmitter is located. You will be forced to go back in time to rebuild the box structure, because the room's access is too high. By jumping on boxes, you will also have to be careful not to throw them down for the same reason.
- After activating the key with the working transmitter, you must go back to the exit portal to activate it and enter into it. There is a hole in the wall inside the room that invites you to push that little box until it drops out. Although, that is a trap, because doing so, the box will fall and break, so you will no longer be able to leave the room. Even if it is small, the walls are high and to get out you need that little box. Luckily, you have the key to time travel, reset the location of the box and use it to your advantage.

Once that is done, you will reach the ending portal again, activate it and enter into it.

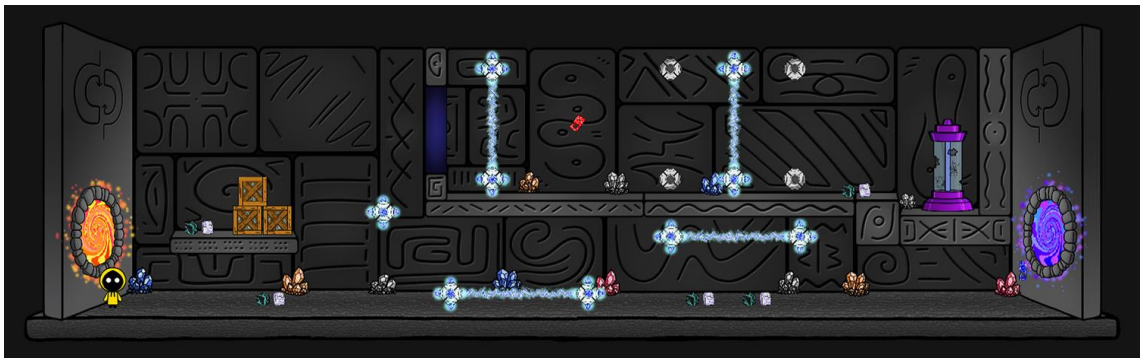


Figure 23. *Between*, Level C

From Figure 23, we entirely observe level C. Again, the player will not have this equally wide view, he will see about half of it. Newly, there are the two portals, the character and the transmitter, but there are two keys. The key near the exit portal is blue and corresponds to the blue door, while the portal key is in the room locked by the blue door. The player's path will be arriving at the exit portal to take the blue key, opening the blue door, taking the portal key, going to the transmitter in the past to activate it and going back to the portal to turn it on and get out.

In this level, the brain teasers are four:

- This time, the first puzzle is a bit more difficult, especially from a practical point of view. Like before, you must create a passage to cross the electrical beam under the locked room. You have to jump on the first platform and

push the three boxes down. Next, you have to push them again to the center of the electric beam. Here is a little catch: if you push the boxes slowly, these probably will not get far enough to allow you to pass. In fact, he will have to benefit from the acceleration to push further than where they might normally have come. The elevated room's floor is very low, so much so that it allows only small jumps. Indeed, you will have to be prepared and precise in jumping.

- Taking the blue key, it will be very difficult for you to jump back on the boxes bridge. Actually, all you have to do is go back in time to when the first beam's electric spheres were deactivated to be able to go over them. To open the blue door, you just need to touch it. Getting there means you want the boxes back on the first platform: you will go back in time, move them closer to the door and jump from there.

- When you open the door, you will have used the blue key, so it will disappear from the inventory. It will no longer be available for time travelling, so you will be stuck in the time you are in until you find another one. If you were not careful, it could be a big problem. Right after the door, there is an electric beam that blocks the passage. It is active in the present, but turned off in the past. Before opening the door, if you were not in the past, you can not proceed and you must start over or die.

- Arriving at the second key inside the room, the last puzzle is missing. Three vertical electric beams are active alternately in past and present. You will have to continuously travel back and forth between present and past and dodge them, in order to cross them. Logically, this puzzle is not very difficult, but it can take some practice.

Once reached the transmitter, you will only have to activate the key, exit the room, activate the exit portal and finally get out.

This level design process can not be automated or generated automatically by a computer. The human mind is vital to structure these kinds of levels. A computer will not be able to predict the immense amount of variables involved. It could not handle all the possible changes of space-time paradoxes. Alongside, objects that at some times are active and at others not, or broken objects that at some times are fixed, etc. Not to mention the playability and probably not very exciting levels automatically generated, which are not meant to be it.

4.3.2 - Making the challenge

Level design is crucial for two main reasons: creating a goal-looking path and ensuring the player an amusing gaming experience. Good designs produce quality games with an immersive experience both in the game and story by challenging the player. Challenge is essential and gives value to a game. Without it, a game is not pleasant and the player will abandon it soon.

There are several ways to create challenges. Two of the main ones are manual difficulty and logical difficulty. In both fields, the difficulty increases by rapidity and accuracy. The developer can push the player to readiness and preciseness in performances and game interaction. On the other hand, he can focus on problem-solving, concretizing his thinking both in speed and meticulousness. Between the two enunciated possibilities, we have chosen the intellectual one, because we consider it more pertinent to the background game concept.

