Escuela Politécnica Superior

Development of a video game with Unreal Engine 4

Degree in Multimedia Engineering

End of Degree Project

Author:

Jose Maria Egea Canals

Guardian(s):

Miguel Angel Lozano Ortega

Universitat d'Alacant
Universidad de Alicante

September 2015

Justification and Objectives

Given the skills learned during the Degree in Multimedia Engineering that allow us to cover various aspects in terms of multimedia content, it is intended to develop a 3D video game that puts these capabilities to the test.

For the construction of the video game, it is intended to use the Unreal Engine 4 engine. In addition, the The project includes both the Video Game Design Document (GDD) and the elaboration of the same.

The game will be made using Unreal's Blueprints visual scripting system. Engine 4. In addition, it will be compatible with the Oculus Rift DK1 peripheral.

For the construction of the game scenarios and menus, editor topics will be covered. terrain, particle effects, lighting, sound, among others.

Index of contents

1. Introduction

As the years go by, people develop new forms of entertainment, because the need for leisure is part of human nature itself. Among these forms we find video games, which since the beginning of their history back in the 1940s[1], have been growing substantially both in followers, as in genres, ways of playing them, number of development teams, and in technology, coming to bill for several years more than the music and cinema together in Spain, and each year increasing[2].

Currently, in addition, given the large number of technologies to develop video games, the number of platforms, and the increasing assimilation of technology by all people due to its decreasing cost, it is possible to find in the video game market from projects with multi-million dollar costs, such as the case of Destiny, whose cost reached €380 million[3], with expert development teams in many disciplines and working hard for several years, to zero-cost projects that can be done by anyone in their home and that can accumulate millions, as in the case of Flappy Bird for Android, which was created by a person in his house for several days, and that each day he earned the creator, Dong Nguyen, a few US$90,000[4].

With all this, it can be seen that the world of video games can be a bet interesting for any multidisciplinary developer, or small development team, and We speak of multidisciplinary because a video game project includes work from programming, sound, design, and also artistic 2D and increasingly, 3D. It is So, this is where the issue comes in. As multimedia engineers, we must be able to handle the development of a project of these characteristics, especially if we have carried out the specialization of creation and digital entertainment, where aspects of Reality are even treated Virtual[5].

For this Final Degree Project, we are going to check if it is possible to cover a project multidisciplinary such as that of a 3D video game, and for this we will use the video games Unreal Engine 4. In addition, we want this project, given the itinerary that has been made, offer compatibility with Oculus Rift DK1 peripheral, virtual reality glasses which the university currently has available to students. We will also create Game Design Document (GDD) of the video game, to describe all the design of the same to be carried out posteriorly.

2. Theoretical framework or State of the art

This block is intended to make an analysis of video game engines in their context, make a comparison between the most popular engines that exist right now, and also, highlight the utilities and features offered by Unreal Engine 4, which have led to the choice of this tool for the creation of the project that we are going to document.

It is necessary to emphasize that it is with the video game engine that we are going to build the game, while modeling and animation tools, audio editing programs, and others for editing graphics, these only serve us for the production of Assets (resources), which through the engine we will use. It is for this reason that in the theoretical framework it has been decided to include As a theme, video game engines, and the video game itself.

2.1. Video game concept

We are going to try to first establish a definition of a video game, so that we can better understand what we are dealing with with this work.

Video games, although complex, are based on something simple. Bernard Suits wrote that “playing a video game is a voluntary effort to overcome unnecessary obstacles”[6]. In turn, Wikipedia gives us a definition, saying that a video game “is an electronic game in which one or more people interact, by means of a controller, with a device equipped with video images”[7].

Taking into account that a game consists of an activity that:

- Requires at least one player.
- It has rules.
- Has a win condition.

The most accurate probably with the definition of a video game is "a game that is played through a video screen".

2.2. Engines to produce video games

2.2.1. Engines in context

Before starting, it is worth defining the following, what is a video game engine?

The term video game engine was born in the mid-1990s, referring to the early first-person shooter (FPS) games, especially the well-known Doom, by id Software. Doom was designed with a very well defined architecture, with an excellent separation between core engine components (such as the rendering system

three-dimensional, collision detection, or the audio system), the art resources of the game, the game worlds, and the rules for playing them.

The value of this separation becomes apparent when developers start creating games that started from that basic structure, but with different art, worlds, weapons, vehicles, and other assets (resources), in addition to the rules of the game, all this only making minimal changes to the existing “engine”. This marked the birth of the "mod" community, a group of individual gamers and small independent studios that built their video games modifying existing games, using free toolkits provided by the original developers. And it is that thanks to video game engines, designers no longer depend on designers, and it is also possible to add or change parts of the game quickly, which before its invention, could be costly[8].

Starting from the id software engine, other companies decided to build their own videogames, as Epic Games decided to do with the Unreal Engine in 1998[9], or Valve with its Source engine in 2004[10], and other later companies.

2.2.2. Comparison of video game engines on the market

Currently there is a very wide list of engines to produce video games, both 2D and 3D, paid and free, and that offer greater or lesser compatibility with platforms for both develop as for those who produce video games.

Naming them all would be complicated, given the amount, Wikipedia makes a list with a large quantity of them classified by license:

V·T·E	Game engines (list)	[hide]
Source port · First-person shooter engine (list) · Tile engine · Game engine recreation (list) · Game creation system		
Free software / open source	2D	Adventure Game Studio · Beats of Rage · Box2D · Chipmunk · Cocos2d · Digital Novel Markup Language · Flixel · Exult · Game-Maker · Gosu · Jogre · KiriKiri · Moai SDK · ORX · Pygame · Ren'Py · StepMania · Stratagus · Thousand Parsec · VASSAL · Xconq
	2.5D	Aleph One · Build · Flexible Isometric Free · Id Tech 1 · Wolfenstein 3D
	3D	Away3D · Axiom · Blender Game · Cafe · Crystal Space · Cube · Cube 2 · Delta3D · Dim3 · Genesis3D · GLScene · Horde3D · HPL 1 · Irrlicht · Id Tech 2 · id Tech 3 (loquake3) · id Tech 4 · JMonkey · Luxinia · OGRE · Ogre4j · Open Wonderland · Panda3D · Papervision3D · Platinum Arts Sandbox Free 3D Game Maker · PlayCanvas · PLIB · Python-Ogre · Quake · Nebula Device · RealmForge · Retribution · Torque 3D
	Mix	Allegro · Construct Classic · Godot · Lightweight Java Game Library · Spring · Visualization Library
Proprietary	2D	Clickteam Fusion · Coldstone · Construct 2 · Corona · CRX · Fighter Maker · Filmation · GameMaker · GameMaker: Studio · Garry Kitchen's GameMaker · Generic Tile · Gold Box · MADE · Mscape · M.U.G.E.N · NScripter · RPG Maker · Shoot the Bullet · Sim RPG Maker · Sound Novel Tsukūru · Southpaw · Stencyl · Vicious · Virtual Theatre · V-Play · Z-machine · Zillions of Games · ZZT
	2.5D	Genie · INSANE · Infinity · Jedi · Pie in the Sky · Super Scaler · UbiArt Framework
	3D	4A · Advance Guard Game · Anvil/Scimitar · Arsys · Beelzebub · Bork3D · BRender · C4 · Chrome · Creation · CryEngine · Crystal Tools · Dagor · Diesel · Digital Molecular Matter · Disrupt · Dunia · EAGL · EGO · Electron · Elflight · Enforce · Enigma · Essence · Flare3D · Fox · Freescape · Frostbite · Geo-Mod · GoldSrc · HeroEngine · HydroEngine · HPL 2 · id Tech 5 · id Tech 6 · Ignite · Iron · IW · Jade · Kinetica · LS3D · Leadwerks · LithTech · Luminous Studio · LyN · Marmalade · Mizuchi · MT Framework · NanoFX GE · Odyssey · Orochi · Outerra · Panta Rhei · Phoenix Engine (Relic) · Phoenix Engine (Wolfire) · PhyreEngine · Q · Real Virtuality · REDengine · Refractor · RenderWare · Revolution3D · Riot · RAGE · SAGE · Serious · Shark 3D · Silent Storm · Sith · Source · SunBurn XNA · Titan · TOSHI · Truevision3D · Unigine · Unity · Unreal · Vengeance · Visual3D · Voxel Space · XnGine · X-Ray · Yebis · YETI · Zero
	Mix	CPAGE · Dark · Gamebryo · Hybrid Graphics · Kaneva Game Platform · Metismo
Proprietary Game middleware (list)	AiLive · Euphoria · Gameware · GameWorks · Havok · iMUSE · Kynapse · Quazal · SpeedTree · Xaitment	

Figure 1. List of engines

Fuente: Wikipedia[11]

We are going to highlight and compare the 3 most famous engines for the production of video games nowadays. These are Unreal Engine 4 from Epic Games, Unity from Unity Technologies, y CryEngine de Crytek[15].

2.2.2.1. Unreal Engine 4.

Figure 2. Unreal Engine logo

Source: Wikimedia[12]

Unreal Engine 4, UE4 for short, has excellent graphics capabilities, including but not limited to other aspects, advanced dynamic lighting capabilities, and a particle system that can handle up to a million particles in a scene at once.

Even though Unreal Engine 4 is the successor to UDK, the version change between UDK and UE4 is really noticeable. Changes made with the goal of improving the ease of video game production.

Among the changes to be highlighted, one of the main ones is the scripting language for the UE4, which in UDK it was the UnrealScript language, and now it has been completely replaced by C++ in UE4. Additionally, UE4 now uses Blueprints for graphical scripting, an advanced version of Kismet with which a video game can be completely made without the need to program in C++.

Unreal Engine 4 has a license to charge 5% of the earnings of a video game from the first 3,000 US dollars for each semester.

2.2.2.2. Unity.

Figure 3. Unity logo

Source: Wikimedia[13]

The Unity engine offers a wide range of features and has a simple user interface.
to understand. In addition, it has a multiplatform integration system that allows you to export games
for almost all existing ones, currently being the best option for 3D development in
Android platforms, for its compression tools that allow video games not to be particularly heavy,
and not to consume excessive resources.

The game engine is compatible with the main 3D modeling and animation applications, such as
3ds Max, Maya, Softimage, Cinema 4D, and Blender, among others, which means that it supports
the reading of files exported with these programs without offering no problem.

As against, Unity is a payment engine, where despite unlocking from version 5.0 almost
all the features that only the pro version had before, in this aspect, Unreal Engine 4 comes out
advantaged by being free and offering anyone who downloads it, 100% of all
engine functionalities.

2.2.2.3 CryEngine.

Figure 4. CryEngine logo

Source: Wikimedia[14]

CryENGINE is a fairly powerful engine designed by the Crytek company, which was introduced to
the gaming world with FarCry.

This engine has been designed to be used on home consoles and PCs, including the current
generation from Sony and Microsoft, i.e. PlayStation 4 and Xbox One.

The graphics capabilities of CryENGINE can be matched to those of Unreal Engine 4, with excellent lighting, realistic physics, advanced animation systems, etc. Furthermore, equally Like UE4, the engine has very powerful editor and level design features.

On the other hand, even though CryEngine is a really powerful engine, its learning curve it's a bit tricky to start using the game engine productively. If I dont know intends to produce a game with triple A level visual characteristics (normally multi-million dollar productions), its use is not usually recommended.

CryEngine has a subscription license that costs €/$9.90 per month. For large projects scope, it is necessary to contact them to obtain a license that allows access to 100% of the engine code and direct support from Crytek.

2.2.3. choice of engine

If we ask ourselves the question at the developer level of which engine to choose, we are faced with a decision to consider, since it has an important learning curve and it is also It is necessary to become familiar with numerous tools. This is why it is recommended to choose the one that best suits our needs, however, it is convenient to take into account a series of details[16]:

- Provide good documentation.
- That it has a good community of users that has not been abandoned.
- Take into account if we want to modify the engine or not.

For this project we have chosen to use the Unreal Engine 4 which, in addition to offering the 100% of its features for free, allow us to easily obtain some results considerably good, and to have a current community of users despite having come out in 2014, which currently allows practically any doubt that may arise to be resolved, incorporates the Blueprints technology, which allows us to theoretically develop a complete video game by 100% without the need to program, and on which we are going to base the entire development.

2.2.4. Architecture concepts of Unreal Engine 4 and its scripting programming system Blueprints

Next we will make a brief introduction to the characteristics of the engine core, and to introduce some concepts of architecture[17]. Later, you will make a introduction to Unreal Engine 4's graphical scripting system, Blueprints[19].

2.2.4.1. UObjects and Actors

Actors are instances of classes that derive from class AActor; the base class of all objects that can be positioned in the game world.

Objects are instances of classes that inherit from the UObject class; the base class for all objects in Unreal Engine, including actors. So, actually, all instances in Unreal Engine are Objects; Despite this, the term Actor is commonly used to refer to to instances of classes that derive from AActor in their hierarchy, while the term Object is used to refer to class instances that do not inherit from the AActor class. The majority of the classes that we create will inherit from AActor at some point in their hierarchy.

In general, actors can be interpreted as items themselves or as entities, while Objects are more specialized parts. Actors often make use of Components, which are specialized Objects, to define certain aspects of their functionality or store values of a collection of properties. Taking a car as an example, the car is the Actor itself, while the parts of the car, like the wheels and doors, are probably components of that Actor.

2.2.4.2. Gameplay and Framework Classes

The basic gameplay classes include functionality to represent players, allies, and enemies, as well as control of these avatars with player input or with an intelligence artificial. There are also classes for creating HUDs (heads-up displays) and cameras for players. Finally, gameplay classes such as GameMode, GameState, and PlayerState establish the rules of the game, and allow you to track how the game and the players

they progress.

All these classes are types of Actors, which it is possible to position in the level through the editor, or spawn (make them appear) of them when necessary.

2.2.4.3. Representing players, friends, and enemies in the world

2.2.4.3.1. Pawn

A Pawn is an Actor that can be an "agent" within the world. Pawns can be possessed by a Controller, since they are created to accept input easily, and can perform each one's own actions. Keep in mind that a Pawn does not have to be humanoid.

2.2.4.3.2. Character

A Character is a humanoid-style Pawn. It comes with a CapsuleComponent for collision, and a default CharacterMovementComponent. You can perform basic human actions like move, has network functionality, and has some animation functionality.

2.2.4.4. Control Pawns with player input or with Artificial Intelligence.

2.2.4.4.1 Controller

A Controller is an Actor that can be responsible for directing a Pawn. typically comes from two forms, AIController and PlayerController. A Controller can “own” a Pawn and have the control over him.

2.2.4.4.2. PlayerController

A PlayerController is an interface between the Pawn and the human player controlling it. The The PlayerController essentially does basically what the player tells it to do.

2.2.4.4.3 AIController

An AIController is an artificial intelligence that can control a Pawn.

2.2.4.5. Showing player information

2.2.4.5.1. HUD

A HUD is a "heads-up display", or 2D screen that is common in many games. in it can display health, ammo, etc. Each PlayerController normally has a HUD.

2.2.4.5.2. Camera

The PlayerCameraManager is the eyes and manages how it behaves. Each PlayerController typically has one of them.

2.2.4.6. Establishing the follow-up and the rules of the game.

2.2.4.6.1. GameMode

The concept of "game" is divided into two classes. The GameMode class is the definition of the game, which includes aspects such as its rules and victory conditions. It only exists on the server. Nope should have a lot of data that changes during the game.

