UNIVERSITAT JAUME•I

Implementation of immersive technologies with Virtual Reality to a video game

Antoni Arlandis Daviu

Advisor: Marta Martín Núñez

Final Degree Work

Bachelor´s Degree in Video Game Design and Development

Universitat Jaume I

May 20, 2019

Acknowledgements

To begin with, I would like to thank my colleagues Joaquin Soler Tomás and Daniel Castro Arjona who have developed the area of art and narrative for the great effort that have been devoted throughout the development of the game, without them it would not have been possible to get to where we are today. On the other hand, I want to thank Marta Martin, who has helped us make this work possible and to all the teachers that we have trained us throughout the university degree. Finally, I want to thank my family for the great effort they have made so that I can be where I am today.

Abstract

This document presents the final report of the Video Game Design and Development project. The work consists in the implementation of game mechanics to implement. From the simple mechanics such as the tracking of the head and the drivers, to the more complex as can be the grip of adaptive. All these mechanisms are implemented in the most efficient manner possible. Unity is the game engine used, working in an immersive experience through the use of sound, profiles of post-processing and lighting designs with a result that is optimized for the VR.

Key words

Video game, Virtual Reality, Puzzle, Immersive technologies, Oculus Rift.

Index

Figure Index

1. Introduction

1.1 Work motivation

At present, we can observe that within the world of video games, Virtual Reality technology is booming, due to this and with the recent acquisition of this technology we decided to explore it and exploit it. Looking at its great ability to dive and its power in the field of video games, we could see that we are facing a growing market and an opportunity that we should take advantage of. It is for this reason that we decided to implement a video game in which it is used, specifically we can qualify it within the genre of horror because we considered that this genre is the most suitable for the extraordinary features of the technology of Virtual Reality such as immersion, surround sound, visual impact, among others.

1.2 Objectives

The work done here is part of a project developed by three students, with the aim of using VR technology to make a complete video game. The project is divided into three main parts, the programming that I will take care of, the art as part of the work of my partner Joaquin Soler and finally the narrative and lighting by Daniel Castro. The part to be covered in this project is the part of programming, that is to say, the one which carries the mechanics and the events. In particular, this project is divided into three main parts:

- Special mechanical world of the VR

The first and most basic is the mechanical movement that follows the devices of VR, so that the hands and the head of the player follow the movement made by the devices located in these parts of the body of the player.

Secondly, the most common actions in video games VR are the actions of capture and drop/throw things, at the end of the project we must have a mechanism to perform these actions efficiently. These actions will be performed so that the code is usable for all interactions with all objects without exception, thus being reusable for other cases.

Thirdly, the actions of moving through the world of virtual reality, where the movement is split into two variants. On the one hand, the movement with the Teleport, which allow the player to move to a point of the scene by pressing a button. And on the other hand, the movement with the joysticks on the remote control so that one of the joysticks controls the rotation and the other one the player progress.

- **Mechanical hand movement**

One of the things that you need to program and that most of the games of the VR does not have is to have a grip dynamic of the protagonist's hands. In this section, we will schedule a natural hands movement when taking objects from our environment, thus providing a greater accuracy to the game and a greater degree of immersion. It is intended to make the fingers movement according to the press of the power buttons, since these buttons are of decimal value, that is to say, from 0 to 1 by adding hundredths of a unit, the player will see as their finger moves forward in the grip progressively. For example, by receiving a button with a pulsation value of 0.5, the player will see that their fingers are closed to 50%. Another essential aspect is the hand adaptation to objects dynamically, that is to say, that when they close your hand the objects are wrapped in the most natural possible way. This is one of the most important aspects, because, currently, there is a total absence of assets in the videogame engine Unity for this type of activity and many of the games of the VR does not have it. So, it would be a great practice and an interesting point.

- **The puzzles and events**

The puzzles will be accomplished individually. Each of them will have one or several scripts that should be related to the script of grip to perform their different functions. On the other hand, all the events of the game will be programmed so that the story proceeds as planned. All of these events will be carried out through different operations such as: the use of triggers with the user, the collision of rays cast from the view of the protagonist, the distance from the protagonist to a certain sector, among others. Any other not specified functionality but necessary for the proper functioning of the game will also be part of the project.

1.3 Initial state

This is a job that is done from scratch, relying solely on the technology of VR, Unity, and the integration of this technology into our game engine. Any kind of test with this technology has been performed, so that it will be necessary to collect information both on the technology as on the application of the same to the engine of the video game. Moreover, it is a project that is part of a collective project of two more people who are in charge of the design and the art of the project, thus the level design, mechanics and events will be the decision of the responsible for the design. I would also like to mention that the use of the active integration of Oculus in Unity, will allow us to use the functionalities of the technology VR, in our case Oculus.

2 Planning

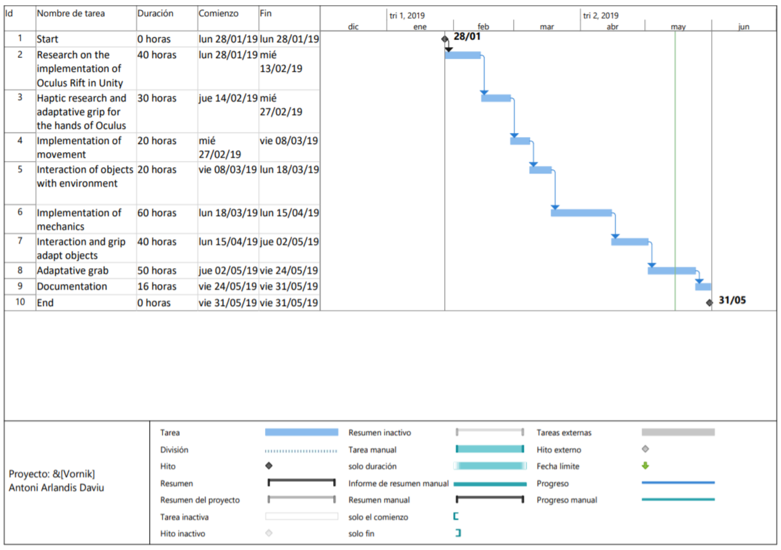
For the proper resolution of the project the work has been divided into parts for being each of these parts essential for the development of the next one.

First, we consider the performance of mechanisms such as the movement and the grip of objects, which is essential for a game of VR. Prior to the execution of these two activities, we searched for information about the technology and the implementation of the same in the game engine in order to assure their optimal use. The resolution of this part of the work is estimated to be around 100 hours. This number of hours comes from the sum of 40 hours gathering information about the technology and its interaction with the engine of the video games. Other 20 hours of implementation of the movement with the joystick and teleport. Adding 40 more hours to the execution of the grip, which is an interaction combined between the objects and of the player controller.

Secondly, the main basis of the project is the dynamic grip, so in order to be able to fulfil a dynamic efficient grip we will provide three types of implementation of grip with two subvariants each of them and choose the most efficient and realistic one. 120 hours will be required for the development of his work, embracing 30 hours of driver research and the physical internal of the engine Unity, adding 40 hours to the study and to tests performance on the different options of use for the correct implementation of the mechanics. And finally, 50 hours have been devoted to its implementation and correction of possible faults that may arise.

The third part is concerned with the implementation of the different puzzles and events that will shape up the video game. 60 hours will be lasted for the aforementioned correct implementation of puzzles, secondary mechanical as the lantern, secondary object animation, etc. On the other hand, it will take us 20 hours the implementation of events, such as activation of different situations or events, modifications of environment, etc.

Task	Time
Research on the implementation of Oculus Rift in Unity	40
Haptic research and adaptive grip for the hands of Oculus	30
Interaction and grip adapt objects	40
Implementation of mechanics	60
Implementation of movement	30
Interaction of objects with environment	30
Adaptative grab	50
Final memory	10
Presentation work	10
Total	300



Id	Nombre de tarea	Duración	Comienzo	Fin
1	Start	0 horas	lun 28/01/19	lun 28/01/19
2	Research on the implementation of Oculus Rift in Unity	40 horas	lun 28/01/19	mié 13/02/19
3	Haptic research and adaptative grip for the hands of Oculus	30 horas	jue 14/02/19	mié 27/02/19
4	Implementation of movement	20 horas	mié 27/02/19	vie 08/03/19
5	Interaction of objects with environment	20 horas	vie 08/03/19	lun 18/03/19
6	Implementation of mechanics	60 horas	lun 18/03/19	lun 15/04/19
7	Interaction and grip adapt objects	40 horas	lun 15/04/19	jue 02/05/19
8	Adaptative grab	50 horas	jue 02/05/19	vie 24/05/19
9	Documentation	16 horas	vie 24/05/19	vie 31/05/19
10	End	0 horas	vie 31/05/19	vie 31/05/19

Figure 1: Gantt chart

2.1 Related courses

VJ1203 – PROGRAMMING I

VJ1218 – PROGRAMMING II

VJ1222 – SOFTWARE ENGINEERING

VJ1227 – GAME ENGINES

2.2 Resorce Evaluation

In order to assess the necessary resources for the correct implementation of the game Vornik I would like to highlight that the implementation of this game has been made as a result of the collaboration between three people, therefore these three people have made their own contribution in this project possible.