Immersion is a direct consequence of how the player perceives challenges. Beyond game aesthetics, the challenge is also a subject's immersiveness key. The deeper the immersion, the more engaging the game is. The player will appreciate the game as fluid, only when his abilities balance challenges. If the player finds the difficulty level too high, he will get tired. Conversely, if his abilities far exceed the levels' difficulty, he will find it boring. In both cases, the player will not be involved enough to continue playing and he will probably abandon the game.

Clearly, levels are not all the same and not all have the same degree of difficulty. Addressing *Between* for the first time, as for all games, you need an initial tutorial and some very basic levels for consolidating gameplay mechanics. Only after having well understood how the game works, you can deal with a bit more difficult levels and, after much practice, you expect to find even impossible levels. Making a charming game is the aim of gradually increasing difficulty, which pairs with increasing experience. A level designer' work will be considered good only in this case. We want the player to keep playing until the end, driven by the desire to face new levels and see how it ends.

Anyways, this incremental difficulty system also helps the level designer. In the same way, he gets familiar with the process and he finds new points of view from which to observe the available pieces. In Subsection 4.3.1, we have studied medium difficulty levels. This means, a new user would not be able to face them, while they would be too easy for an experienced player. For the purpose of this thesis, those three levels have an acceptable difficulty to detailing examine the potential that this videogame can offer.

According to what we explained, level design process was very remarkable specifically for our videogame. It was one of the two pillars of the game beside coding and it is certainly what makes the game appreciable and exciting at the player's eyes. We want to conclude this thesis starting precisely from the level designer work to talk about the game openings and its possible evolutions from different points of view.

Chapter 5
Future Work and Conclusions

In this last Chapter, we will discuss the crowning achievement of the project. First, we will talk about how to complete the videogame and the story. Next, we will explore more about the level editor. Particularly, we will understand why we need a custom level editor and how we want it to be. Also, we will open a window on a new game mode starting from this custom level editor. In conclusion, we will sum up the whole thesis and the project's course itself.

5.1 - Completing the story with more levels

At this point in the development, it is quite clear which direction the videogame will take from now on. Its structure, as its background story, has been well outlined. As for the time travel concept, there is no reason to change it. The same applies to the functioning of time travels and how they affect the space-time continuum. Even the actions the player can perform and his basic interactions with the main scene elements are already defined enough. However, there is a couple of things we should integrate or improve.

The first point concerns scene objects. At the moment, they are limited in number, so even the level designer's options become very scanty. By promoting the development of many other objects, the possibilities panorama of creating levels would become much wider. In addition to new objects design, we should articulate individual physics for each of them and provide them with new features. We can relate either new items, interactive objects or dangerous objects. Each time we add a new element, the game is enriched, providing greater attractiveness and curiosity.

Further, we can create new hierarchy tree branches, whom we saw in Subsection 3.1.1. If we have always talked about specializing more branches that we already had, until now, why not create new ones? The tree structure allows it easily, indeed, programmers created it for this usage. We chose object-oriented language precisely for playing with the class structure.

On top of this, although the story is completely defined, the levels do not actually cover the same spectrum. While the story has a beginning, a development and an end, the currently available levels do not offer the same coverage. In fact, they are just a small game demo which needs further development by adding many more levels. Firstly, we want to introduce a tutorial and basic levels to leave the player becoming familiar with the controls and game mechanics (see Subsection 2.1.1). Secondly, we have to introduce many more levels that will become more and more difficult, keeping the experience interesting (see Subsection 4.3.2). Finally, we will create the final level, where the player will eventually realize what his future will be (see Subsection 2.1.2).

Always linking to this point, the insertion of writing fragments, whose lead to the background story to make the player appreciate it better. Otherwise, he may not comprehend why certain things happen inside the game and may even find them awkward. By adding small story fragments scattered here and there, perhaps he would be even more curious, because he wants to know the whole story.

5.2 - Level editor

Level designers create locations and scenarios using a level editor and other useful tools. They usually work from pre-production to the completion of the videogame, intervening with both incomplete and complete versions. Almost always, development software already has an internal level editor, such as Unity (see Subsection 1.1.2). The editor helps level designers to formulate level structures without having to modify the game code.

5.2.1 - Custom editor

For the development of *Between*, we used the Unity level editor to position the scene objects only when we had already decided the level design. Certainly, the editor helped to place the GameObjects by hand and not by code, but during the design phase it proved to be obstructing. Being Unity a very powerful game engine created for 3D games, to model our scenes there are many more tools than necessary. This creates heaviness in the designer, who prefers opting for pen and paper during the design phase.

Another drawback is that, while Unity simulates the objects' physics during tests, the editor does not. This prevents objects from being precisely positioned. It is not a negligible detail in a game like *Between*, which relies on objects' location calculation to determine whether space-time paradoxes occurred (see Subsection

4.2.2). To overcome this problem, the designer must test many times the position of each object as the game starts.

How to resolve this? We have decided that, in order to generate the next levels, it is necessary to program a personal customized level editor for *Between*. The features it must have are:

- Setting the correct dimensions for each object. Whenever you import a new object's sprite into the software, you will have to set its standard size, scale and rotation. In a side column, all the imported objects will then be available. The level designer will simply have to drag them into the scene. The items in the column will act as stencils, which generate a self-copy every time they are activated. Once in the scene, the user can move or modify them at will.