2.2.4.6.2. GameState

The GameState class contains the state of the game, which could include things like the list of connected players, the score. It's what the pieces are in chess, or the quest list that must be completed in an open world game. GameState exists on the server and on all clients, and can freely replicate information to keep all machines updated.

2.2.4.6.3. PlayerState

PlayerState is the state of a participant in a game, such as a human player or a bot playing simulate a player. AI that exists as part of the game should not have PlayerState. What Example data that would be appropriate in a PlayerState includes the player's name, their score, or if you are carrying a flag in a game of capture the flag. there is a class PlayerState for each of the players that exist on each of the machines, and can freely replicate over the network to keep everything in sync.

2.2.4.7. Relationships of the framework classes

The following diagram illustrates how all of these core game classes relate to each other. A game is made up of a GameMode and a GameState. Human players who join the game are associated with PlayerControllers. These PlayerController allow players own Pawns in the game, and thus be able to have physical representation in the level. The PlayerControllers they also provide players with input controls, a HUD, and a PlayerCameraManager to manage camera views.

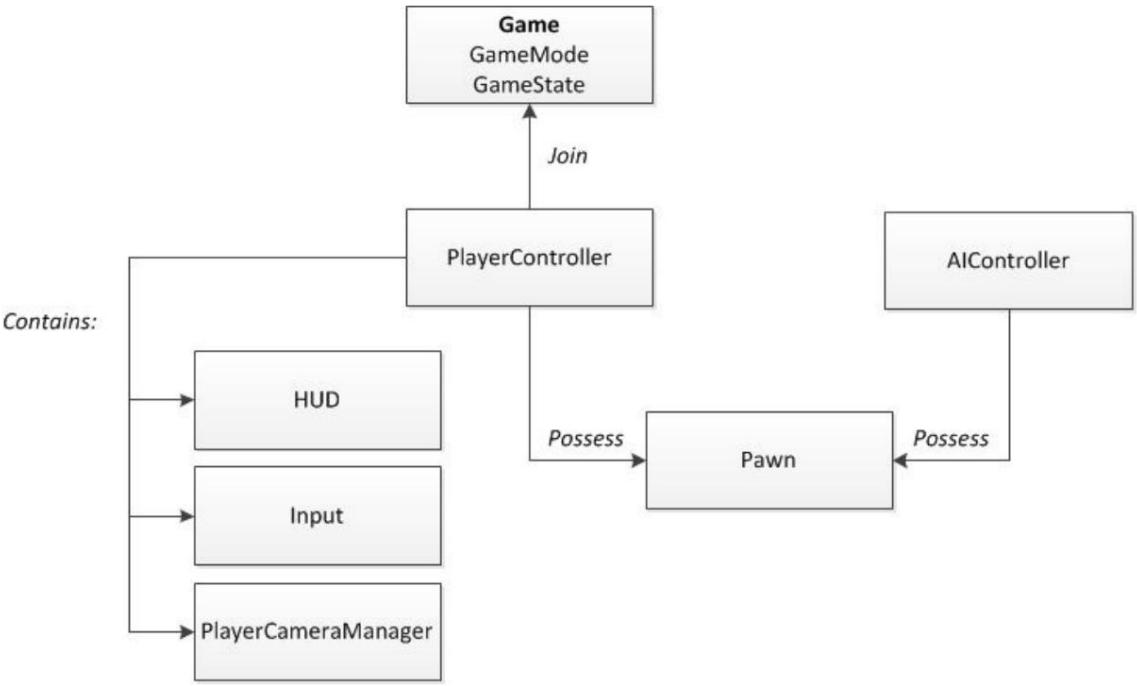


Figure 5. Class relationship diagram in Unreal Engine.

Font. Unreal Engine website[18]

2.2.4.8. Blueprints Visual Scripting

The Visual Scripting of Blueprints in Unreal Engine is a complete gameplay system implemented with the concept of a node-based interface to create gameplay elements through the Unreal Editor. This system is very flexible and powerful since provides the ability for designers to use virtually the full range of concepts and tools previously only available to programmers.

Through the use of Blueprints, designers can prototype, implement, or modify virtually any gameplay element, such as:

- Games. The game rules, victory conditions, etc.
- Players. Modify and create meshes and materials or customize characters.
- Cameras. Prototype camera perspectives or change the camera dynamically during the game.
- Input (Inputs). Change player controls or allow players to pass their controls to items.
- Items. Weapons, spells, triggers, etc.
- Environments. Create random assets or procedurally generate items.

2.2.4.8.1. Blueprints in general terms

Blueprints are special assets that provide an intuitive node-based interface that
can be used to create new types of actors and script level events,
giving gameplay designers and programmers the tools to create and iterate
gameplay quickly through the Unreal editor without the need to write code.

2.2.4.8.2. Types of Blueprints

Blueprints can be of various types, each type having its own functionality, we are going to
define the most important ones that we intend to use in the project.

2.2.4.8.2.1. Clase Blueprint

A Blueprint class, usually abbreviated as Blueprint, is an asset that allows
content creators quickly add functionality to existing game classes. The
Blueprints are created within the Unreal Editor visually, rather than by typing
code, and are saved as assets in a content pack. They basically define
new classes or types of actors that can then be positioned on maps as instances
that behave like any other kind of Actor.

2.2.4.8.2.2. LevelBlueprint

The LevelBlueprint is a specialized type of Blueprint that acts as the global node graph.
Of the level. Each level in a project has its own Level Blueprint created by default that can be
be edited through the unreal editor. It is not possible to create new Level Blueprints through the
editor de interfaces.

Events pertaining to the level itself, or specific instances of Actors throughout the level,
are used to trigger sequences of actions in the form of calls to functions or operations
of control flow. It's basically the same concept that Kismet was used for in UDK.

The LevelBlueprint also provides a control mechanism for streaming and Matinee as well
as to communicate with events of the actors located within the level.

2.2.4.8.2.3. Blueprint Interface

A Blueprint Interface is a collection of one or more functions that can be added to other
Blueprints. Any Blueprint that has the interface should have those features.

The functions of the interface can give functionality in each of the Blueprints that have them.
added. This is like the concept of interface in programming, which allows different types

share objects and access variables from a common interface. For simplicity, a Blueprint Interface allows different Blueprints to share and send data to each other.

Blueprint Interfaces can be made by content creators through the editor of a similar way to other Blueprints, but they come with some limitations that it is not possible to do in them:

- Add new variables.
- Edit graphs.
- Add components.

2.2.4.8.3. Anatomy of a Blueprint

The functionality of the Blueprints is defined by several elements, some of which are presented by default, while others can be added depending on our needs. This gives us the ability to define components, improve initialization, and initialization operations, respond to events, organize and modularize operations, define properties etc

2.2.4.8.3.1. Components Window

With the understanding of what Components are, the component window within the Blueprint Editor allows us to add new to our Blueprint. This gives us ways to add collision geometry such as CapsuleComponents, BoxComponents, or SphereComponents, add geometry in the form of a StaticMeshComponent or SkeletalMeshComponent, control movement using MovementComponents, etc. Components added in the component list can also be assigned to instance variables providing access to them in your Blueprint graph or in other Blueprints.

2.2.4.8.3.2. Construction Script

The Construction Script kicks in following the list of components when a instance of a Blueprint class is created. Contains a graph node that is executed allowing the instance of the Blueprint class to perform initialization operations. Is feature is useful for modifying mesh and material properties, or painting strokes in the world. For example, a Blueprint with a light could determine what type of terrain there is located and choose the correct mesh to use from a set of meshes.

2.2.4.8.3.3. Event Graph

The EventGraph of a Blueprint contains a graph of nodes that uses events and function calls to perform actions in response to gameplay events associated with the Blueprint. The EventGraph is used to add functionality that is common to all Blueprint instances. Furthermore, it is where interactivity and dynamic responses are established. For example, a light Blueprint could respond to a damage event by shutting down, and this behavior would be all the lights that we have created from that type of blueprint, so that as their corresponding damage event jumps, they go off.

2.2.4.8.3.4. Functions

Functions are nodes of the graphs that own a particular Blueprint, and can be executed or called from another graph through the Blueprint. Functions have a single point designated by a node with the same name as the function containing a single execution pin exit. When a function is called from another graph, the output execution pin is activated, which makes it possible for us to proceed with the scripting logic.

To explain it more clearly, if we have a function, we will have a single input and output pin. output, the input pin is the signal for the function to be activated, while the output pin is a signal for another component to be executed, in this way, we can create execution chains easily.

2.2.4.8.3.5. Variables

Variables are properties that store values or references to Objects or Actors in the world. We will be able to access these properties internally through the Blueprint that contains, or they can be published to be able to access them and modify their values from other Blueprints.

3. Objectives

3.1. Main objective of the Final Degree Project.

The objective of this project is the realization of a 3D videogame for PC using the Unreal Engine 4 video games, and focusing its entire development on the Visual Blueprints Scripting system.

With this objective, it is intended to carry out a previous design of the video game, analyze the characteristics offered by the engine, and learn both its operation and the necessary pipeline for the inclusion of Assets in it.

3.2. Breakdown of Objectives.

Below are the objectives of the TFG in the form of more specific tasks:

1. Make the GDD (Game Design Document) of a video game.
2. Cover the aspects of the pipeline necessary for the import of Assets in the engine from Autodesk 3ds Max.
3. Creation of a Menus and HUD system.
4. Create a game world using the tools provided by Unreal Editor.
5. Implement the entire game logic using Blueprints.
6. Make the game compatible with the Oculus Rift DK1 virtual reality device.
7. Sound the video game.

4. Methodology

This section covers the aspects of Project Management and Development Methodology.
As an extra, a slight subsection is added dealing with aspects of version control and repository.

4.1. development methodology

The development methodology chosen for the development of this project has been Kanban[20], the agile methodology based on boards with labels.

The reason for this decision is marked by the scarcity of impositions on development that
It offers the methodology, the orientation to the number of people to which it is directed (normally small teams), and the facility to control the workload adequately. In addition, as it is an agile methodology, it allows a less marked development in specific phases.
and that supports changes more easily than other types of more closed methodologies or with some
Too strict design specifications.

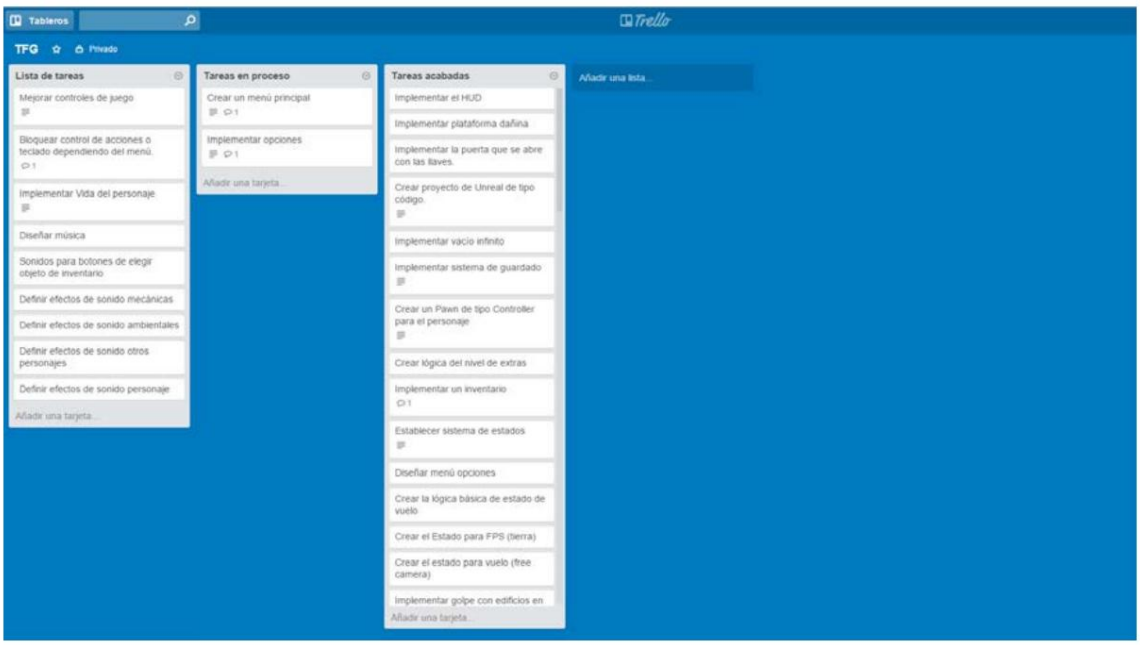


Figure 6. Capture of the Kanban organization of the project

Source: self made.

The workflow in this project has been to add new pending tasks to the board.
When one or several tasks were intended to be carried out, they were placed in a list of tasks in progress, and when they were done, they were placed last on a list of completed tasks.

4.2. project management

For the management of project tasks, it has been decided to use the Trello application, which allows the creating a board and organizing tasks by tags. The tasks in this application support a variety of options that are convenient when defining tasks or adding modifications or notes to these (in the form of comments in our particular case, as it is only a person). As it is multiplatform, it allows quick management of these from smartphones or tablets, to make possible changes or add new tasks at any time.

4.3. Version control and repository

This work is based on the realization of a Video Game in Unreal Engine 4, and also having Keep in mind that the number of participants in its creation is only the author himself. As of this memory, version control has not been established, since an Unreal project Engine 4 is usually quite heavy and making periodic uploads of files of this nature can require a considerable time, given also that as there are no more members in the development, only one person needs to work on the project, and finally, considering that the The same UE4 engine performs programmable periodic autosaves to avoid data loss. information, it has been decided to limit a couple of copies of the project for safety in a way manual.

5. Body of work

The body of development work is divided into two main blocks.

1. A first block dedicated to **the Video Game Design Document (GDD),** where the They cover all aspects of the video game design developed.
2. And a second block, **Development and implementation,** corresponding to the aspects most relevant technicians of the production of the study project.

5.1. Video Game Design Document (GDD)

Figure 7. Video Game Logo.

Source: self made

This block describes in detail all the design aspects necessary for the correct subsequent implementation of the developed video game. It is so much history, experience of gameplay, and level design, such as user interface and technical details.

This document is based on the GDD template offered in the subject of Fundamentals of Video Games[21], compulsory third year in Multimedia Engineering.

The name that has been decided to give to the project is **Arbennig: Wrong World.**

5.1.1. The game in general

Figure 8. Video Game Title.

Source: self made.

5.1.1.1. Summary of the argument

We move to an isolated and fantastic world, an isolated land of spirits embodied in totems and other artifacts, and beings from some prosperous extinct civilization and another new. We have born from a tree as a member of this ancient civilization, and our species is no longer belongs to this land.

However, all is not lost, as the ancient civilization traveled to another world and it is possible meet with them. To do this, you must access the interdimensional travel room, which is sealed and it is only possible to enter it by obtaining 8 runes, which are in the custody of 8 totems spread over this land.

The totems are possessed by spirits, and they will grant us the rune they guard in exchange for dealing with them and their needs.

Once all the runes are obtained, the seal of the final room will be broken, and we will be able to travel with our civilization.