Individually my material costs are, the use of a computer powerful enough to withstand without problems the Virtual Reality goggles and VR of Oculus technology, including, in this case, the trackers, controls, goggles, etc., apart from the videogame engine Unity with a free license to develop the game. In addition to the effort to find and pick up additional information on this technology due to our lack of experience it took us some time to get used to it because it is an emerging technology and there is still some new information that do not work properly as the SDK Oculus for Unity.

3 Game design

3.1 Synopsis

Horror Video game for Oculus Rift, whose main character is Zledic, a mysterious figure with a terrible background who we must control, throughout the game, the player will have to solve different puzzles in order to understand the story of our protagonist. The mechanic provided by the virtual reality will be used for the resolution of these puzzles.

3.2 Overview

3.2.1 Game Concept

Vornik is a game of puzzles based on the memories of an old commander, in this game we must go over various memories of our main character using the Virtual Reality technology to discover its story. For this implementation, we will use the technology, Oculus Rift in order to get a better immersion. The art style will be realistic and volumetric with 3D sound implemented.

3.2.2 Genre

Horror, puzzles.

3.2.3 Target audience

Vornik is designed for players who like terror games and puzzles, who, in addition, own the technology Oculus Rift.

3.2.4 Look and feel / game experience

The game is designed to achieve a high degree of immersion in the player. Using an intuitive and realistic mechanical.

4 Work development

4.1 Principles of programming

Vornik is a horror video game that tells the story of a commander and his evolution during and after the war. Our video game will have a simple but realistic mechanical to improve the player's immersion. We decided that this was the most appropriate kind of mechanics to the genre whilst ensuring that is different from the large number of games that already exist with a specific mechanical. This is the specific part on which to base this project. For the correct accomplishment and compliance of the objectives the code will be implemented in a way that it is fully reusable in any VR game. This requires to follow the programming guidelines based on SOLID.

SOLID, are guidelines for a proper programming, based on five guidelines:

> 1- (S)PR or Principle of single-responsibility: the notion that an object should cover only a liability.
>
> 2- (OR)CP or Principle of beginning/closed: where it is marked that the scripts must be open for re-use but closed for modification.
>
> 3- (L-SD) or Principle of replacement Liskov: given an object, this object can be substituted for a subpart of the same and work properly.
>
> 4- (I)SP or Principle of segregation of interfaces: supports that a general interface is worse than many specific client interfaces.
>
> 5- (D)IP or Principle of dependency inversion: based on that you should not depend on implementations.

Throughout the project we will try to follow the principles of programming based on SOLID, for the possible reuse of code, create an individual code with a smaller number of dependencies, and a great power of re-use without modifying the scripts.

4.2 Basic programming concepts

I would also like to mention a series of guidelines that will be followed throughout the project to improve its performance.

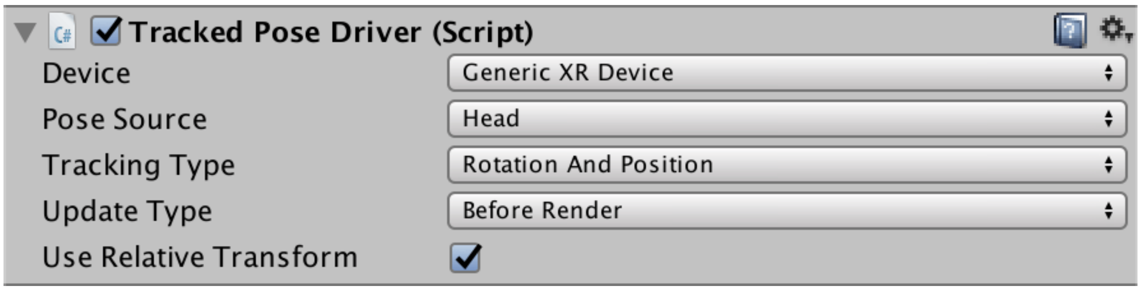
In the first place, in the game, the collision detection is carried out using colliders, which is very expensive when you use meshColliders. Therefore, throughout the work, we will use primitives' colliders for the composition of the collision mesh of the objects. In this way, the potential problems of in these meshes as removed and resulting everything related to the collisions under better control. This is also done through the use of x-rays for the detection of objects, because the raycast is extremely more expensive if it has to be check collision with objects, whose objects have meshCollider.

Another important aspect is the pairing of the objects using the hierarchy of Unity and positioning and rotating the objects from its parent object. This means that when we join one object with another, actually what we are doing is marking it as child of the object to which we want to join it. In this way, with Unity, we will facilitate the rotation and the positioning of the object in their respective corresponding place. This is a good practice due to it is not very expensive for Unity to have a very wide hierarchy, but it would be more expensive if you push the rotation and position of the object following another, or even if we used a physical union to do this as this would cause internal collisions with the object.

We also must, keep in mind that when an object, it becomes a total trigger, that is to say, the collision is turned off. In this way we ensure that if two different objects with rigidbodys collide at a fairly high-speed Unity will be able to be a good representation of that collision, avoiding that weird effects, as to put the object in an incorrect position or that the collision triggers a deformation in the object and colliders of the objects, can happen.

4.3 Tracking

At this point I should like to add the importance in the mechanics tracking and devices positioning used by the VR technology. To synchronize the controller's position and the virtual reality goggles we have used some components belonging to the Unity package that applies the Oculus VR. This component is called the Tracker Pose Provider.
Such a component performs the function of tracking the position of the chosen device to the assigned object. It has variables for its operation, as a Boolean that specifies whether to use a follow-up with position relative to the transformation of the object, a pivot where you can put two of these traces, and an instance of the Pose class that provides to the Pose Tracker Pose Provider the pose to be taken by the object. Thanks to this component, the tracking is a very simple task. Nevertheless, it turns into a disadvantage, since the Tracker Pose Provider is not an internal package of Unity but a part of the Oculus VR asset, so it is not known exactly if it would work with another device.


Tracked Pose Driver (Script)

Device	Generic XR Device
Pose Source	Head
Tracking Type	Rotation And Position
Update Type	Before Render
Use Relative Transform	✓

Figure 2: Tracker

4.4 Grabbing Objects

The main mechanics of the game is what it is expected of a Virtual Reality game. This mechanics consists of trapping, throwing and interacting with objects in one way or another to achieve our goal. This mechanism has been implemented through the creation of two scripts, one for the object to be capture and the other one for the hand that is meant to capture the object, these two scripts work independently.

The functionality of the hand is to spot when it collides with an object, in addition, it detects the touch of a button, implemented as a grip button, when it happens, it saves the reference of the object to know which object is being taken and calls to the Grab() method of the object (which will be explained later). On the other hand, the hand also provides the reverse detection, when the user releases the button of the grip is also sensed and when this happens the action to release the object will be delegated. With this we make the object implement all its actions, consequently all what the hand has to do is call the Throw() function of the object and agree that the object which being taken is nil, by specifying that there is none.

With respect to the functionality of the object, it becomes a little more difficult due to it must implement functions that serve to any change that affects to this object and for any other event. The object is composed by two main methods, Grab() and Throw(), which will be explained below.

We will begin by explaining the Grab() method, the grab method is in charge of linking the object with the hand, eliminating the physics of the object to avoid unnecessary collisions between the hand and the object, by setting its great local rotation for gripping and keeping the hand that keeps it, as a reference point. When the object is related to the hand, the position and the rotation of the object becomes relative to the hand, this means that when the hand moves or rotates, the object will rotate with it. What is more, we modify its local rotation to position the object with what we call a natural rotation, and with this we achieve that the case of forced grip or an unnatural position disappears.

The second function of the script is the Throw() function, this function takes care of releasing the object, to do this we need two steps. In the first place, we activate all of the physics as it is to use the gravity, update speed thanks to the current position and the position in the last frame of the hand, with this, we get the speed that has followed hand in hand in the last frame, and we set their colliders to re-collide with the objects in the scene ensure that this not transpose walls and does has a correct physical. In the second place, the object is set as no related to anyone, this means that it will go to the highest part of the hierarchy and will not belong to anyone. Any more, thanks to this we get a realistic functionality to catch and throw objects due to that when objects are thrown, the gravity and the speed at which the hand moves will be applied.

Prior to this implementation we have performed two tests with another type of association of one object to another, without using the relationship.

The first test was the use the tracking of objects, when you picking up an object, it was tracked to the position of the pivot assigned inside the player hand. In this case, such an implementation caused a lot of errors because the script was forced to use the object Pose, which is an external library to Unity, thus the Unity manual did not have all the information necessary for its proper function. Wherewith errors of positioning of the object were displayed and when using of two types of follow-up in the same object, the own and the of the hand, the object started to make strange rotations due to the clashes of these two components.

The second test was the use of an object Join in the hand which was tied to the hand and the object through a physical articulation. But this practice turned out to be a great failure as a collider and a rigid body was required in many of the objects. These objects caused errors when colliding with the hands and it could be noticed as the object trembled in contact with the hand due to the internal shocks between the object and the hand.