- Guides for positioning props. To make it easier and more comfortable to locate objects, it is user-friendly to have guidelines. As in many other software, guides serve as magnets for nearby objects. These objects will snap to them. Our editor will automatically generate guidelines for fixed flat surfaces, such as floors or walls. The user will be able to add other guidelines manually, too, if necessary.

- Grid for automatic creation of elevated platforms. Unlike scene objects, we do not want to drag the platforms to insert them. We need a grid to indicate which squares we want to be platforms. The program will then manage to create them and generate guidelines for them. Likewise, the squares selection must anyways be later modifiable.

- Differentiated layers for present and past. Like in many draw, design and editing software, this software will have layers. By default, there will be two principal layers: one for the present and one for the past. Each layer can then be opened and divided into as many layers as the user prefers. This will give him more freedom in porps visualisation and control.

- Automatic background generation. Depending on the level size and on the elevated platforms' location, the background must spawn autonomously. Assorted stones and decorative elements will be provided to the software, so it will then sort them in the background, thanks to an algorithm.

- Scene export for Unity. Once the scene is finished, we will need to export it to continue working on it in Unity. We did not say we wanted to create a new game engine, just a new level editor.

5.2.2 - Custom level mode

Once this custom level editor will be available, nothing prevents us from making the most of it. For a level editor it is certainly great, but imagine turning it available to a lot more people. A one-man creativity is broad, but limited to his point of view. Distinct people lead to several points of view. Combining many of them can achieve an unimaginable result.

Finding the way to create levels and be able to play them immediately, without exporting them to Unity to finish them, would make level design even easier. Basically, it would simplify it to the point of no longer being a strictly developer-related process. Anyone could place the various objects in a scene, start the level to test it and play it.

A regular level designer will always see levels from the development side, while a player can sniff out new possibilities, watching them for the first time. He does not know why a level is like that, he can only guess what the objects are for. This puts new cards on the table. A player will see more creative ways to intend the available elements, because he failed using them in the first place.

In that respect, we can define a new game mode: a custom level mode. Users could choose whether to play the story mode or at the custom level one. It would be a dual off and online mode. Players could design their own levels and then decide eventually to upload them online or play them locally. Once online, all players can access and play it. This game mode definitely recalls the mold of the recent Nintendo videogame, Super Mario Maker[18]. That game is mainly focused on users creating the levels themselves. New online levels at any time allow the game not to die quickly.

5.3 - Conclusions

In the context of this thesis, we examined in detail all the development steps of our videogame *Between*. First of all, we exposed how the idea behind the game began, to get to its entire concept. We have thus entered into the two main focus fields: game design and code implementation. We discussed the player's involvement, both on a narrative and a gameplay level. Later, we entered specifically into the game designer's role, differentiating his tasks in the game feel, such as content and world design. After that, we presented the class hierarchy

[18] *Super Mario Maker* (first version), September 10, 2015, Wii U, Nintendo, Japan.

which the code is divided into and we extrapolated its most interesting parts. Finally, we talked about the project's highlight: time travel. We studied its functioning and the multiple timelines system caused by space-time paradoxes. We did not stop at the concept, so, once again, we went in between the code lines and the level editor's grid. In the final analysis, we have explained the project's future work. Actually, we even thought about how a custom editor would make level design easier, opening a window on a new game mode.

Focusing on game design and code implementation was the right choice for this project's realization. Due to the central time travel theme, these two have certainly been the sectors most affected. It was important to deepen how they were accomplished both from a design and coding point of view. In other videogames, there are equal or more important departments we have not considered here. There are so many specialists for every sector and it would have been wrong to incorporate all the aspects of videogame development here. There are hard studies behind every field in this industry, studies that take years or even decades and will never end.

In conclusion, as a beginner, the work was intense and fascinating, but sometimes complex. Deepening the time travel theme has been a very exciting and exciting challenge. It has led to think in other ways and make the best usage of every available component. If on the one hand the experience was engaging, on the other it was critical. Despite the great online material, reading or watching expertise people at work seems always simpler than it is. Unity has made the journey easier, providing very advanced tools, sometimes even too much. One time, the affordability for inventing something complex was weak and to be capable of attaining it, you had to add new capacities. Today it is the opposite. To create articulated projects you have all the tools in front of you, but to produce simple things you have to remove functionalities.

References

Bibliography

[1] *Swink, S. (2008), Game Feel: A Game Designer's Guide to Virtual Sensation, CRC Press.*

Sitegraphy

[2] Unity, https://unity.com.

[3] Unity Documentation, https://docs.unity3d.com/ScriptReference/index.html.

[4] Microsoft Docs, https://docs.microsoft.com/en-us/dotnet/csharp/.

Ludography

[5] *Donkey Kong* (first version), July 9, 1981, Arcade, Nintendo, Japan.

[6] *Super Mario Maker* (first version), September 10, 2015, Wii U, Nintendo, Japan.