5.1.1.2. Feature Set

The main attractive features of the video game are the following:

- **Attractive** visual section perfectly explorable.
- Compatible with the **Oculus Rift peripheral.**
- Developed with the recent **Unreal Engine** 4 engine.

5.1.1.3. Gender

Arbennig's genre is that of an open-world first-person video game, with puzzle component and skill. A clear example of this genre of this genre corresponds to games like The Elder Scrolls: Skyrim.

5.1.1.4. Audience

Given the characteristics of the design, the game is within an age rating PEGI12 type, according to the description offered by the official PEGI website on PEGI12 [23]:

Figure 9. Pegi12 logo.

Source: Wikimedia[22]

> *"This category includes video games that show violence of a somewhat more graphic nature towards fantasy characters and/or non-graphic violence towards human-like characters or recognizable animals, as well as video games that show nudity of a somewhat more graphic nature. Foul language must be mild and must not contain sexual profanity."*

Despite this, the video game is suitable for people of any later age.

5.1.1.5. Game Flow Summary

Below is a diagram of the basic game flow. In it we can see each one of the parts of the game, including menus, game and the game itself.

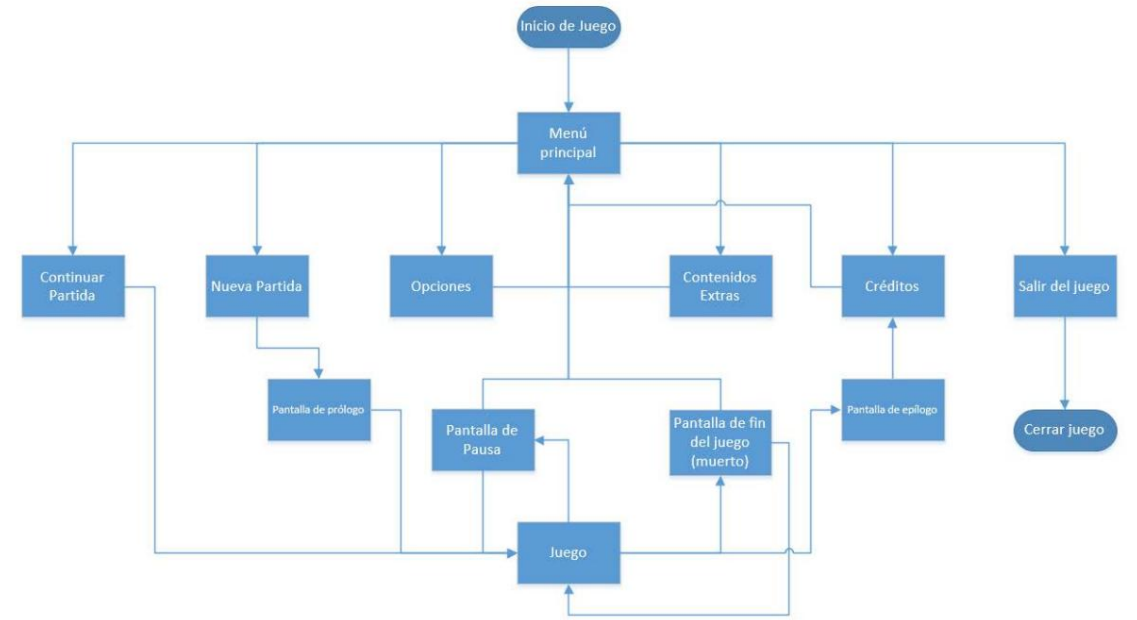


Figure 10. Flow of game screens.

Source: self made.

This game flow basically consists of:

1. Start game app.
2. A main menu opens with different options, among which we can continue the game (disabled if we don't have any yet), or start a new one, modify some configuration parameters, access to extra content of the story, to the credits of development, and, to the option to exit the game, which would take us out of the application. Will follow for starting a game to continue with the flow of the game.
3. We start a new game, in which we can beat the game 100%, which will would eventually lead to a game credits screen through the final room. Or, on the other side, we can access the pause menu and return to the main menu.
4. If, on the contrary, the player dies, an end game menu appears that gives the option to play again from the last save point, or to return to the main menu.
5. If we beat the game 100%, after accessing the credits, these take us, at the end, to the main menu.

5.1.1.6. game appearance

Regarding the gaming experience, there are two aspects that we intend to highlight mainly, the freedom and fantasy.

On the one hand, give the appearance of freedom to the player when it comes to discovering the environment that surrounds him. The ability to fly over a fully open and perfectly explorable world is the main point for this. In addition, the fact of being a video game in the first person causes a greater immersion and a greater sense of freedom.

On the other hand, fantasy, the game calls for a fantastic, idyllic world, a fantastic world, a space in which from the beginning of the game you will appreciate that you are going to be part of a adventure.

The game's soundtrack, both with music, ambient sounds, and sound effects, tries to enhance these sensations described above.

5.1.1.7. Scope

When we find ourselves before an open world video game, given the specific characteristics that this has, we have a single large general level, a common space on which it develops the adventure and the different challenges. This single world corresponds to the land of Delfrydoland.

Figure 11. Map of Delfrydoland from the editor.

Source: self made.

On the other hand, regarding the characters that we are going to meet, there will only be NPCs apart from our player. The NPCs that we will find correspond to 3 types:

1. **Help NPC (father tree),** who will assist us on controls and on what objectives the player has.
2. **NPC of adventure (totems),** for which it will be necessary to carry out some test and we reward.
3. **Generic NPCs (new inhabitants of the place),** to better inhabit the land of Delfrydoland, they do not speak or give any generic message about the place.

5.1.2. Gameplay and mechanics

This section describes both the gameplay aspects, which includes progression of the player, game missions, game controls, and others, as well as aspects of the mechanics of game, save system and a flow of the game in greater detail, among others.

5.1.2.1. gameplay

It should be emphasized, before going into gameplay issues, to know what it represents. A A clear definition is the one provided by Wikipedia[24]:

> *"The degree to which gamers achieve game-specific goals with effectiveness, efficiency, flexibility, security, and especially satisfaction in a playable context of use"*

Which we could summarize in how comfortable it is to play a video game.

5.1.2.1.1. game objectives

The player aims in the game to overcome all the challenges that are presented to him, either described above.

Once all of them have been overcome, the game is complete. Given the dynamics of the game, where the The player can overcome the different challenges that are presented to him in the order he wishes, both in the flight school, like those of the characters, the game does not launch an ending once the last one is overcome of the challenges, but access will be unlocked through a special room that leads to the credits of game when all of them have been completed, and whose credits symbolize the end of the game. This special room is accessible after getting all the special game keys.

5.1.2.1.2. Progression

The player's progression in the video game is marked by obtaining the runes of the totems.

As the player overcomes the missions or challenges, he will acquire greater skill in the video game, as well as learn more about the world around him and the ancient civilization.

In addition, after managing to obtain all the runes, you can use them to access the final room that allows you to finish the game.

5.1.2.1.3. Missions and challenge structure

As for the challenges in the game, they all have in common that they are posed by the 8 totems that are scattered around the stage.

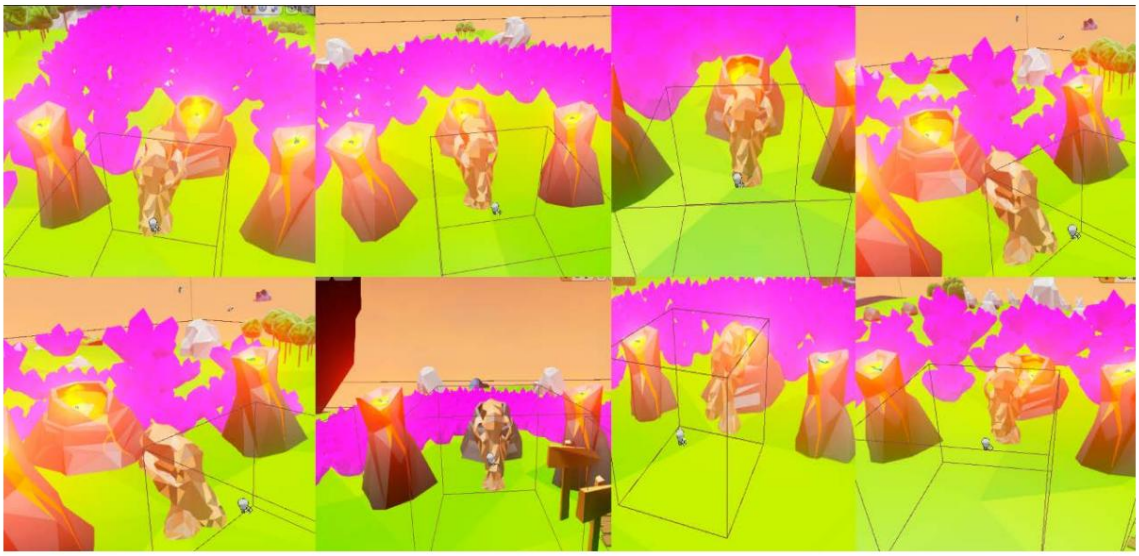


Figure 12. Screenshots of the 8 totems from the editor.

Source: self made.

In addition, it is not possible to perform more than one challenge at a time, that is, when starting a totem check, no further checks possible until test ends, or the totem is spoken to with the intention of canceling the test.

The game features the following challenges based on each totem.

1. **Totem 1. Listening challenge.** In this challenge it is only necessary to find this totem, since he is the first, and talk to him until he finishes telling everything he has to say about the game world.

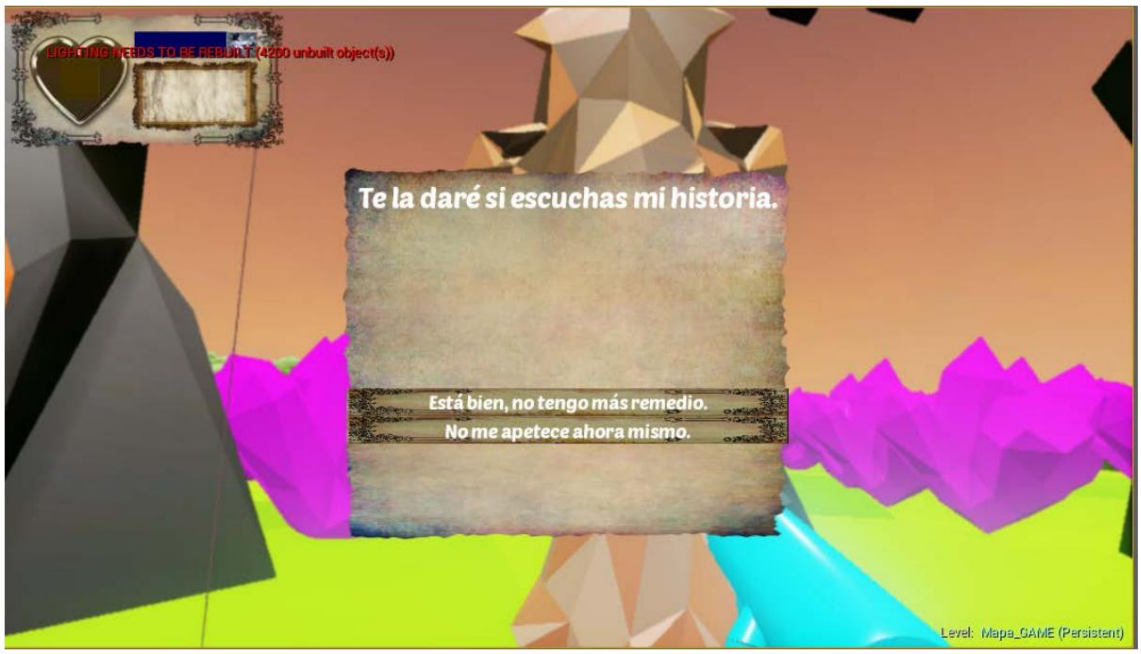


Figure 13. Capture of the challenge of the first totem.

Source: self made.

2. **Totem 2. Diffusion in the town.** The totem commands us to spread a message to 9 NPCs in the populated island of the game. For this, it provides us with 9 message letters that we must deliver them and, after this, return to the totem to notify him that we have completed the challenge.

Figure 14. Capture of the second totem challenge.

Source: self made

3. **Totem 3. Cleaning.** In this challenge, the totem indicates that your island is the farthest from everything the world, and that weeds (10 bushes) have appeared on the island that you want let's withdraw The weeds must be placed on a piece of wood that indicates, and once all are removed, the challenge is over.

Figure 15. Capture of the third totem challenge.

Source: self made.

4. **Totem pole 4. Rings challenge 1.** A series of rings is set up on stage through which the player must pass (through its interior) before the marked time runs out. One time gone through all the hoops, the player overcomes the challenge. In case of not achieving the challenge in the indicated time, the player must repeat the challenge by speaking to the totem again.

Figure 16. Totem 4 challenge capture.

Source: self made.

5. **Totem 5. Labyrinth.** On this island the player must enter through a labyrinth underground, in which at the end of it is the totem. This challenge consists of overcome the maze.

Figure 17. Totem 5 challenge capture.

Source: self made.

6. **Totem 6. Pursuit.** The totem orders us to run to the other end of the island where is found, pick up a flag, and go back and talk to him. In this race, the player will be chased by several Pursuing Golem-type NPCs, which should not reach us if we want to rise to the challenge. Also, flying is not allowed, only running, and in case of flying, the player loses the challenge and is teleported to the entrance.

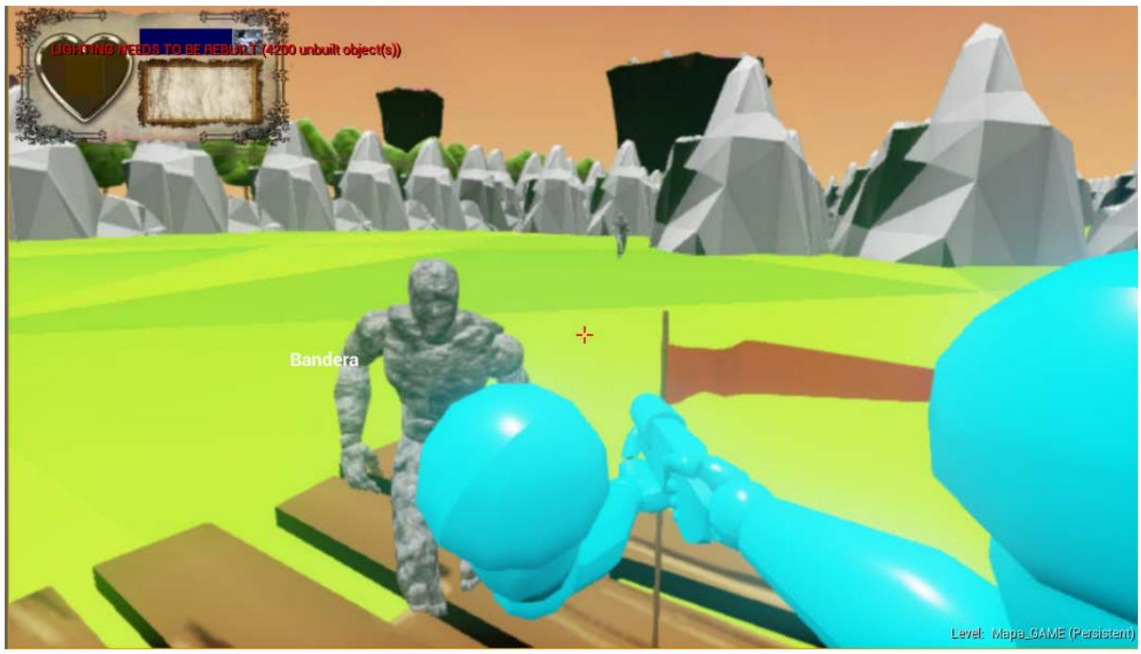


Figure 18. Totem 6 challenge capture.