4.5 Special mechanics for objects:

Another of the mechanics studied is the interaction with many different objects: the player can interact with a large number of objects with entirely different functions. This way the player will have to use each object in an appropriate way to pass the various puzzles. In this section of the project different scripts have been carried out, one for every single one of the features of the objects so that each object will have a different script that will implement its functionality. Down below, I will describe each one of the scripts of the project.

4.5.1 Lantern

The lantern is an object whose key function is to illuminate the stage, but to add a note of realism, we have decided that it works in such a way that at the touch of a button located on the thumb, it can be switched on and off as if it had a switch turned on. The script has been implemented in a way that it senses that the push of the thumb button will change the status of the on and off flashlight or vice versa. The state of the flashlight goes on and off by activating and deactivating the light emitting object which the flashlight, the operation of the issuer follows the following logic: when you turn it on, it emits light and when defusing it stops shining.

4.5.2 Matrioska

The matrioska is a set of objects that are located one within another. You have to open the objects one-to-one, and at the end you will find the key of the music box. This script hides a special functionality, because when it finds the error of placing the different objects of the matrioska one within the other with their colliders and their respective rigidbodys (component that simulates the physics of objects) these collide among themselves and move around the stage.

This situation led us to implement an extra functionality to the matrioska to keep the reference of its inside object and when it was caught its physics could be removed. Then the physics of the inside object would be activated so as to make sure that there is not any failure in the physics.

The implementation has been carried out thanks to the relationship between this code and the access code from which you can obtain information about whether the matrioska is being collected or not at the time. So, this is an example of the ease of apart from the effective use of the writing grip of the object.

4.5.3 Teapot

The teapot is the most complex puzzle. Its goal is to join the pieces of a teapot which is broken in 4 pieces, when you post it, this will trigger a memory. For the realization of the script of behaviour the main idea has been to give a level of importance to each piece of teapot, following the logic of the piece with less importance would join the piece with more importance.

But then there was a problem, because there are a lot of possible combinations and it was difficult to control the position and rotation of the object to attach the piece. For this reason, the object has as many pivots as pieces below its level, so that each pivot belongs to a part level. Therefore, each pivot will have its own script that will store the information of what level corresponds to. Moreover, each piece of the teapot will maintain its importance apart from its functionality of union to the rest of the pieces.

The main functionality of the script is to detect the collision of the workpiece with a pivot, to check if the pivot corresponds to the level of the piece, if so join the pivot and mark it as its father and rotating to its default position. Research has found that if by chance the parts were close and two of them collided into each other, they joined without the player necessarily do anything. So, we added an extra check by using the script of grip that provided information about whether a piece is being taken by hand or not, and if so, it could be united. At the end of this process the logic of the script is the following: If a piece collides with its pivot, this piece will check whether it is being caught and if the piece that contains its pivot too. If so, this will become the daughter of its pivot and therefore it will join to this one and it will remove all the functionality to this piece.

4.5.4 Music box key

The key of the music box is an object of placement, that is to say, it is an object whose primary function is the detection of the collision with the object into which it must fit. In other words, this object will detect collisions with the lock of the music box and when colliding with it, the object will be positioned automatically and perform the animation of rotation of 360 degrees, simulating that it is being rolled up like old music boxes.

Furthermore, the script would also encourage the lid of the music box that would open to 90 degrees. This script also uses the relationship with the object script of grip, as when it senses the collision of the lock, it calls the hand script to release the object and place it in the music box so that the key is completely free as well as the hand. If it were not so, an error would occur in the game because when trying to position an object that is already connected (with the hand) with another one (the lock of the music box), the hand will not be updated and would still be related to the object therefore when releasing the button to catch the key such a hand would have null as a parent and would not be positioned correctly.

4.5.5 Monocular

The monocular has a very powerful function in the story, even though it has one of the simplest capabilities. As with the flashlight, the monocular uses the object script of grip to be activated. So, this will only be activated if you take it on and it will be disabled when you drop it. The functionality of the monocular is part of the project of another of my co-workers.

4.5.6 Labyrinth box

The box and the maze have two different functions. The first function comes from the relationship between the ball running through the maze. Inasmuch we have to carry the ball through the maze, trying to lead it to the goal and it is at that time when we must detect the collision, that is to say, we must verify that it has reached the end. When this occurs, the second functionality, which is the animation of opening the box, will start.

When it comes to the ball, we must consider that it has a rigid body, which hampers its operation. This means that when making fast movements the object can be ejected and it will have been achieved by restricting the gravity if the object is not captured and will be checked by the object script of grip. In this way, the verification will enable or disable the use of gravity in the rigid body of the ball. In the second part we have a collider at the end of the labyrinth, which detects the collision of the object, when colliding with the box will be opened by following an animation provided by scripts where the cover will rotate 90 degrees at a speed of 1 degree/frame.

For this puzzle another functionality for the ball has been implemented to avoid possible failures in the physics. This is based on the fact that if the ball comes out of the music box due to a bad collision, this will be returned to its initial position.

4.6 Vornik special mechanics:

When we speak of RV special mechanics we are referring to the mechanics of objects capture and launch, with the hands tracked and a tracked vision that helps a lot in the immersion and interaction of hands with objects. Nonetheless our game stay one step ahead and we have included special mechanics designed for this game, such as, for example, to respond to questions with head movements, and interact with a numeric keypad with the index finger and not with the whole hand, for a greater sense of realism to the simulation and the manipulation of locks with this same finger.

4.6.1 Answer mechanics using head movements

One of the unique mechanics to our game, is the responsiveness to the questions by using only the player head movements. The possible answers are yes (equivalent to a vertical movement repeated a minimum of two times) and no (equivalent to a horizontal movement with the same repetition as above). Four colliders placed above and to the sides of the head and two different types of script have been used to this mechanic.

The logic of the mechanics is to use a raycast to detect collisions with the colliders that are located in the four positions mentioned above. When a collision is detected, the object with which it collides is stored in a list of objects to be able to access it at any time. As soon as the list exceeds three objects, it is checked whether the last three collisions form a correct answer. If so, the response will be sent to the script of narration narrative implemented by Daniel Castro and this script will be disabled.

The verification of the response is performed using the names of the colliders and getting the four possible configurations "Up, down, up", "Down, up, down, left, right, left" or " right, left, right ". So, if the last three inserts do not form an answer, this step will be repeated every time you answer is inserted. It has also been added the condition that the first insertion checked and the last have a spacing between them less than a stipulated period of time so as not to confuse a response of an observation of the environment. This is calculated by maintaining at all times the time in which every iteration it has been performed and seeing the period of time between the first check and the last.

4.6.2 Pressing buttons with the index

It is a mechanical not very well developed in the virtual reality world since in the vast majority of games we find the press of buttons by the collision with any part of the hand. In our game it is different, the collision is special. The collision is performed with a hadron collider located at the top of the index finger so that you only can press the object with this part of the finger. The explanation of the script is simple, if the button detects that it has been touched by an object whose collider belongs to the label "index" it will be activated with the single touch of the index finger, otherwise it will not work.

4.6.3 Padlock with drag

This is another of the mechanics that we have not seen in any virtual reality game. The mechanics consists on being able to move one of the wheels with the numbers on the padlock when you pass a finger on it. In other words, if we pass the finger down we will add a digit at the lock and the wheel will spin down, otherwise, if it is upward, the wheel will rotate in this direction and one unit of that digit padlock will be subtracted. For this mechanism two ways of interaction and logic have been implemented. Both totally valid and with the same result.

The solution has been applied with two different colliders, one located at the top and the other at the bottom of the object. In this way we detect when the finger hits a collider. When we collide with one of them we keep this collider as a collider of insertion and, in the same way, if we already have the insertion one and it re-collides again, it is saved as a collider of output. By having these two colliders we understand that, if they are different then the movement has been vertical, on the contrary, we will ignore the iteration. If we deal with a vertical movement we find the direction by comparing the heights of the Transforms of the colliders, and restore the direction.

The second script of operation follows a similar logic, but in this case, you only use a single collider. This collider will detect the collision with the index finger, when this happens, it will keep the position in which the input collision has been detected, and the location in which the output collision is detected. With these two facts in mind we compare their heights, so that if the difference in height is equal to the height of the collider this will mean that the movement has been made up or down, so it will have been a right move. Thanks to the input and output locations we can find the movement, since in the entry movement is greater than the output one, it will be a downward movement, and if the entry is less than the output, it will be an upward one.

4.7 Grab

This is the most relevant part of my project due to the large number of tests that I have done and all the documentation that I had to find about the topic. This part of the work is based on implementing a dynamic and adaptable grip to the objects of the scene with which we interact. To better explain this part of the project, I have divided it into three different parts. The parties will be explained in chronological order.

4.7.1 Study of movement

The first part has been based on the collection of information about the three factors. First, the factor of the natural movements of the hand and familiarisation with the concept of haptic. The haptic is the science that deals with the world of touch, the relationship between any part of the body and its interaction with the environment. As a result of this practice I discovered that the possible movements of a hand are divided into three basic movements.