Source: self made.

7. **Totem pole 7. Hoops challenge 2.** Follows the same operation as the game island challenge.
totem 4 but change the earrings for another type.

Figure 19. Totem 7 challenge capture.

Source: self made.

8. **Totem 8. Search.** The player must enter the forest in the south of the world,
and find a flag placed there. He must figure out how to get through the forest (passing

through the path without vegetation without flying high), because if it does not go through the path suitable, is teleported to the entrance of the forest.

Figure 20. Totem 8 Challenge Capture.

Source: self made

Regarding the last aspect of the game's missions, that is, access to the final room for completing it, the player can explore the world and access a terrain containing a building with 8 locks. As you get keys, they will appear in your HUD, and when you get them all, you will be able to interact with the 8 locks, which will now have the runes, and the door blocking the passage to the room will open.

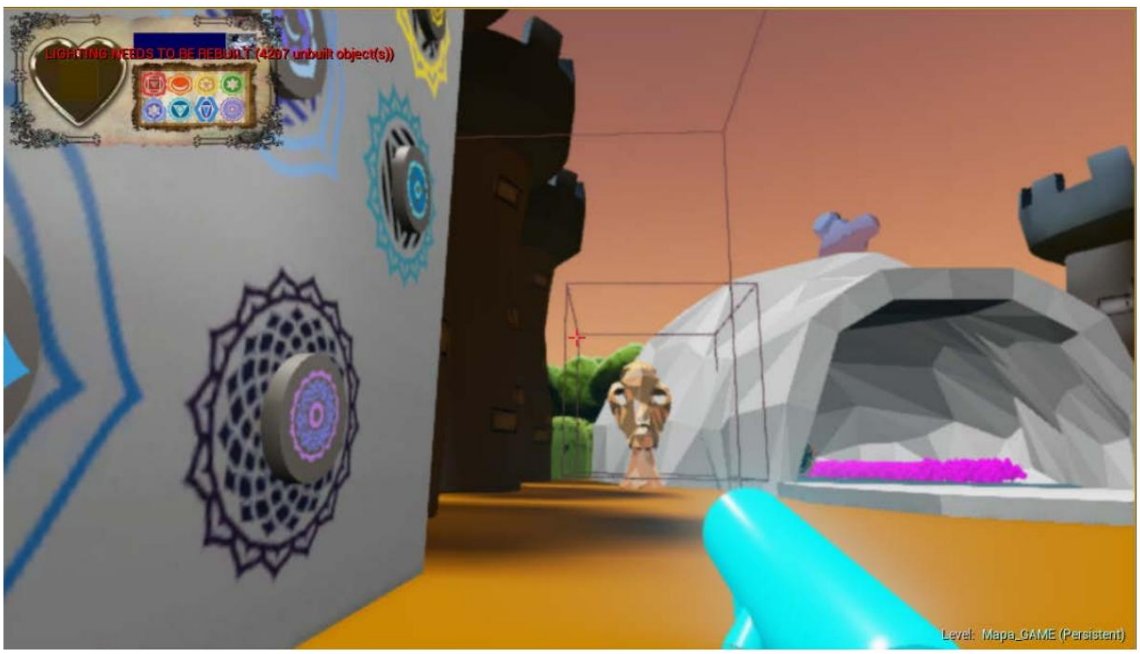


Figure 21. Screenshot of the door at the end of the game opened by placing the runes.

Source: self made.

5.1.2.1.4. character actions

The character can perform actions both on the ground and in the air. It has three states clearly differentiated, between which you can freely switch:

ground state.

In this state the character cannot fly, and can perform the different actions:

a. Move in all directions you want (but not up or down).
   Yo. Forward.
   ii. Backward.
   iii. To the right.
   IV. Toward the left.

b. turn around
   Yo. ±360º on the Y axis
   ii. ±90 on the X axis

c. Skip.

d. To run.

and. Interact with the environment.
   Yo. Talk to other characters.
   ii. Pick up/Use/Interact with non-character items on stage.

F. Activate flight status

flight status.

In this state, much like the ground state, the player can perform the following actions:

g. Move in all the directions you want.
   Yo. Forward.
   ii. Backward.
   iii. To the right.
   IV. Toward the left.
   v. Upwards.
   saw. Down.

h. turn around

Yo. ±360º on the Y axis.

ii. ±90 on the X axis.

Yo. Interact with the environment.

Yo. Talk to other characters.

ii. Pick up/Use/Interact with non-character items on stage.

J. Activate ground state.

k. Activate the state of fast flight or Racing.

l. Dry braking.

m. Plummet.

2. **Fast or Racing flight status.**

In this state, the player has a speed that increases progressively until reaching a maximum, which, once reached, remains constant. actions that can perform the player are as follows:

a. Move (cannot move backwards because the Racing state makes you move forward to the player with increasing speed, in addition, since automatically moves forward, the character does not have control of the forward movement):

Yo. Upwards.

ii. Down.

iii. To the right.

IV. Toward the left.

b. turn around

Yo. ±90º on the X axis

ii. ±90º on the Y axis

c. Go to flight state.

d. Activate extra turbo. This turbo consumes the player's energy bar, which is regenerates when the turbo is not active and when passing through a hoop.

5.1.2.1.5. game controls

It is possible to play using keyboard and mouse, and with or without oculus rift.

The system automatically detects when you start the game if we have the device connected or not oculus rift, if so, the camera can be controlled by it, regardless of whether we are using the mouse for it.

On the other hand, the game controls for keyboard and mouse, including the optional controls of the Oculus Rift son peripheral:

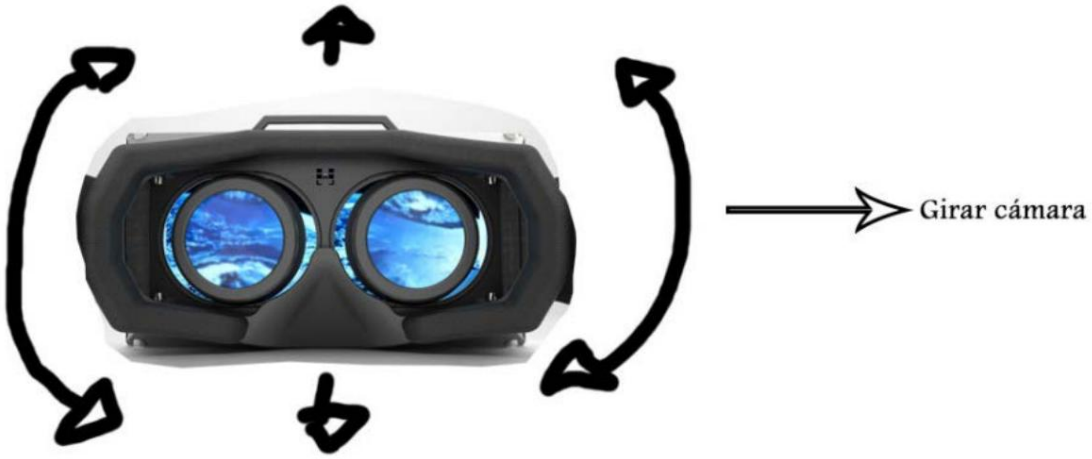


Figure 22. Using Oculus in game.

Source: self made.

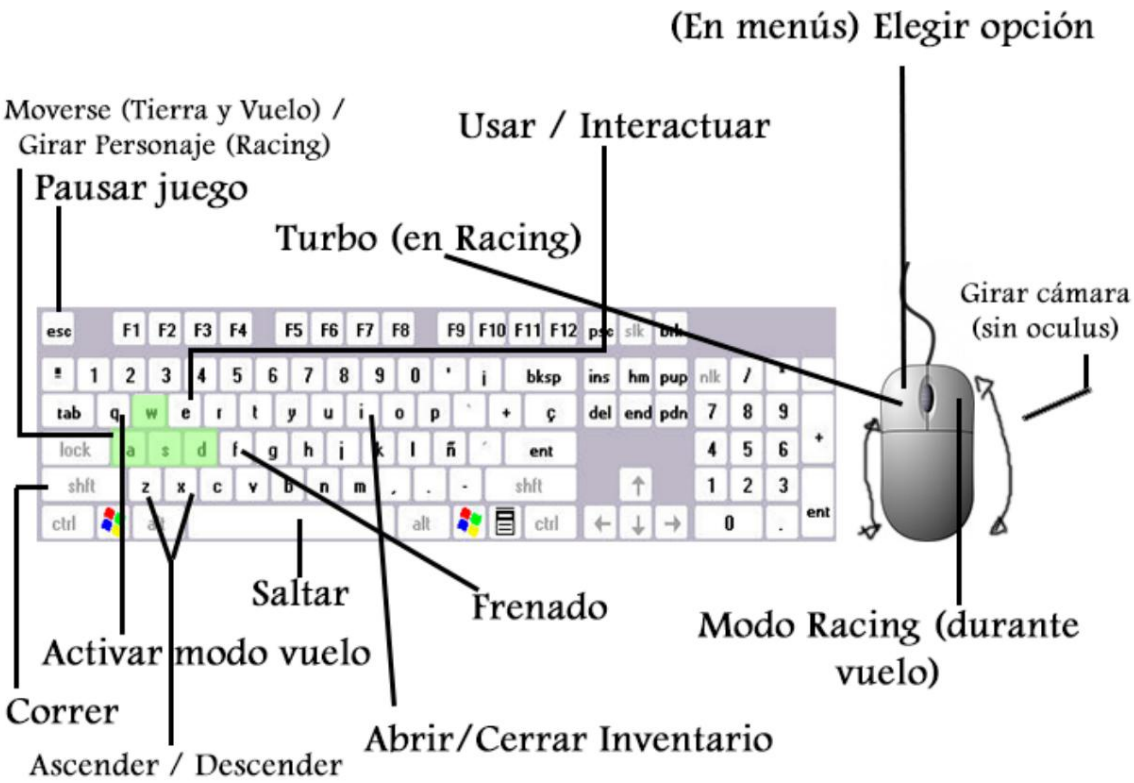


Figure 23. Game controls with keyboard and mouse.

Source: self made.

5.1.2.2. mechanical

By game mechanics we mean something that the player interacts with in order to create or contribute to the gameplay, such as platforms that move, ropes to sway, or slippery ice. In this section, in addition to game mechanics, we will include hazards (mechanics that can hurt or kill the player, but have no intelligence, such as damaging platforms), power-ups (an item that is picked up by the player to help them with their gameplay, such as a turbo), and collectibles (items that are picked up by the player, but they do not have an immediate impact on gameplay, such as game keys).

That said, in Arbennig, the mechanics we find are the following:

mechanical

teleporter.

Figure 24. Teleporter and teleport destination painted with debug lines, respectively, from left to right.

If the player comes into contact with that component, they are teleported to another area of the map.

Door that opens.

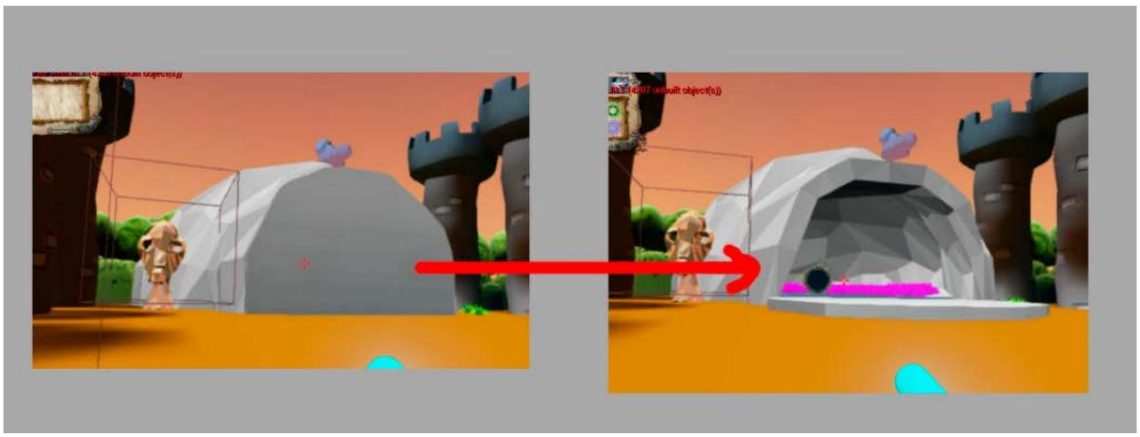


Figure 25. Closed and open door captures.

Source: self made.

Doors that by meeting certain conditions or simply interacting with them, open.

~~Door at the end of the game.~~

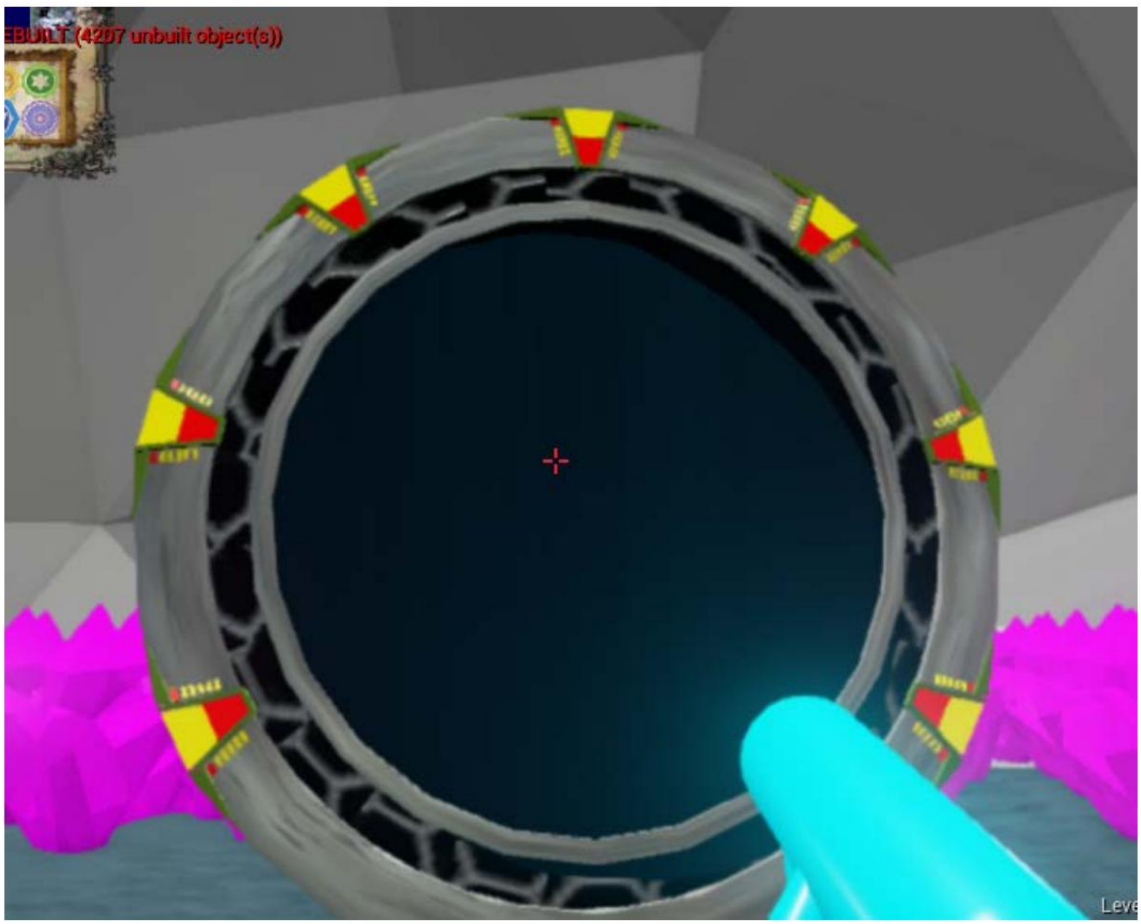


Figure 26. Final Gate Capture.