The first of the movements is the force grip, so called because it is the bending of all the joints of the fingers toward the inside of the hand so that the hand is almost closed in the form of a fist. In this way, all the joints will spin towards the centre of the hand until they collide with the object, and at this point they will be stopped. This is the most common grip in nature.

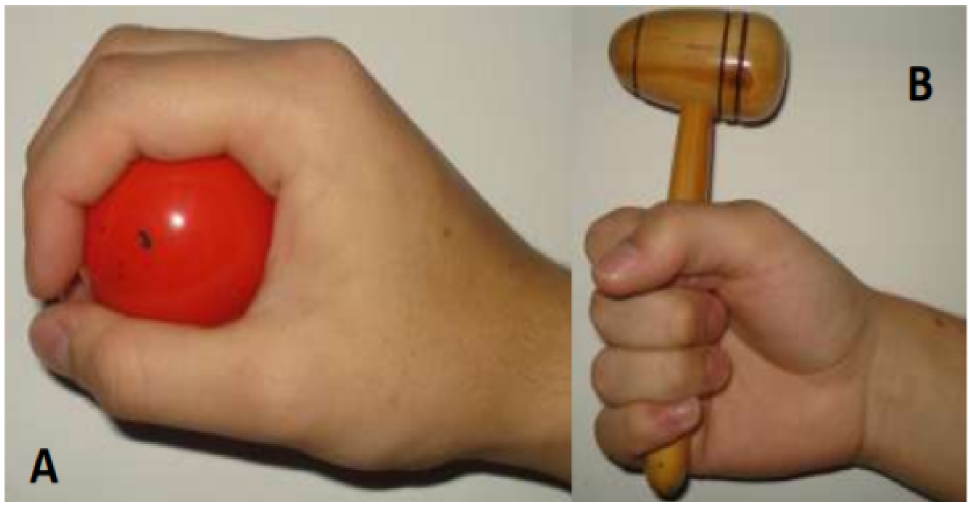


Figure 3: Force grip

In second place we face the grip precision, that is the distinctive grip of a pen, a pencil or even a key. This grip is characterized by a provision of the fingers with a single rotation, in the first joint of the fingers holding completely the rest of the joints straight, this grip is called precision because you get a great accuracy of movement due to all the force is focused at the tips of the fingers.

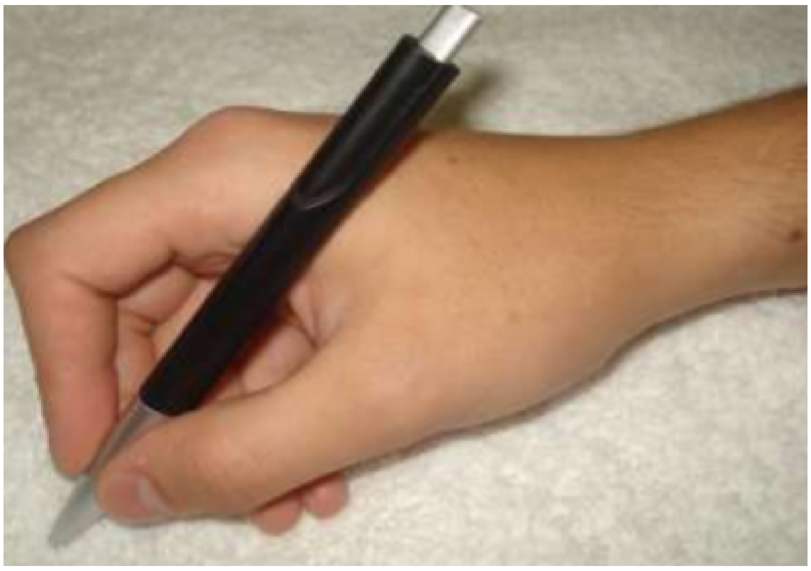


Figure 4: Precision grip

The third and last type of grip is the category of specific or derivative grip that includes all the grabs which are not included in the previous categories, such as the grip when picking up a book to read, the grip of a console controller, etc. This type of grip is independent for each of the objects to which it is addressed.

In summary, in nature there is a great variety of grabbing, but the most common is the force grip. As a result of this research we decided that I would not have time to implement separately the two types of general grabs and that we only would focus on one of them and on the basis of the studies carried out and the objects to interact with in the game, we decided that finally we offer you the grip strength.

4.7.2 Study for effective implementation

The second part of the implementation of a grip on the models of the hands for our game, was to study which of the ways to carry it out was more favourable. As a result, I decided to investigate the best methods to use this functionality and carry out various tests in order to put it into practice. While researching a little about this functionality, I realized that there was no information at the time of implementing a grip dynamic based on the surface of the objects or any type of asset on the Unity AssetStore that met this functionality. As a result, this handicap was divided into different topics.

4.7.2.1 Coordination with controller pulse

To mark the percentage of grip we will use one of the buttons on the controller that has a range of touch, and depending on the force with which it is pressed, it will return a value between 0 and 1. Being 0 the lack of touch and 1 the total touch. After having checked these values it was decided to make a maximum rotating for each joint of the fingers, and use this value in the controller as a multiplier of the rotation. For example, if the button is pressed by half, we obtain a value of 0.5, hence with a turnover of maximum 50, the rotation at the point of 0.5 will be 25 degrees which is equivalent to 0.5 X-50.
This was our initial idea but when we put it into practice we realized that it was wrong since when pushing the butting quickly in could not be updated with sufficient value and with a quickly pulse the object went directly from 0 to 1 by forcing us to close the hand suddenly without making any type of transition. So, we decided that we would use the previous value as a target for the turn, and not as a final turn. In this way, the rotation would advance each frame a specific number of degrees in order to reach the goal.

4.7.2.2 Collision detection:

For the detection of collisions, the two main detectors of Unity are the colliders. They consist of geometries of collision which use the methods OnTriggerEnter and OnTriggerExit to detect the collisions of the objects with other colliders that hit them and also when they stop colliding. On the other hand, there is the raycast which throwing a virtual lightning bolt to some distance succeeds in detecting the collision with other objects. We implemented two types of tests, with detection of raycast and another with colliders. At the time of the test, we realized that the colliders were the best option by far due to two major points.

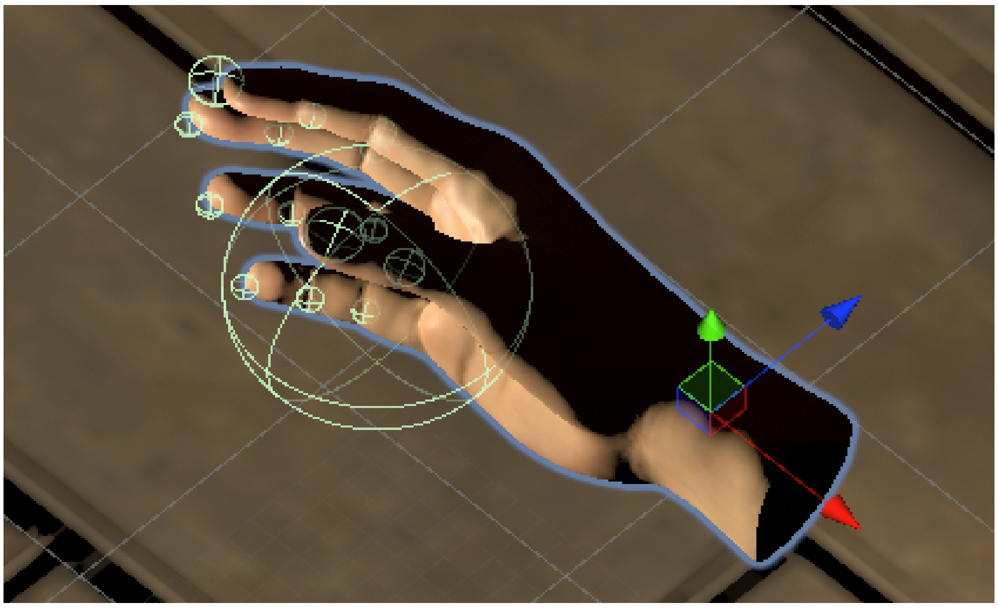


Figure 5: Primitive Colliders

The first error of the raycast is that they follow a special direction, and when it detects the collision of an object that is in constant rotation this may be the case that, at the time of the collision the raycast is not pointing to the correct side. Due to this we observed that when catching different objects, we must detect the collision in different directions, not only toward the inside of the hand. On the other hand, this error does not affect the colliders because as it is a geometry of the collision the incoming collisions can be detected from each of the points of the edges of the geometry.

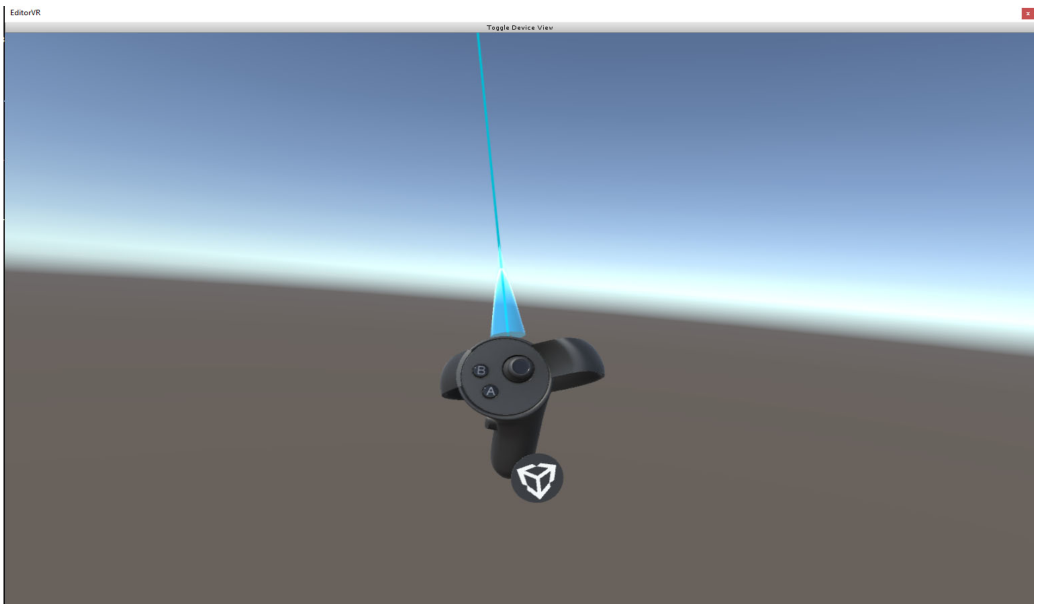


Figure 6: Raycast

The second error is due to that there are two ways to detect a collision with the raycast, the first is to establish a close distance to the origin point, so that when the object to collide is at this distance the raycast will detect it. Another way is to establish a greater distance, and when the raycast detects that the object with which it collides is at a distance less than the minimum it will perform the action that may have been assigned. These two types of collision detection do not achieve our goal, because our object will rotate instantly and iteratively a series of angles to mimic a natural grip. Therefore, in both options there may be an occasion in which this fails because if the rotation is so large as to enter the object, the object will not be able to detect the collision and will not perform the action. This happens when crossing the surface of the object is inserted in it and does not detect collision with the surface. This does not happen with the colliders, because even if one of them enters the collision will be detected without problems.

4.7.2.3 Iterative implementation with controller or recursive

After several tests, I decided that the implementation would be a iterative controller script. A script located in each of the joints of the fingers that it would work individually informing the controller of their situation and that is updating the state of the other if necessary. But later we agreed that there was no need for a controller of this type, because with a simple reference between a script and its predecessor in the finger was more than enough. We did not decide to make a recursive implementation as we do not find an effective way to check the collision between two unions that are not connected, without the need of a reference to each one of them, that would prove to be more expensive and less elegant.

4.7.3 Implementation

Finally, I will explain the implementation of the script that establish itself in each of the joints of the fingers as it follows a bit difficult but effective logic. Each joint of the fingers is initialized with a script and a collider located at the same joint. The logic of the script is explained below.

In a first step, the script receives the user's heart rate information, and estimates the objective rotation of the finger. Once calculated this information, the script is divided into two parts that work the same way. If the objective rotation is lower than the current one, five unites will be rotated to detect the collision or reach the maximum, if on the contrary is greater, will be rotated five units but in reverse. When a joint detects a collision, it warns the previous one to stop, but not to the subsequent, so that if it collides with the tips of the fingers, the whole finger will stop, but if on the contrary, if it is an intermediate joint the tip of the fingers will not stop. This way you can easily follow the times followed by the player.

On the other hand, there is another setback condition that is not applied to the advancement of the rotation and in case the object collides, the rotation of the collision is stored, and the recoil does not begin to recede if the rotation is not smaller than the rotation of the collision.

4.8 Movement

The player can move through the setting in two different ways. First, we have the teleport, the this will allow the player to move immediately through the setting by selecting a point of emergence, that is to say, the player will mark with the driver, a place that is in sight to teleport and when releasing the selection button their will be teleported there.

4.8.1 Movement with Joystick

The rotation with joystick is the only type of rotation that can be used by the player. The rotation follows the following logic, moving the joystick to the right it will rotate 30 degrees to the right relative to the body at once, so that there can be a case of concatenating too many times the rotation and cause dizziness because the virtual reality is likely to cause this type of sensation. The same factor is applied to the left rotation. This rotation will be in the position of the avatar and all the transformation, which is the component that receives the information of position and rotation. The rotation will be applied using the Rotate of the component Transform function, which rotates the object to the values that are assigned to and in the axis that is assigned to.

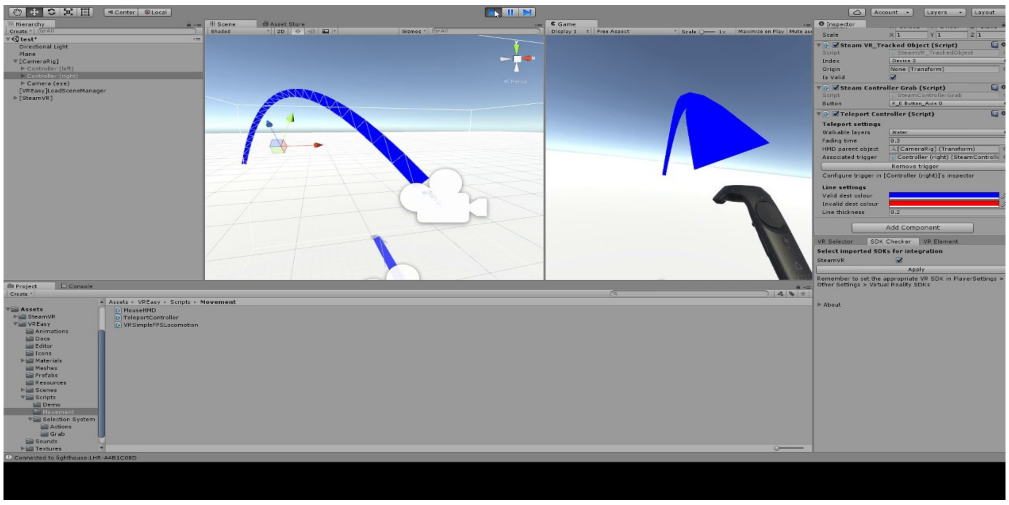
The movement implementation with joystick has been carried out by using the joystick as a multiplier of movement. In other words, the joystick returns a value for each axis of movement. In the first place, the x-axis that covers the movement to the left has a value of -1 to 0, and the movement to the right which has a range of values from 0 to 1. The same way works the axis, it has two movements with these ranges over 0 to 1 unit and below -1 to 0 units, respectively. These units returned by the driver determine a 3D space direction.

With this in mind, an example of direction would be the tween to move the joystick with a diagonal movement towards the upper right corner that we would return the same direction. But this interpolation would refund to a value greater than 1, so that we should standardize it. These directions would depend on where the player is looking at, not on the direction to which it is positioned.

4.8.2 Teleport

The reciprocating motion is realized using a Teleport, this mechanism is very simple, by pressing the button to teleport, a bolt of lightning could be thrown to collide with a surface marked as tele transportable, a marker would appear. When you release the teleport button if the marked position is correct, the player would teleport to that position without changing its rotation. We decided that the beam of the teleporter would not a simple straight line to mark but it would be through a parable to simulate a jump into the space.
This we developed a script with a default number of rays that vary their angle with respect to the vertical, that is to say, they pretend that the farther away they are from the source, the more affected by gravity and the angle is lower. This only serves to simulate the beam, which with the use of the component LineRenderer makes the player observe at all times the parable that follows his Teleport.

In the same script is continuously updated within the Update function where it is colliding the first section of the curve that it detects a collision. Its role is to set a layer of collision, and if the object with which it collides belongs to this layer is activated, the right collision of the marker will be activated, and the button is released here, the position of the avatar will be set as the position of the marker. To do a teleport of all the avatar is we include the hands, head, and all that in that moment belong to the son of the oriented avatar.



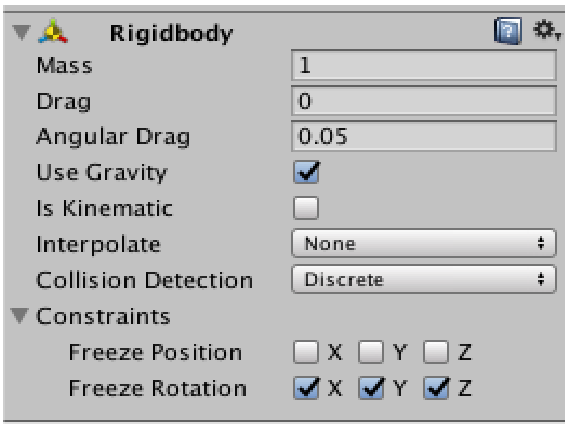
```
Steam VR_Tracked Object (Script)
Script              SteamVR_TrackedObject
Index               Device 3
Origin              None (Transform)
Is Valid            ✓
Steam Controller Grab (Script)
Script              SteamControllerGrab
Button              K_E Button_Axis 0
Teleport Controller (Script)
Teleport settings
Walkable layers     Water
Fading time         0.3
HMD parent object   [CameraRig] (Transform)
Associated trigger  Controller (right) (SteamControll
Configure trigger in [Controller (right)]'s inspector
Line settings
Valid dest colour   🟦
Invalid dest colour 🟥
Line thickness      0.2
Select imported SDKs for integration
SteamVR             ✓
Remember to set the appropriate VR SDK in PlayerSettings >
Other Settings > Virtual Reality SDKs
```

Figure 7: Teleport.