Source: self made.

This door allows us to finish the game once we interact with it or go through it.

dangers

Platforms/Harmful substance.

Figure 27. Capturing various harmful platforms (purple)

Source: self made.

Components that if the player touches them, they lose life. It is possible to find them in the challenges of hoops.

Instant death platform.

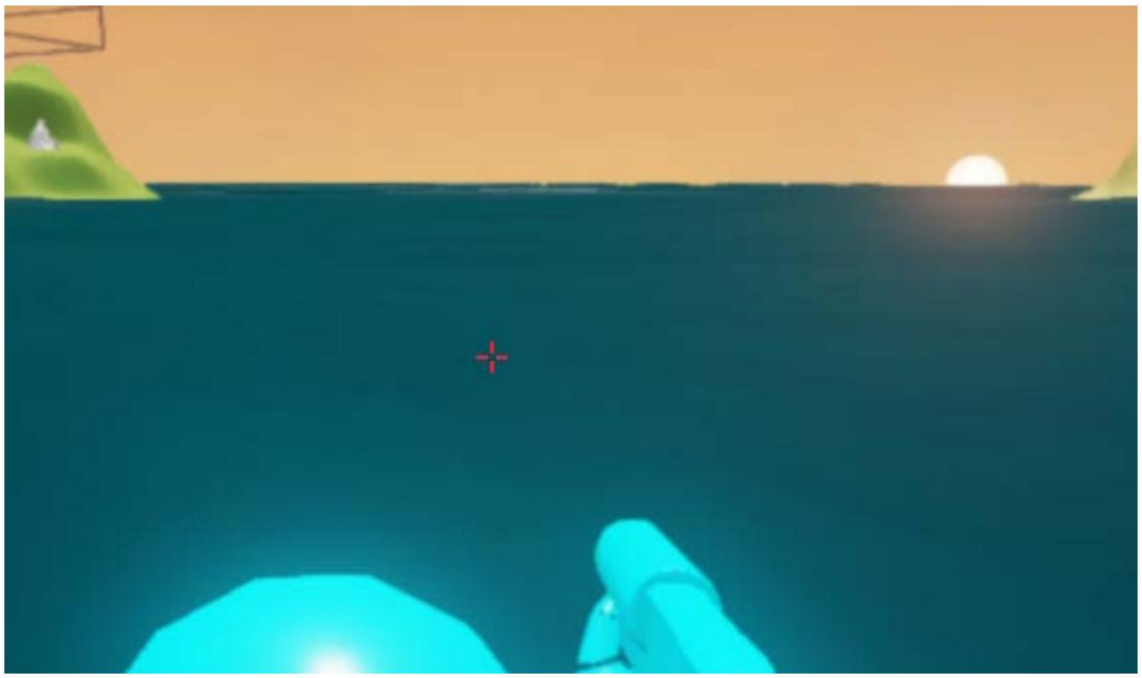


Figure 28. Game Sea Capture (Instant Kill Pad)

Source: self made.

The player dies upon contact with her.

Power Ups

Aro.

Figure 29. Capture of the two types of hoops in the game.

Source: self made.

Provides the player with an instant recharge of their energy for turbos.

Save point.

Figure 30. Capturing a save point.

Source: self made.

Provides the player with a recharge of their health and energy for turbos.

collectibles

Speech.

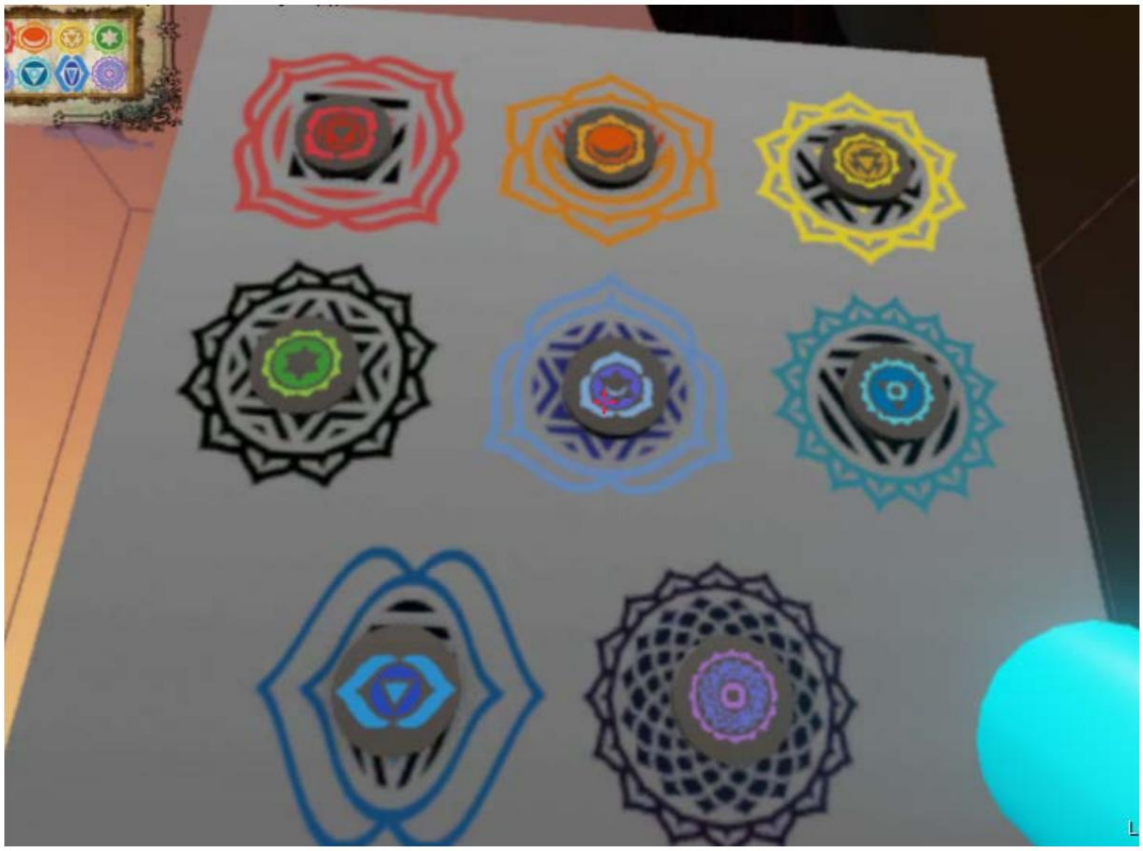


Figure 31. Capture of the 8 runes placed on the final wall.

Source: self made.

When the player overcomes the tests and challenges that are imposed on him, he gets some special keys that allow him to access a final room to finish the game. You can also find
in the world more keys.

Island totem elements 2. Brochures.

Figure 32. Screenshot of informative brochure.

Source: self made.

The player can pick them up for later delivery through their inventory.

Island totem elements 3. Bushes.

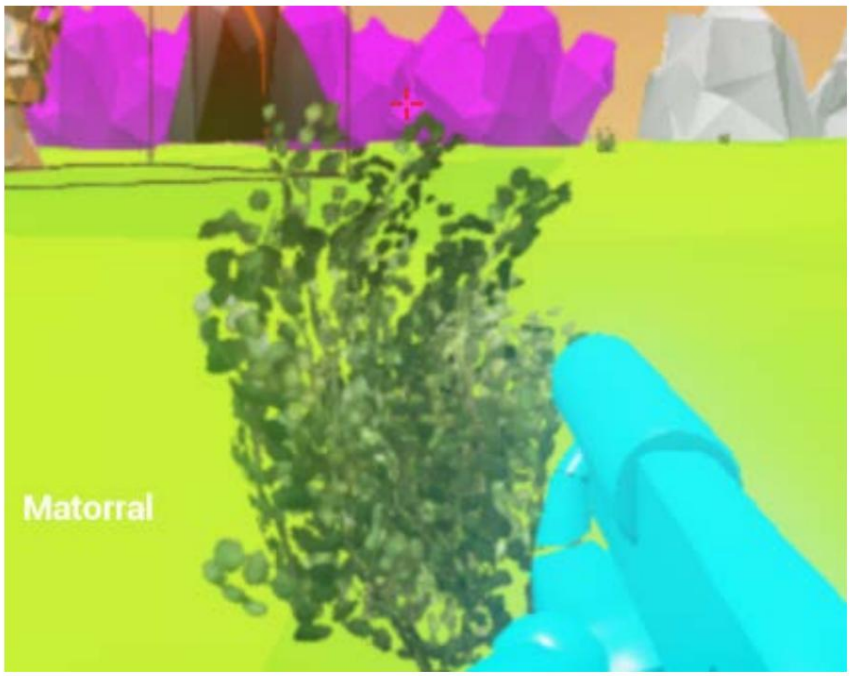


Figure 33. Scrub capture.

Source: self made.

The player can pick them up to drop off elsewhere through their inventory.

5.1.2.3. game options

Figure 34. Capture of the game options.

Source: self made.

The game options, only accessible through the main menu, allow you to modify video and audio parameters.

These parameters are:

- Modify the game resolution, giving you a choice between several options of 4:3 format and 16:9 format.
- Modify the general volume of the game.

5.1.2.4. replay and save

To save the game, it is possible to access different save points, distributed throughout the game. scenery.

Figure 35. Capturing a save point.

Source: self made.

These save points store the player's progress (totems passed), and leave the player player in the position where the save point is located.

When the game is launched from the desktop, the game is automatically loaded if the there is, showing locked or not the option to continue game in the main menu (depending on whether or not there is a saved game).

Figure 36. Screenshot of the main menu with the option to continue blocked (there is no game saved).

Source: self made.

In addition to this, when loading the level of game extras, it contains more or less amount of extras based on the progress made by the player.

Figure 37. Capture of the level of extras.

Source: self made.

If you select the option to start a new game when there is already a saved game, the saved game would be lost as soon as the player saved to the new game, overwriting the previous one. The player is warned of this step, in case they clicked unintentionally.

Figure 38. Screenshot of the new game confirmation menu.

Source: self made.

On the other hand, if the in-game player dies, they are sent to a game over screen, which gives you the option to play again from the last saved point. losing all progress made before saving the game.

Figure 39. Capture of the end of game menu.

Source: self made.

5.1.3. Story, Features, and Characters

5.1.3.1. History

The land of Delfrydoland is a mysterious place. Isolated from time and space with the world Outside, lives a society that does not know much about its past, present, or future. A society of beings created by a tree of life that maintains a natural balance, giving rise through of his powers to creatures.

This society of beings with the appearance of an old man, has all the same appearance, in addition, they do not eat, they do not sleep, they do not carry out any activity, they do not age, and they do not even have a name or occupy their leisure. They usually live in a village that is on an island where they are simply waiting for some event to occur, which probably will not be due to them. What's more, the village has not even been built by them, although that does not matter to them.

In this environment, where nobody asked many questions about things, the tree of life creates a member of the pre-current civilization of elders in Delfrydoland, creates a Special.

Arbennig belongs to the Barod civilization that decided to leave the world of Delfrydoland to seek new adventures in other worlds. The Barod were the creators of the tree of life, and timed the creation of a member of their species centuries after their march so that he could inform them about the world they left behind, but in order to inform them, Arbennig should prove that he is capable of traveling to other worlds to live adventures with them. That is, in order to change worlds, they left a world teleporter in inside a stone room sealed by 8 magical runes, which would be in the custody of 8 totems that would deliver it to Arbennig in exchange for performing a challenge.

Thanks to the totems and some fairies, Arbennig manages to find out that the Barod, a very prosperous, intelligent species, with a great desire to discover new worlds, ended up tired of the wonderful but limited world of Delfrydoland when there was nothing, in his opinion, they could discover or achieve in it, and they created a special technology that allowed them to travel to another larger world to explore.

And finally, when Arbennig manages to collect all the keys and gain access to this machine transporter, makes the decision to return to his kind and live many new adventures.

5.1.3.2. game world

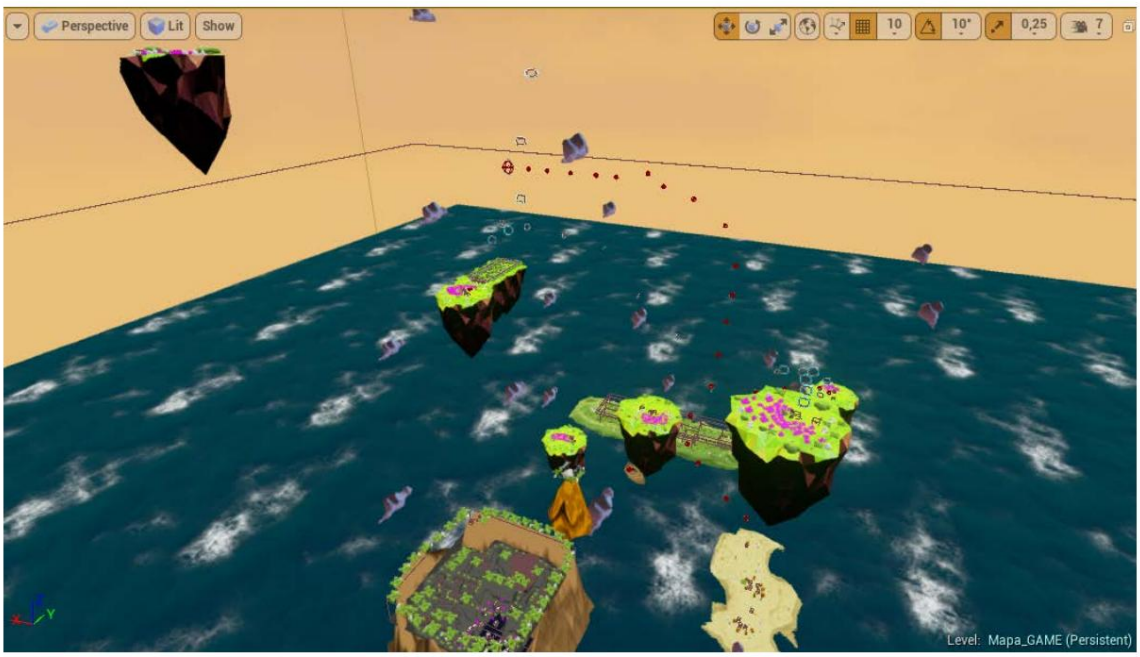


Figure 40. Capture of the game world from the editor.

Source: self made.

The game world is a varied place in the middle of an ocean that features both islands in it, like floating islets in the sky. In total we find 12 formations, 5 islands in the sea, and 7 islands floating.

5.1.3.2.1. Start Island.

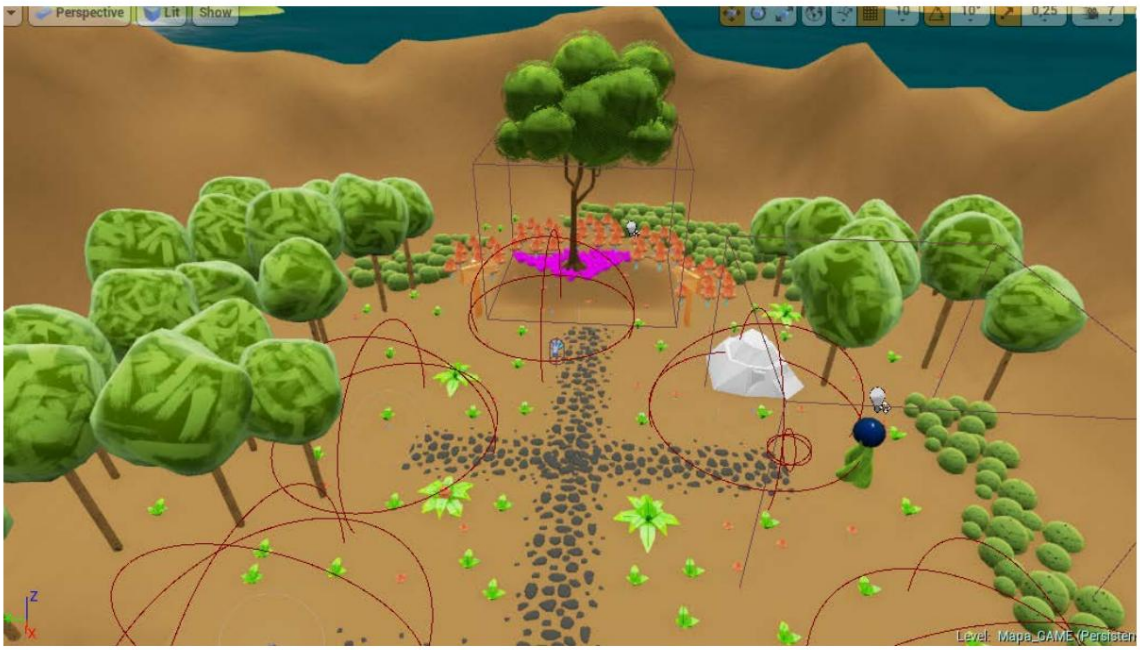


Figure 41. Screenshot of the home island from the editor.

Source: self made.

On this island is where the player is born, here we find the tree of life and a series of items that teach us how to play.

5.1.3.2.2. Totem Island 1.

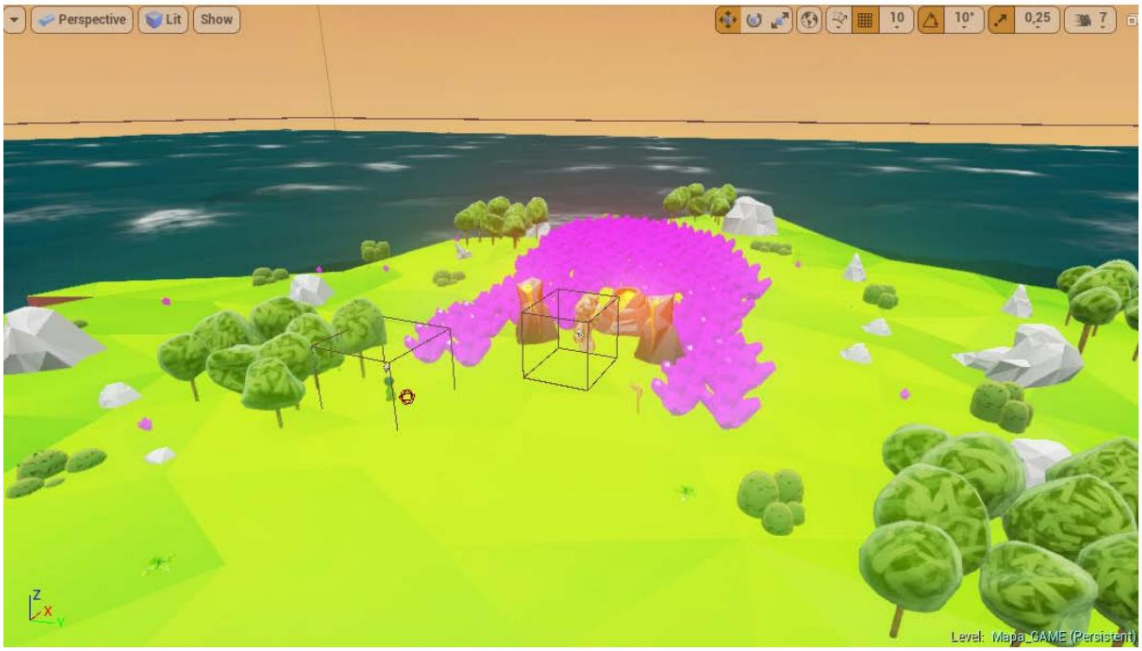


Figure 42. Screenshot of totem island 1 from the editor.

Source: self made.

Totem 1 island is the closest to the starting island, it is a floating island.

5.1.3.2.3. Totem Island 2.

Figure 43. Screenshot of totem island 2 from the editor.

Source: self made.

Like the island of totem 2, we find ourselves before another floating island, where we find the defiance totem that wants us to let you know that it will disappear when it gives us the rune that protege.

5.1.3.2.4. Totem Island 3.

Figure 44. Screenshot of totem island 3 from the editor.

Source: self made.

Totem Island 3 is the farthest island in the entire game, located at one end and far altitude.

5.1.3.2.5. Totem Island 4.

Figure 45. Screenshot of totem island 4 from the editor.

Source: self made.

On this floating island we find the totem that sends us to perform the first flight challenge.

5.1.3.2.6. Totem Island 5.

Figure 46. Screenshot of totem island 5 from the editor.

Source: self made.

Totem Island 5 is the one that forms a labyrinth created by the Barod, and is full of teleporters that take us to the exit, while the totem pole is at the end of the teleporter.

5.1.3.2.7. Totem Island 6.

Figure 47. Screenshot of totem island 6 from the editor.

Source: self made.

Totem Island 6 is the largest of the floating islets, since there are 3 islets together, where find a test of speed and ability to dodge some stone characters from the challenge.

5.1.3.2.8. Totem Island 7.

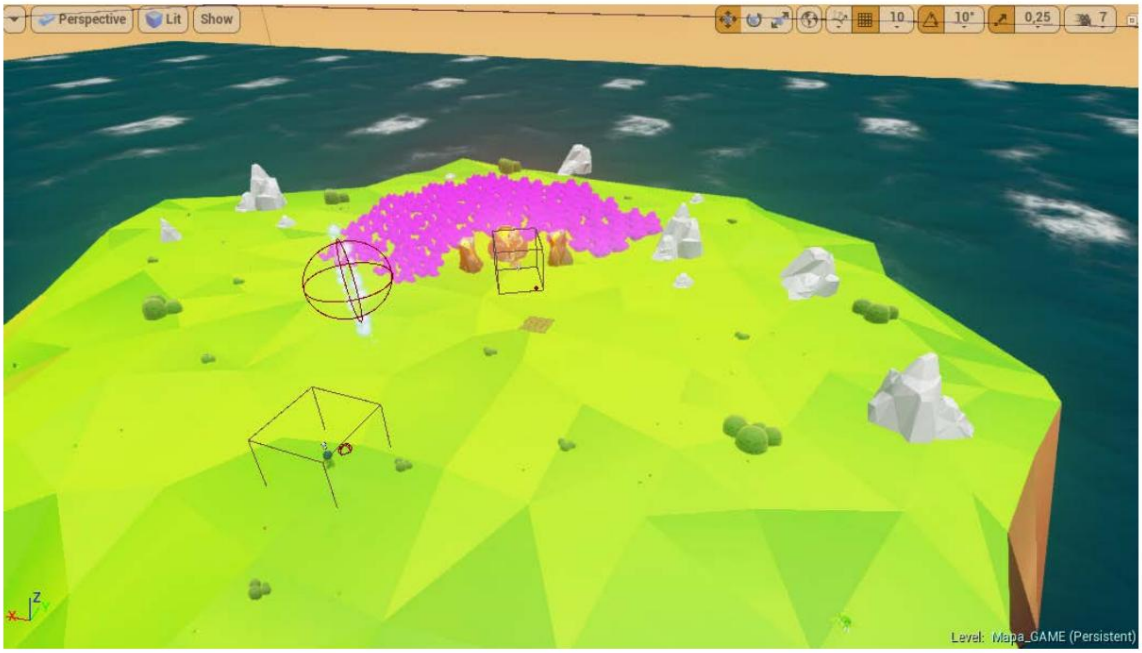


Figure 48. Screenshot of totem island 7 from the editor.

Source: self made.

Island 7 contains another hoops challenge, more complex than the one on Island 4.

5.1.3.2.9. Totem Island 8.

Figure 49. Screenshot of totem island 8 from the editor.

Source: self made.

Island 8 is where the totem is located that sends us to look for a flag on the forest island.

5.1.3.2.10. Town Island.

Figure 50. Capture of the village island from the editor.

Source: self made.

On this island we find the new civilization created by the tree. The houses are houses of Barod, they do not have doors because they entered using teleporters.

5.1.3.2.11. Forest Island.

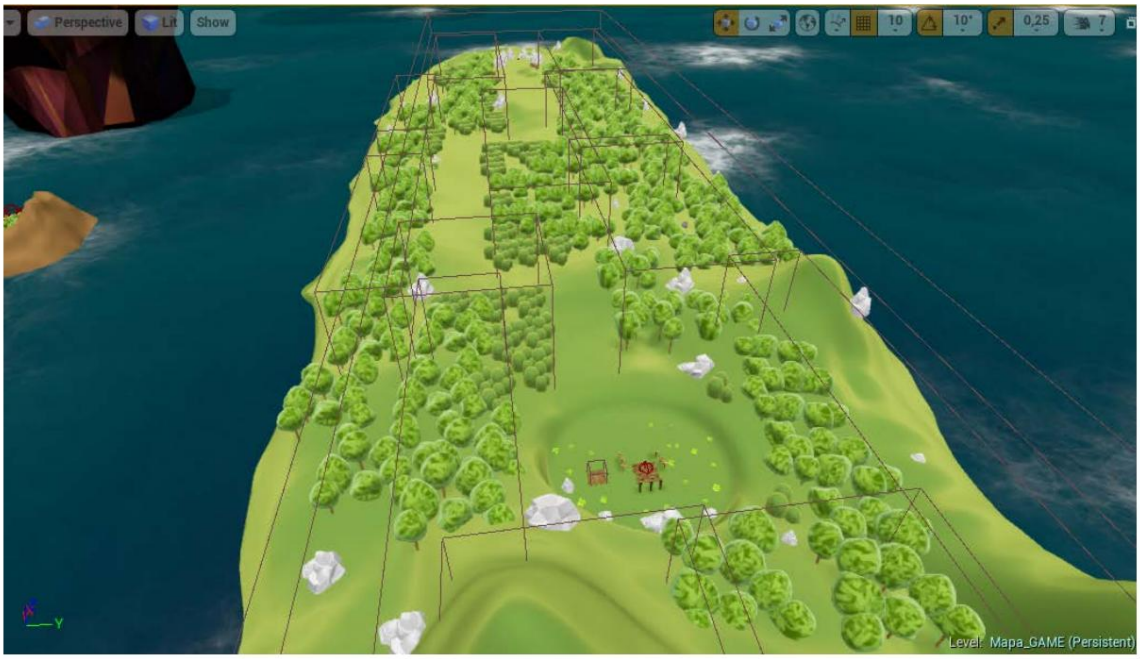


Figure 51. Capture of the forest island from the editor.

Source: self made.

The forest island is a place full of teleporters, which to cross it is necessary to follow a path marked on the floor of it, and that we must follow if we want to overcome the challenge of the island totem pole 8.

5.1.3.2.12. Isla final.

Figure 52. Capture of the final island from the editor.

Source: self made.

This final island is where we find the stone room that must be opened once we have all runes.

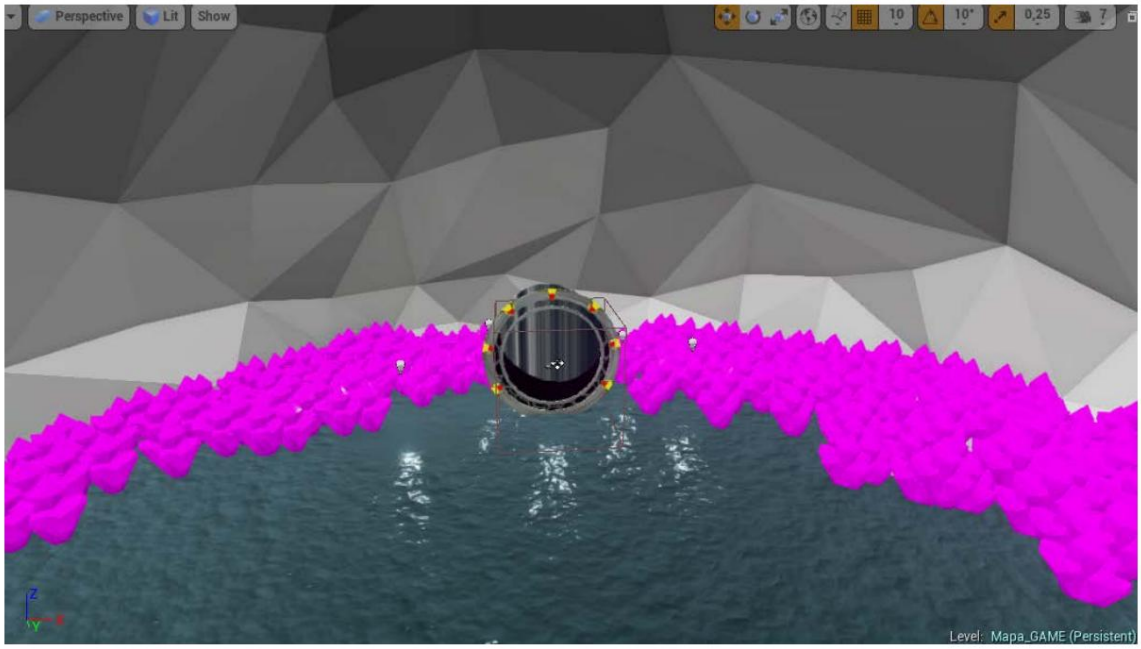


Figure 53. Capture of the interior of the final room from the editor.

Source: self made.

Inside the room is the teleporter.