4.9 Physics and objects

The objects and the physics are influenced by the type of scene in which we find ourselves. When we find ourselves in a real scene, the objects will have a realistic behaviour and when you lift them and launch them they will be affected by realistic physics. On the contrary, when we are in the memory, the objects may have behaviours that are unrealistic to warn the player that is in a state of mind and not in reality. Some of these behaviours may be disappearing, anti-gravity, dematerialization, etc.

All of the physics of the game has been set using the component Rigidbody, which is a component used for the simulation of the physics. Most of their variables have been used to extract all the potential of this component. Thanks to variables such as isKinematic to anchor an object so that it does not move, useGravity that activates and deactivates the gravity, mass that allows us to choose the mass of the object, and many more variables have been implemented easily in different configurations for different behaviours of the objects. All of these changes have been made from scripts in order to facilitate the understanding of its operation and a better reading of the same.



Rigidbody

Mass	1
Drag	0
Angular Drag	0.05
Use Gravity	✓
Is Kinematic	☐
Interpolate	None
Collision Detection	Discrete
Constraints	
Freeze Position	☐ X ☐ Y ☐ Z
Freeze Rotation	✓ X ✓ Y ✓ Z

Figure 8: Rigidbody component.

5 Results

The obtained results are a completely finished and functional part of the game. All the objectives have been successfully achieved. In the first place, the result of the implementation of a fully functional character, including the follow-up features, grip objects, and the movement implemented in the two ways were expressed at the start of the work. In the second place, it has implemented a movement of adaptive grip in a satisfactory and functional way. Finally, all the puzzles proposed by the designer of levels were implemented and they all work according to the instructions.

The following links provide access to the project and to a video of the start of the game:

-Github: https://github.com/danicas92/Vornik

-Executable: https://drive.google.com/file/d/1Zm0iAGuxle1NxKVKEG410bIvxQBCgSZa/view

-Youtube video : https://www.youtube.com/watch?v=o0g3VlglNdA
-Youtube video 2: https://www.youtube.com/watch?v=FR-EVspA608
-Mechanics video: https://www.youtube.com/watch?v=45XQgF_ikxA&t=2s
-Trailer: https://www.youtube.com/watch?v=kN0BrxNzex4

6 Conclusions

Vornik is the largest project we have done and one of the most interesting challenges and self-learning for us. We have obtained an original product, with a fully exclusive mechanical, a dynamic grip, an immersive storyline and a spectacular design. I would also like to mention that we had no previous experience in the field of Virtual Reality and this has been a wonderful way to learn about this and all what you have to offer. The experience has been very beneficial both when working on a team and when working as an individual. Thanks to the constant communication, the development of the game has been very fluid.

7 Future work

The history of the video game Vornik continues to expand, and along with it will expand the field of the programming and the art will expand too. This way we have already decided that this game will continue until it is terminated in its entirety.

8 References

Unity User Manual. https://docs.unity3d.com/Manual/index.html?_ga=2.228110229.944004910.1557852500-122365420.1557063783

Oculus Developer Center Documentation. https://developer.oculus.com/documentation/unity/latest/

Courses documentation.

9 Bibliography

Unity Technologies. (2019). Unity User Manual. 2019, de Unity Sitio web: https://docs.unity3d.com/Manual/index.html

Unity Technologies. (2019). Unity Intermediate Couses. 2019, de Unity Sitio web: https://unity.com/learn/partner-courses

Unity Technologies. (2018). Oculus Unity . 2018, de Unity Sitio web: https://unity3d.com/es/partners/oculus

Jesse Glover, Jonathan Linowes. (2019). Complete Virtual Reality and Augmented Reality Development with Unity. Packt: Packt.

Jonathan Linowes. (2015). Unity Virtual Reality Projects: Explore the world of virtual reality by building immersive and fun VR projects using Unity 3D. Packt: Packt.