5.1.3.3. Personaje Principal. Special.

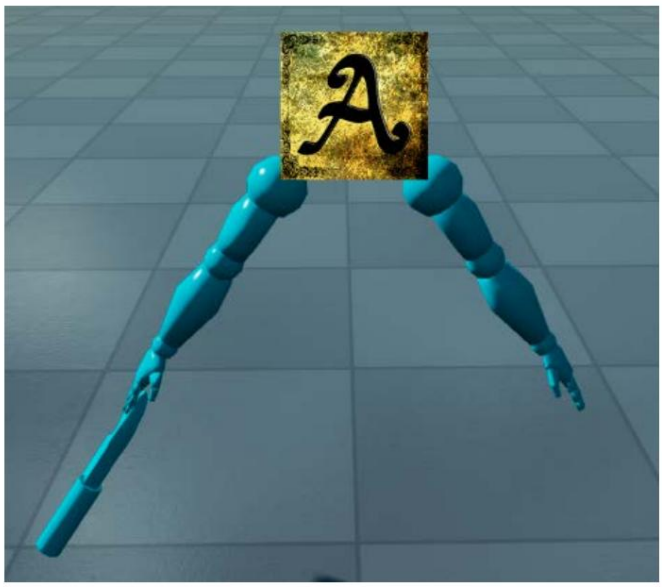


Figura 54. Special Personaje en primera persona.

Source: self made.

Arbennig is the Barod created by the tree of life. You must deal with the 8 totems of the world to be able to obtain the runes that will allow you to travel with your civilization.

His birth was timed by the Barod for centuries after he had left, from
so that it could inform about the world of Delfrydoland.

Arbennig is the character who can perform character actions (5.2.1.4), and also who
It has life and energy for turbos flying.

5.1.3.3.1. NPC character 1. Tree of life.

Figure 55. Capture of the tree of life.

Source: self made.

The Tree of Life is one of the Barod's greatest inventions. It can create living beings and communicate telepathically with any being in the world of Delfrydoland. He is the creator of Arbennig, and also of the new civilization with the appearance of an old man.

5.1.3.3.2. NPC character 2. Totems.

Figure 56. Capture of a totem from the game in debug mode.

Source: self made.

They have been created by the Barod to store the runes that will allow Arbennig to access the final teleportation room to another world. They are all the same, although each character has their own personality.

When they give the rune to Arbennig, their soul is no longer bound to the totem, so they stop be able to communicate with the world.

It exists despite this, a totem that never leaves its body nor does it have a rune to give up, it is the totem pole found in the final room, placed there to explain to anyone who approaches how can he get into the sealed room.

5.1.3.3.3. NPC character 3. Beings with the appearance of old people.

Figure 57. Capture of the beings with the appearance of an old man.

Source: Own elaboration (free mesh from the unreal mixamo pack).

It is the new civilization that Delfrydoland inhabits, they don't have much to say other than a couple of sentences to whoever wants to talk to them, nor do they have much to do. Plus they all look the same.

5.1.4. Game level – Delfrydoland.

Figure 58. Game level height.

Source: self made.

5.1.4.1. Summary

This world is the only one in the video game. It is an open world that the player can explore and also do 8 missions that will allow him to access the final room where the game ends. This is where we find all the characters and places described above, and all the mechanics and missions.

5.1.4.2. introductory material

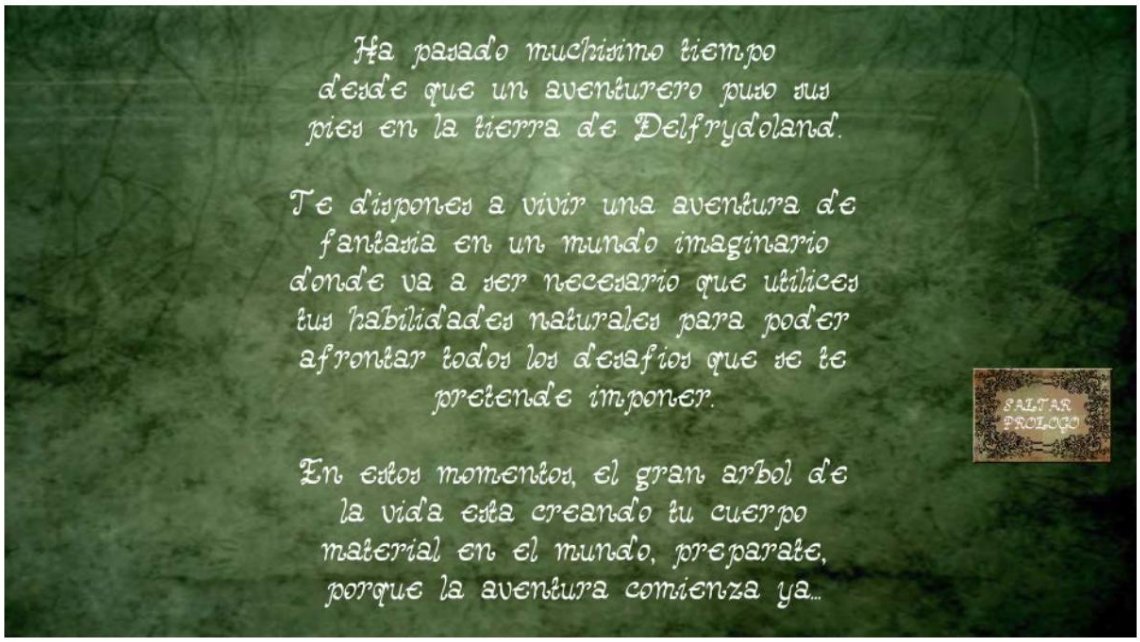


Figure 59. Prologue capture.

Source: self made.

Before accessing the level as such, a prologue screen appears that allows us to know a little of its history. This prologue can be skipped with a button.

In addition to this, once the game starts, the player finds himself on an initial island full of of help messages that will let you know the game controls, and more details about the world.

5.1.4.3. Objectives

The objectives to achieve in the world of Delfrydoland are all those described in the section on missions and challenge structure (5.1.2.1.3).

5.1.4.4. Map

Figure 60. Game map.

Source: self made

5.1.4.5. Roads

The player, since he can perform flying actions, and since he is in a world open, you can do any of the game's missions in any order you like. That is, from

the starting island where it appears, you can freely go following the path you want until
any island with a totem to perform the challenge indicated by it.

5.1.4.6. encounters

Exploring the world, it is possible to meet the different characters described in the section
of characters (5.3.3). That is, you will be able to meet the 8 challenge totems or the island totem
end, with the tree of life, and with the beings with the appearance of an old man that are found almost every
in the village, except for one, which is on the labyrinth island. Every character mentioned
has something to say to the player, providing information.

5.1.4.7. level guide

As such, there is no level guide, as mentioned in the Paths section (5.4.5),
the player can perform any of the challenges in the desired order, the logic of which we find
described in the missions and challenge structure section (5.2.1.3).

Once the player completes each of the challenges, they can access the final room,
interacting with a pillar where the runes are placed, which will open the access door to the final room
of the game, with a teleporter that when going through it or interacting with it, makes the game
term.

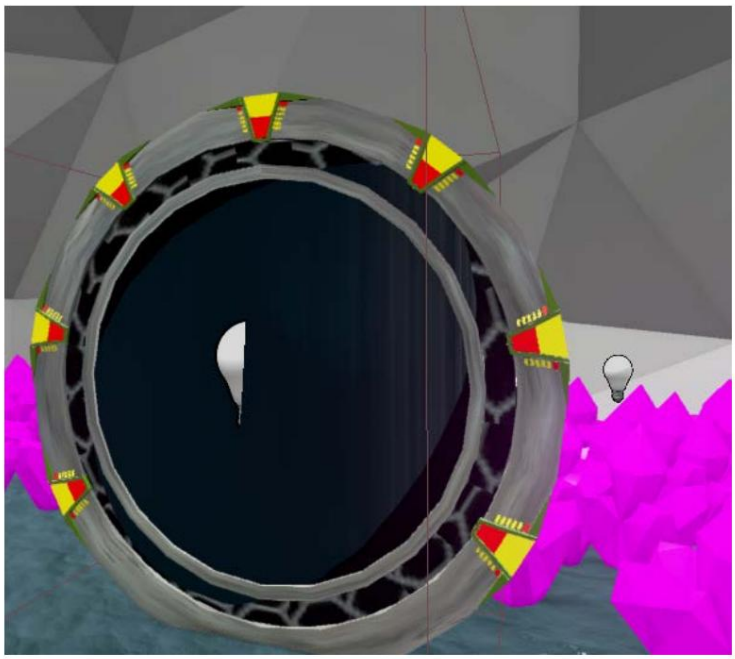


Figure 61. Capture of the door at the end of the game from the editor.

Source: self made.

5.1.4.8. closure material

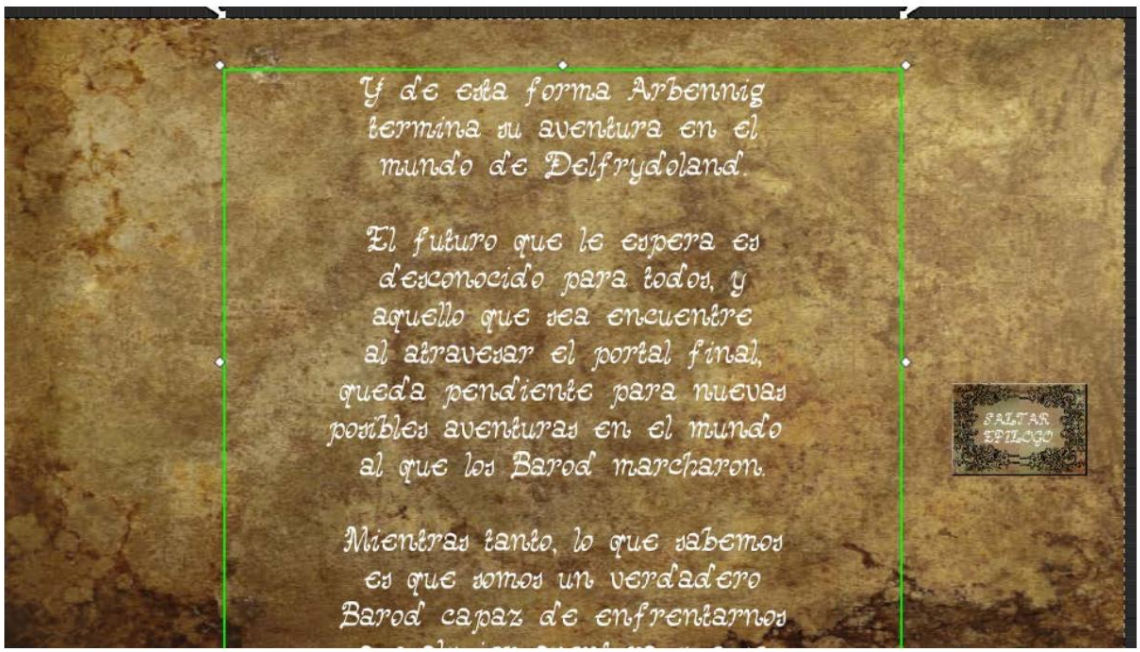


Figure 62. Capture of the epilogue from the editor.

Source: self made.

When the player touches or interacts with the teleporter, a screen with a text is loaded anime, containing the game's epilogue. The epilogue tells the player who has beaten Hit the level and you are venturing into new worlds.

When the epilogue ends, which you can skip if you don't want to see it, you go to the credits, which follows the same operation of the epilogue and the introduction. And finally to finish the credits or skip them, you access the main menu.

5.1.5. Interface

5.1.5.1. HUD

The HUD (heads-up display), remember as briefly mentioned in the section on state of the art, it is 2D information that appears drawn on the game screen, showing useful information to the player.

Figure 63. Screenshot of the permanent HUD in the game level from the editor.

Source: self made.

In Arbennig, we show on the HUD permanently once the game starts:

- The life of the character.
- The energy of the character.
- The runes obtained and not obtained.

On the other hand, the HUD shows the game inventory when the player activates it, which in turn In turn, it has an action submenu that appears when we select an object from the inventory.

Figure 64. Capture of the Inventory from the editor.

Source: self made.

We also show information about the rings collected and time remaining when you perform the ring challenge tests.

Figure 65. Capture of the timer and hoop counter from the editor.

Source: self made.

And we'll pop up the notification whether or not the player has passed any totem challenges or saved a game, or whether or not they can use any items via the HUD.

Figure 66. Other notifications from the HUD.

Source: self made.

5.1.5.2. menus

The menus will allow us to access the game level, modify aspects of the game (resolution and general volume), access extra levels, exit the game, pause it, or access the credits,
among other things.

Next we are going to explain each one of these menus and screens, for which we are going to to support the game flow figure in the Game Flow Summary section (5.1.1.5).

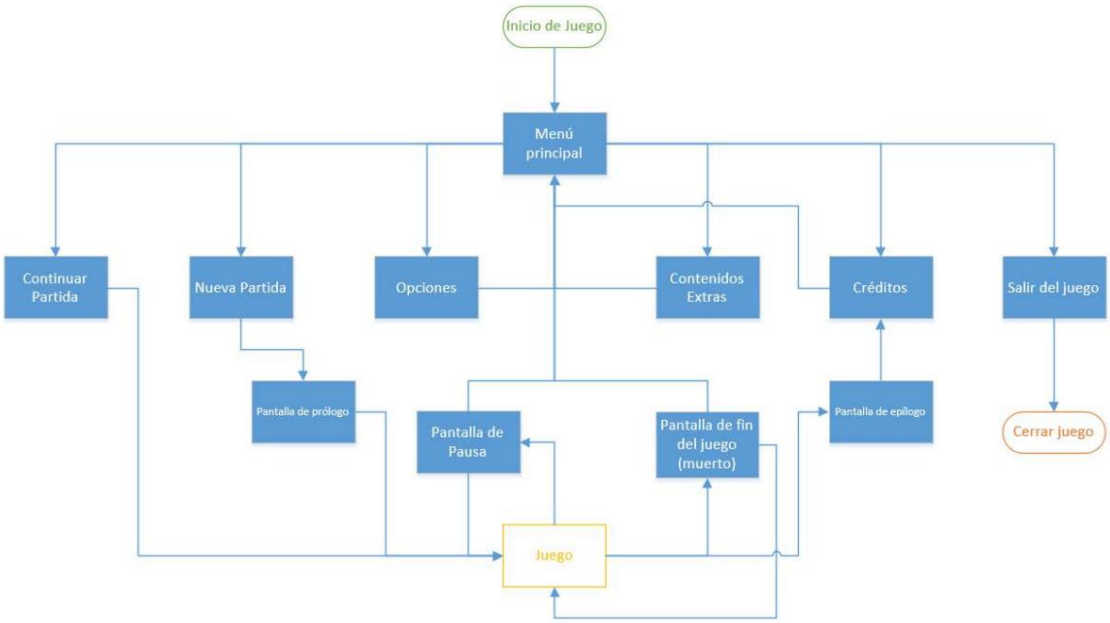


Figure 67. Game menu diagram.

Source: self made.

5.1.5.2.1. Main menu.

Figure 68. Screenshot of the main menu from the editor.

Source: self made

This menu is the one that opens when you start the game. In it we find the option to start or load (if we have saved data) the game, as well as access to the options menu, at the level of credits, and extras. It is through this menu that we can exit the game, from the option leave.

5.1.5.2.2. Option menu.

Figure 69. Screenshot of the options menu from the editor.

Source: self made

It allows us to modify the screen resolution, giving us a choice between three 4:3 formats, which are 640x480, 800x600, and 1024x768, and three other 16:9 formats, which are 1280x720 (the option that appears by default), 1366x768, and 1920x1080. When a resolution is set, the button to select it is disabled.