10 Appendix A – Additional documentation

All codes in C# language.

```
public class ActivatePictureWithCup : MonoBehaviour
{
  [SerializeField] private GameObject picture;
  [SerializeField] private string nameToCollide;

  private bool _colocado = false;

  private void OnTriggerEnter(Collider other)
  {
    if (other.name == nameToCollide && !_colocado)
    {
      var objectGrabb = other.GetComponent<ObjectGrabber>();
      objectGrabb.Throw(Vector3.zero,Vector3.zero,false);
      objectGrabb.transform.SetPositionAndRotation(transform.position,
transform.rotation);
      Destroy(objectGrabb);
      _colocado = true;
      ActivatePicture();
    }
  }

  private void ActivatePicture() { picture.SetActive(true); }

}
```

```
public class ActivationScript : MonoBehaviour
{

  [SerializeField] private GameObject ObjectInf;

  private ObjectGrabber _og;

  private void Awake()
  {
     _og = GetComponent<ObjectGrabber>();
  }

  private void Update()
  {
     if (_og.GetGrabbed())
     {
        ObjectInf.SetActive(true);
     }
  }
}
```

```
public class ButtonPressed : MonoBehaviour
```

```
{

  [SerializeField] private Transform falsoFondo;

  private bool _active = false;
  private float _augmento = 0;

  private void OnTriggerEnter(Collider other)
  {
     if (!_active && other.CompareTag("Index"))
     {
        _active = true;
     }
  }

  private void Update()
  {
     if (!_active) return;

     if (_augmento < 0.4f)
     {
        falsoFondo.Translate(new Vector3(0, -0.001f, 0));
        _augmento += 0.001f;
     }
  }
}
```

```
public class AgarreMano : MonoBehaviour
```

```
{
```

```
[SerializeField] private string InputName;
[SerializeField] private Transform pivot;
[SerializeField] private bool isRight;

private GameObject _currentObject;
private Vector3 _lastPosition;

public bool GetMano() => isRight;

 void Update()
 {
   if (_currentObject !=null && _currentObject.GetComponent<ObjectGrabber>() == null)
     _currentObject = null;

   if (Input.GetAxis(InputName) <= 0.15f && _currentObject != null )
   {
     _currentObject.GetComponent<ObjectGrabber>().Throw(transform.position, _lastPosition,true);
     _currentObject = null;
   }
   _lastPosition = transform.position;
 }

 public void OnTriggerStay(Collider colliderObject)
 {
   if (_currentObject != null || colliderObject.GetComponent<ObjectGrabber>() == null)
     return;


 if (colliderObject.CompareTag("Grab") && Input.GetAxis(InputName) >= 0.15f && _currentObject == null && ! colliderObject.GetComponent<ObjectGrabber>().GetGrabbed())
   {
     _currentObject = colliderObject.gameObject;
     GetComponentInChildren<ControllerCollidersGrab>().ResetGrab();
     _currentObject.GetComponent<ObjectGrabber>().Grab(pivot, isRight);
     _currentObject.GetComponent<Rigidbody>().useGravity = false;
   }
 }
}
```

```
public class BolaLaberinto : MonoBehaviour
```

```
{
  [SerializeField] private ObjectGrabber laberintoObjectGrabber;
  [SerializeField] private GameObject tapa;
  [SerializeField] private GameObject taza;

  private Rigidbody _rbBola;
  private Vector3 _positionInitial;
  private bool _haveRigidbody;
  private bool _open = false;
  private float _rotacion = 0;

  private void Awake()
  {
    _rbBola = GetComponent<Rigidbody>();
    _positionInitial = transform.localPosition;
  }

  void Update()
  {

    if (laberintoObjectGrabber.GetGrabbed() && !_haveRigidbody)
    {
      StartCoroutine(nameof(AddRigidbody));
      _haveRigidbody = true;
    }
    else if(!laberintoObjectGrabber.GetGrabbed())
    {
      Destroy(GetComponent<Rigidbody>());
      _haveRigidbody = false;
    }

    if (_open && _rotacion < 90)
    {
      if (!taza.activeSelf)
      {
        taza.SetActive(true);
        taza.transform.parent = null;
      }
      tapa.transform.Rotate(-1, 0, 0);
      _rotacion++;

    }
    else if (_rotacion >= 90)
    {
      Destroy(this.gameObject);
    }
  }
 
  IEnumerator AddRigidbody()
  {
    yield return new WaitForSeconds(0.3f);
    gameObject.AddComponent<Rigidbody>();
  }

  private void OnTriggerEnter(Collider other)
  {
    if (other.CompareTag("Ball"))
    {
      _open = true;
    }
  }

  private void OnTriggerExit(Collider other)
  {
    if (other.CompareTag("Laberinto"))
    {
      if (transform.localPosition != _positionInitial) transform.localPosition = _positionInitial;
    }
  }
}
```

```
public class ButtonInfo : MonoBehaviour
```

```
{
  [SerializeField] private bool _isRight;
  [SerializeField] private ControladorCandado controladorCandado;

  private void OnTriggerEnter(Collider other)
  {
    if (other.CompareTag("Index"))
    {
      controladorCandado.Rota(_isRight);
    }
  }
}
```

```
public class ColliderInfo : MonoBehaviour
```

```
{
  [SerializeField] private int colliderIndentificador;

  public int GetColliderIdentificador() { return colliderIndentificador; }
}
```

```
public class ComportamientoObjetos : MonoBehaviour
{
  private ObjectGrabber _scriptObjectGrabber;
  private Rigidbody _rb;
  private VLB_Samples.Rotater rotater;
  private bool _grabbed;
  private bool _firstInteraction = false;


  private void Awake()
  {
    _scriptObjectGrabber = GetComponent<ObjectGrabber>();
    rotater = GetComponent<VLB_Samples.Rotater>();
    _rb = GetComponent<Rigidbody>();
  }

  private void Update()
  {
    if (!_firstInteraction && rotater.enabled)
    {
      _firstInteraction = true;
    }

    _grabbed = _scriptObjectGrabber.GetGrabbed();
    if (_grabbed)
    {
      rotater.enabled = false;
    }
    else if(_firstInteraction)
    {
      rotater.enabled = true;
      _rb.isKinematic = true;
      _rb.useGravity = false;
    }
  }
}
```

```
public class ControladorCandado : MonoBehaviour
{
```

```
[SerializeField] private ControladorRueda controladorRueda;
[SerializeField] private int posCorrect;
[SerializeField] private ButtonInfo[] buttons;

private int _actualPos = 0;
private bool _imCorrect = false;

public bool GetCorrent() => _imCorrect;

 public void Rota(bool direccion)
 {
   _actualPos += direccion ? _actualPos==3? -3:1 : _actualPos == 0 ? 3 : -1;
   transform.Rotate(new Vector3(0,direccion? 90:-90,0));
   CheckISCorrect();
   controladorRueda.Check();
 }

 private void CheckISCorrect()
 {
   if (posCorrect == _actualPos)
     _imCorrect = true;
   else
     _imCorrect = false;
 }

 private void OnDisable()
 {
   foreach (var button in buttons)
   {
     button.enabled = false;
   }
 }
}
```

```
public class ControladorRueda : MonoBehaviour
{
```

```
[SerializeField] private ControladorCandado[] controladoresCandado;
[SerializeField] private GameObject Matrioshka;
[SerializeField] private GameObject Jarron;
[SerializeField] private AudioSource audio;

private int _contador;

public void Check()
 {
   audio.Play();
   foreach (var controlador in controladoresCandado)
   {
     if (controlador.GetCorrent()) _contador++;
   }
   if (_contador == 3)
   {
     transform.Rotate(new Vector3(0,0,-90));
     Matrioshka.tag = "Grab";
     Jarron.tag = "Grab";
     this.enabled = false;
   }
   _contador = 0;
 }

 private void OnDisable()
 {
   foreach (var controlador in controladoresCandado)
   {
     controlador.enabled = false;
   }
 }
}
```

```
public class ControllerCollidersGrab : MonoBehaviour
```

```
{
  private Collider[] colliders;
  private StopFinger[] stopFingers;

  private void Awake()
  {
    colliders = GetComponentsInChildren<Collider>();
    stopFingers = GetComponentsInChildren<StopFinger>();
  }

  public void ForcedThrow()
  {
    foreach (var fingerScript in stopFingers)
      fingerScript.PlayThisFinger();
  }

  private void OnEnable()
  {
    foreach (var collider in colliders)
      collider.enabled = true;
  }

  private void OnDisable()
  {
    foreach (var collider in colliders)
      collider.enabled = false;
    foreach (var finger in stopFingers)
      finger.Reset();
  }

  public void ResetGrab()
  {
    foreach (var finger in stopFingers)
      finger.ResetRotation();
  }
}
```

```
public class DesactivaPared : MonoBehaviour
```

```
{
    [SerializeField] private Transform puerta;
    [SerializeField] private GameObject fotoZledic;

    private bool _collide = false;

    private void OnTriggerEnter(Collider other)
    {
        var trozoTetera = other.GetComponentInChildren<TrozoTetera>();
        if (!_collide && trozoTetera != null && trozoTetera.GetHijos()==3)
        {
            DisableGrabbing(other.gameObject);
            _collide = true;
        }
    }

    private void DisableGrabbing(GameObject tetera)
    {
        tetera.GetComponent<ObjectGrabber>().Throw(Vector3.zero, Vector3.zero, false);
        tetera.transform.SetPositionAndRotation(transform.position,transform.rotation);
        Destroy(tetera.GetComponent<ObjectGrabber>());
        puerta.Rotate(0, 0, -82);
        fotoZledic.SetActive(true);
    }

}
```

```
public class JarronScript : MonoBehaviour
{
 [SerializeField] private GameObject jarronTrozos;

 private Rigidbody _rb;
 private ObjectGrabber _objectGrabber;
 private bool _destroyed = false;

 private void Awake()
 {
    _rb = GetComponent<Rigidbody>();
    _objectGrabber = GetComponent<ObjectGrabber>();
 }

 private void OnCollisionEnter(Collision collision)
 {
    if (!_destroyed && !_objectGrabber.GetGrabbed() && GoodVelocity() > 0.3f)
    {
      _destroyed = true;
      Instantiate(jarronTrozos,transform.position,transform.rotation);
      Destroy(gameObject);
    }
 }

 private float GoodVelocity()
 {
    var aux = _rb.velocity.x + _rb.velocity.y + _rb.velocity.z;
    return aux;
 }
}
```

```
public class Linterna : MonoBehaviour
{
    [SerializeField] private Transform pivote;
    [SerializeField] private GameObject light;
    [SerializeField] private GameObject marker;

   private ObjectGrabber _objectGrabber;
   private bool _active = false;
   private string _buttDer = "Oculus_CrossPlatform_Button2";
   private string _buttIzq = "Oculus_CrossPlatform_Button4";

    private void Awake()
    {
      _objectGrabber = GetComponent<ObjectGrabber>();
    }

    private void Update()
    {
      if (_objectGrabber.GetGrabbed())
      {
        marker.SetActive (false);
        var hand = _objectGrabber.GetHandGrabbing;

 if ((Input.GetButtonDown(_buttDer) && hand.GetComponent<AgarreMano>().GetMano())
|| (Input.GetButtonDown(_buttIzq) && !hand.GetComponent<AgarreMano>().GetMano()))
        {
          _active = !_active;
          light.SetActive(_active);
        }
        return;
      }
      marker.SetActive(true);


      if (transform.position != pivote.position)
      {
        transform.position = pivote.position;
        transform.rotation = Quaternion.Euler(90,0,0);
        _active = false;
        light.SetActive(false);
      }
    }
}
```

```
public class LlaveCajaMusica : MonoBehaviour
{
    [SerializeField] private Transform pivoteInsert;
    [SerializeField] private Vector3 rotation = new Vector3(0, 0, 270);
    [SerializeField] private Transform tapa;
    [SerializeField] private GameObject taza;
    [SerializeField] private GameObject UI;
    [SerializeField] private AudioSource audioSource;

    private Vector3 _posInit;
    private Quaternion _rotInic;
    private bool _rotate = false;
    private bool _activate = false;
    private float _keyRotation = 0;
    private bool _open;
    private float _tapRotation = 0;

    private void Awake()
    {
        _posInit = transform.position;
        _rotInic = transform.rotation;
        audioSource = GetComponent<AudioSource>();
    }

    private void Update()
    {
        if (!_activate && GetComponent<ObjectGrabber>().GetGrabbed())
            _activate = true;

        if (_activate && !_rotate && !_open && transform.position != _posInit && !
GetComponent<ObjectGrabber>().GetGrabbed())
        {
            transform.SetPositionAndRotation(_posInit, _rotInic);
            GetComponent<Rigidbody>().velocity = Vector3.zero;
        }

        if (_rotate && _keyRotation < 360)
        {
            transform.Rotate(0, 2, 0);
            _keyRotation += 2;
            _open = _keyRotation == 360? true : false;
            taza.SetActive(true);
            Destroy(UI);
        }

        if (_open && _tapRotation < 90)
        {
            tapa.Rotate(-1, 0, 0);
            _tapRotation++;
        }

    }

    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("MusicBox") && !_rotate)
        {
            GetComponent<ObjectGrabber>().Throw(this.transform.position,this.transform.position, false);
            GetComponent<Rigidbody>().isKinematic = true;
            transform.parent = pivoteInsert;
            transform.localPosition = Vector3.zero;
            transform.localRotation = Quaternion.Euler(rotation);
            Destroy(GetComponent<ObjectGrabber>());
            _rotate = true;
            audioSource.Play();
            //
        }
    }
}
```

```
public class Monoculo : MonoBehaviour
{
  [SerializeField] private Camera camera;

  private ObjectGrabber _objectGrabber;

  private void Awake()
  {
    _objectGrabber = GetComponent<ObjectGrabber>();
  }


  private void Update()
  {

    if (_objectGrabber.GetGrabbed() && !camera.enabled)
    {
      camera.enabled = true;
      return;
    }
    else if(!_objectGrabber.GetGrabbed())
    {
      camera.enabled = false;
    }

  }
}

```

```
public class ObjectGrabber : MonoBehaviour
{
    [SerializeField] private Vector3 rotacionAgarreDer;
    [SerializeField] private Vector3 rotacionAgarreIzq;

    private bool _isGrabbed;
    private Quaternion _pivotRotationInic;
    private Vector3 _lastPosition;
    private GameObject handGrabbing;

    private float MultiplyVeloc = 1.25f;//Multiplicador de la velocidad de lanzamiento

    public GameObject GetHandGrabbing => handGrabbing;

    public void Update()
    {
        _lastPosition = transform.position;
    }

    public bool GetGrabbed() { return _isGrabbed; }

    public void Grab(Transform pivot, bool derecha)
    {
        handGrabbing = pivot.parent.gameObject;
        transform.parent = pivot;
        transform.SetPositionAndRotation(pivot.position,pivot.rotation);
        var rot = derecha ? rotacionAgarreDer : rotacionAgarreIzq;
        //Debug.Log(rot);
        transform.Rotate(rot);
        Rigidbody rb = GetComponent<Rigidbody>();
        rb.useGravity = false;
        rb.isKinematic = true;
        SetCollidersTrigger(true);
        _isGrabbed = true;
    }

    public void Throw(Vector3 hand, Vector3 handLastPosition, bool activeRigidBody)//Se usa la pos de la mano para dejar el objeto ahy, la ultima pos de la mano para calcular la vel y la rotación para dejarlo con la rotaión actual de la mano
    {
        _isGrabbed = false;
        handGrabbing.GetComponentInChildren<ControllerCollidersGrab>().ForcedThrow();
        handGrabbing = null;
        Rigidbody rb = GetComponent<Rigidbody>();
        if (activeRigidBody)
        {
          rb.useGravity = true;
          rb.isKinematic = false;
          Vector3 CurrentVelocity = (hand - handLastPosition) / Time.deltaTime;
          rb.velocity = CurrentVelocity * MultiplyVeloc;
        }
        SetCollidersTrigger(false);
        transform.parent = null;
    }

    private void SetCollidersTrigger(bool set)
    {
        var colliders = GetComponents<Collider>();
        var colliderChildren = GetComponentsInChildren<Collider>();
        foreach (var collider in colliders)
          collider.isTrigger = set;

    }

  }
```

```
public class StopFinger : MonoBehaviour {

  [SerializeField] private Transform indexAnt;
  [SerializeField] private string triggerButton;
  [SerializeField] private List<StopFinger> fingersAnt;
  [SerializeField] private bool _isThumb;

  private float _multiplyer;
  private float _multiplyerAnt = 0;
  private float _maxRotacion;
  private Vector3 _rotInic;
  private bool _collisionDetected;
  private bool _otherCollided;

  private float _maxThumb = -0.4f;

  private void Awake()
  {
    _rotInic = indexAnt.transform.localRotation.eulerAngles;
  }

  private bool CheckThumb()
  {

 if (indexAnt.transform.localRotation.eulerAngles.z < _rotInic.z + 0.1f && indexA
nt.transform.localRotation.eulerAngles.z > _rotInic.z - 0.1f)
      return false;
    return true;
  }

  private void Update()
  {
    _multiplyer = Input.GetAxis(triggerButton);
    if (_isThumb)
    {
      if (_multiplyer == 1 && !_collisionDetected && !
_otherCollided && _multiplyerAnt < _multiplyer)//Avanza
      {
        indexAnt.Rotate(0, 0, -0.05f * 50);
        _multiplyerAnt += 0.05f;
      }
      else if(_multiplyer == 0 && _multiplyerAnt> _multiplyer)
      {
        indexAnt.Rotate(0, 0, 0.05f * 50);
        _multiplyerAnt -= 0.05f;
      }
    }
    else
    {
      if (_multiplyer > _multiplyerAnt && _maxRotacion == 0)//Avanza
      {
        if (!_collisionDetected && !_otherCollided)
        {
          if (_multiplyer - _multiplyerAnt < 0.05f) return;
          indexAnt.Rotate(0, 0, -0.05f * 50);
          _multiplyerAnt += 0.05f;
        }
      }
      else if (_multiplyer < _multiplyerAnt)//Retrocede
      {
        if ((_maxRotacion == 0 || _multiplyer < _maxRotacion) /*&&
indexAnt.transform.localRotation.z < rotInic.normalized.z*/)
        {
          if (_multiplyer - _multiplyerAnt > -0.05f) return;

          indexAnt.Rotate(0, 0, 0.05f * 50);
          _multiplyerAnt -= 0.05f;
        }
      }
      else if (_multiplyer == 0)
      {
        indexAnt.localRotation = Quaternion.Euler(_rotInic);
      }
    }
  }

  private void OnTriggerEnter(Collider other)
  {
    if (other.gameObject.CompareTag("CollisionFingers"))
    {
      _collisionDetected = true;
      _maxRotacion = indexAnt.transform.localRotation.z;
      StopLastFingers();
    }
  }

  private void OnTriggerExit(Collider other)
  {
    if (other.gameObject.CompareTag("CollisionFingers"))
    {
      _collisionDetected = false;
      _maxRotacion = 0;
      if (_isThumb) _maxThumb = -0.4f;
      PlayLastFingers();
    }
  }

  private void StopLastFingers()
  {
    foreach (var finger in fingersAnt)
    {
      finger.StopThisFinger();
    }
  }

  private void PlayLastFingers()
  {
    foreach (var finger in fingersAnt)
    {
      finger.PlayThisFinger();
    }
  }

  public void PlayThisFinger()
  {
    _otherCollided = false;
    _collisionDetected = false;
  }

  public void StopThisFinger()
  {
    _otherCollided = true;
  }

  public void Reset()
  {
    _multiplyerAnt = 0;
    _maxRotacion = 0;
    _maxThumb = -0.4f;
    _collisionDetected = false;
  }

  public void ResetRotation()
  {
    indexAnt.localRotation = Quaternion.Euler(_rotInic);
    Reset();
  }
}
```

```
public class TrozoTetera : MonoBehaviour
{
    public EventsSystem eventSystem;

    [SerializeField] private int identificador;

    private int _hijos = 0;
    private bool _todoUnido;
    public int GetHijos() => _hijos;
    public void SetHijos(int hijosSum) => _hijos += hijosSum;
    public int GetIdentificador() { return identificador; }

    private void OnTriggerEnter(Collider other)
    {

        if (other.CompareTag("TeapotPiece") && other.GetComponent<ColliderInfo>().GetColliderIdentificador() == identificador)
        {

            if (other.GetComponentInParent<TrozoTetera>().GetIdentificador() > identificador) return;

            if (!GetComponent<ObjectGrabber>().GetGrabbed() || !other.GetComponentInParent<ObjectGrabber>().GetGrabbed()) return;

            other.gameObject.GetComponentInParent<TrozoTetera>().SetHijos(_hijos+1);
            GetComponent<ObjectGrabber>().Throw(transform.position,transform.position,false);
            Destroy(GetComponent<ObjectGrabber>());
            Destroy(GetComponent<Rigidbody>());
            Destroy(GetComponent<VLB_Samples.Rotater>());
            Destroy(GetComponent<ComportamientoObjetos>());
            transform.parent = other.transform;
            transform.localPosition = Vector3.zero;
            transform.localRotation = Quaternion.Euler(Vector3.zero);

            RemoveCollidersHijos();
            RemoveCollsionWithFingers();
        }

        if (identificador == 1 && _hijos == 3 && !_todoUnido)
        {
            Destroy(GetComponent<VLB_Samples.Rotater>());
            Destroy(GetComponent<ComportamientoObjetos>());
            eventSystem.DesactivateToys();
            _todoUnido = true;
    }

}

private void RemoveCollidersHijos()
{
  var colliders = GetComponentsInChildren<Collider>();
  foreach (var coll in colliders)
    Destroy(coll);
}

private void RemoveCollsionWithFingers()
{
  var fingersCollision = GetComponentsInChildren<Transform>();
  foreach (var collisionGO in fingersCollision)
  {
    if (collisionGO.CompareTag("CollisionFingers"))
    {
      collisionGO.tag = "Untagged";
    }
  }
}
}
```

```
public class UIObjectScript : MonoBehaviour
{
 [SerializeField] private GameObject UI;

 private ObjectGrabber _objectGrabber;

 private void Awake()
 {
   _objectGrabber = GetComponentInParent<ObjectGrabber>();
 }

 private void Update()
 {
   if (_objectGrabber.GetGrabbed() )
   {
     Destroy(transform.gameObject);

   }
 }


}
```