This menu also allows you to modify the general volume of the game through a slider, and by Finally, return to the main menu.

5.1.5.2.3. Pause menu.

Figure 70. Capture of the pause menu from the editor.

Source: self made

The pause menu, which appears when the player pauses the game (obviously), gives us the option to return to the game or return to the main menu.

5.1.5.2.4. New game confirmation menu

Figure 71. Screenshot of the new game confirmation menu from the editor.

Source: self made

This menu jumps when we click on the option of the main menu of new game, existing an existing saved game already. From this menu we can return to the main menu, or confirm that we want the game to be deleted and start a new one.

5.1.5.2.5. Game over menu (when player dies)

Figure 72. End of game menu after the player dies.

Source: self made

When the player touches the death platform (the ocean water in the game), or by colliding with various damaging platforms and runs out of health, this menu is loaded.

Here we are given the option to start at the last save point, or to return to the main menu.

Prologue screen.

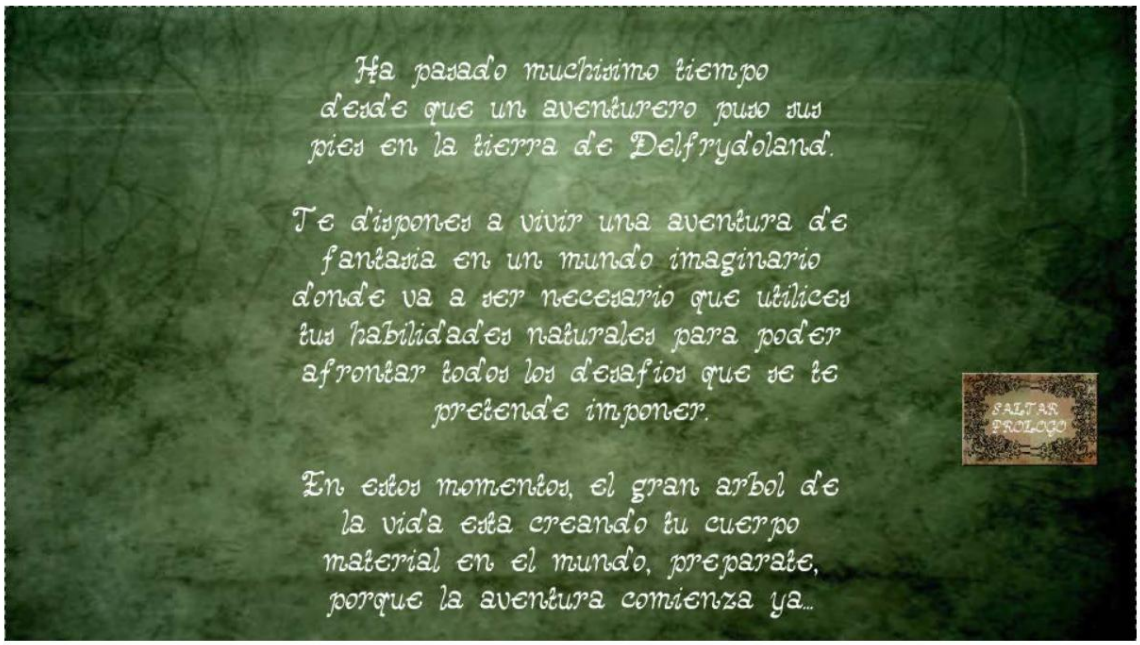


Figure 73. Screenshot of the prologue screen from the editor.

Source: self made

This screen also functions as a menu in itself. It is a temporary screen with an animated text going up, when the text finishes going up, the game is loaded.
play. It also offers the possibility to skip the prologue with a button.

Epilogue screen.

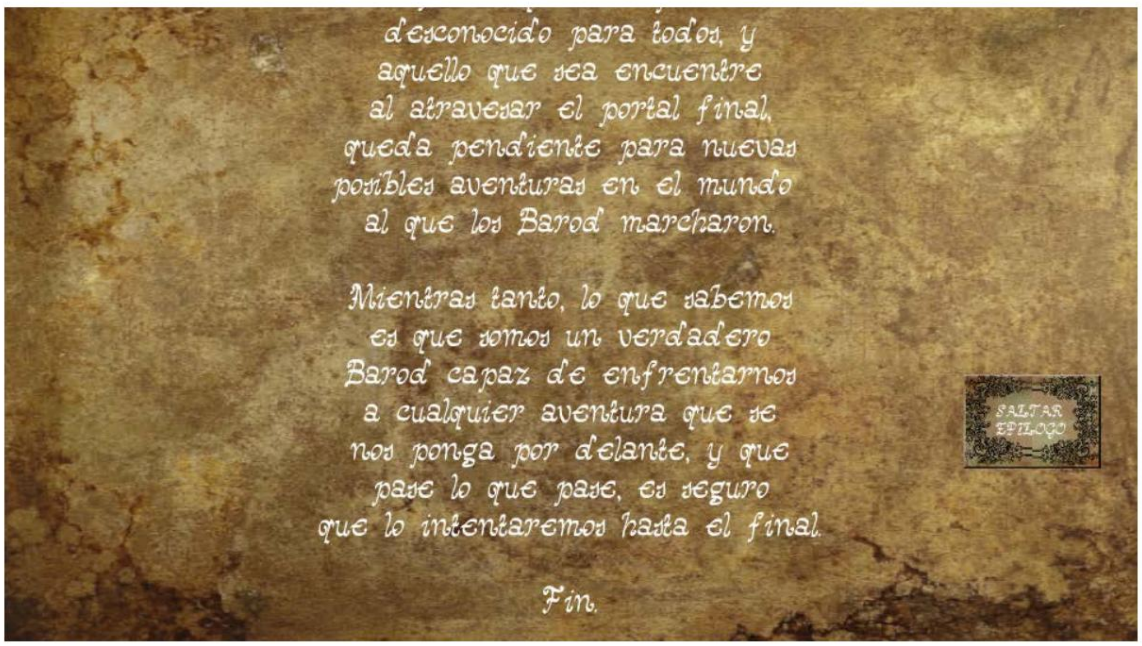


Figure 74. Screenshot of the prologue screen from the editor.

Source: self made

Its operation is the same as with the prologue screen, but with the difference that Through this screen we do not access the game, but the game credits.

Credits screen.

Figure 75. Screenshot of the prologue screen from the editor.

Source: self made

As with the prologue and epilogue, the credits work the same way, but they point to the main menu.

5.1.5.3. Camera

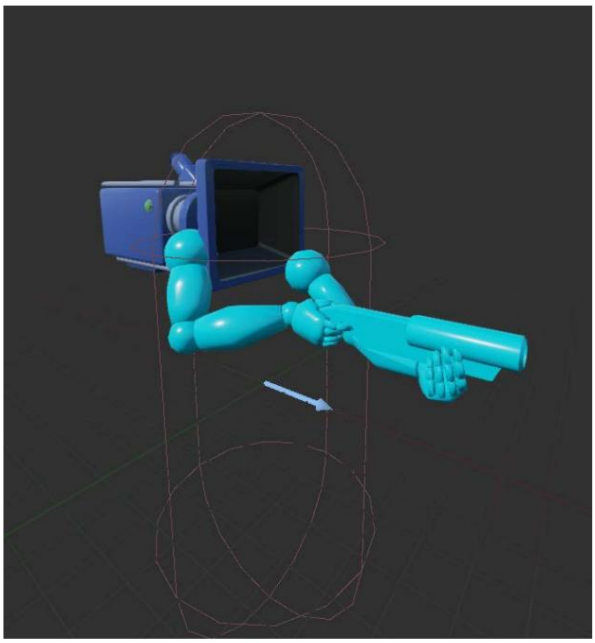


Figure 76. Capture of the character from the editor showing its components.

Source: self made.

Arbennig's camera is a first-person only camera. corresponds to the vision
of the character within the game, representing the world from his point of view.

In addition, it is the only camera that we have in the project, since the rest of the components
they are menus and cinematics that are shown in 2D on the screen.

5.1.6. Sound

The audio in the video game is intended to accompany both menus and cinematics and game levels.
We will divide it for convenience into music, sfx, and UI (User Interface), so with music
we refer to audio that are more like background songs that play constantly, and SFX and
UI are sound effects, although the SFX can be player actions (they sound like the ones in
UI by not having a position in the world) or not (they sound in some position in the world, and how much
the closer we get to them, the more they sound).

Regarding the music, depending on whether we are in a menu, a cinematic, the level
game, or extras, one song or another will play in the background on a loop.

The SFX and UI, as described, will accompany the different components of the world,
character, and also to the player's actions through the menu (when pressing buttons on them
or hover over).

5.1.7. help system

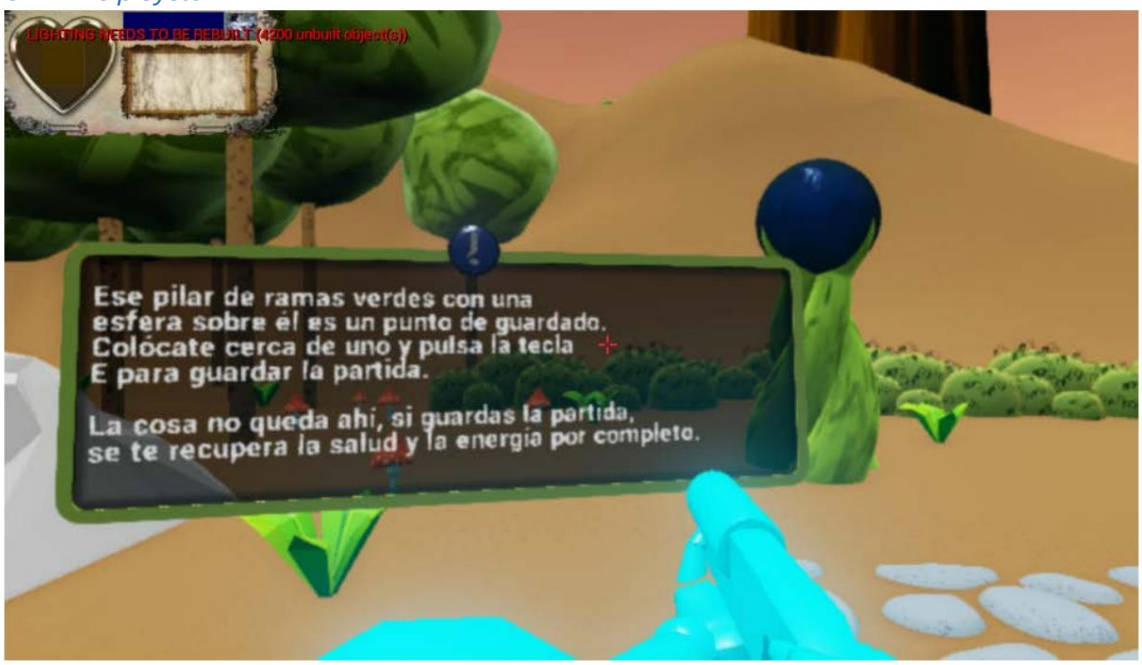


Figure 77. Information point capture activated.

Source: self made.

In order to assist the player when starting the game, the starting island contains a series of informative help messages, which appear closed, and are displayed when the player get close enough to them. These messages provide the player with relevant information about the game controls, and some tips to play.

Figure 78. Information point capture without activation.

Source: self made.

5.1.8. Artificial intelligence

5.1.8.1. NPC AI (Tree of Life, Elderly Villagers, and Final Island Totem)

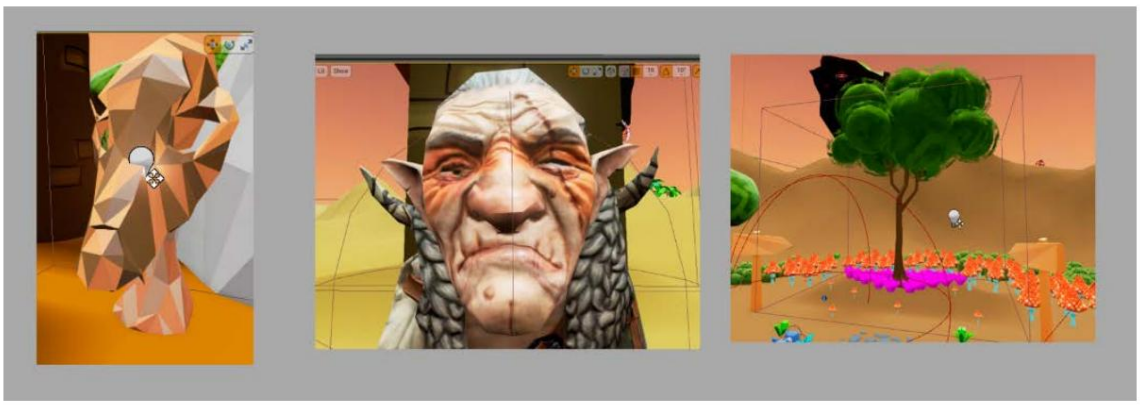


Figure 79. Totem, villager looking like an old man, and tree of life from the editor.

Source: self made.

Regarding the tree of life, old-looking villagers, and the final island totem, his only
game logic consists of the possibility of being able to start a conversation with them.

When the player interacts near them, a dialog window opens displaying the NPC's message and one or more responses from the player.

These characters do not perform any other type of action, they always appear still and do not patrol.

5.1.8.2. AI of the guardian totems of the runes.

Figure 80. Totem capture from the editor.

Source: self made.

Rune guard totems have almost exactly the same functionality as NPCs
normal, with the addition that by talking with them, it is possible to accept the challenge they have prepared (if they have one), and once finished, they give us the rune they keep. So, after this, we stop being able to talk with them, since their soul leaves their body having
delivered the rune.

5.1.8.3. Golem AI.

Figure 81. Capture of the golem from the editor.

Source: self made.

The Stone Golem, which appears in the sixth rune challenge, works as follows:

1. Stand still while unable to see the player.
2. When he sees the player, he chases him around the map.
3. When it reaches it and collides with it, it becomes still again.

Its operation is like this since the conditions of the challenge of rune 6 consist in that the player must run to a flag that is at the end of the island, and after this,

take her to a place indicated by the totem. The path is full of golems, who upon seeing the player, they will chase it, and if they touch it, the player loses the challenge and has to try again.

5.1.9. Technical Guide[25]

Some technical details about the game's hardware, software, and architecture requirements are described below.

5.1.9.1. Hardware

Regarding the hardware necessary to play, it is recommended to have the following minimum:

Memory RAM: 8 GB

Processor: Intel or AMD Quad-Core type

Graphics Card: Compatible with DirectX11.

5.1.9.2. Software

Windows 7 or 8 with 64-bit architecture is recommended.

5.1.9.3. game architecture

The architecture that the game follows corresponds to the one already explained in the Framework section. theory or State of the art (2.2.4). That is, the architecture on which the Unreal Engine is based

4.

5.2. Development and implementation.

We are ready to explain the process of making the video game that we have designed previously in the GDD, where we proceed to explain each of the aspects developed, emphasizing them more or less depending on the most relevant aspects that these contain.

5.2.1. Project Creation

The first of all the steps is the creation of the project and its corresponding set up.

In our case, given the compatibility it offers with Oculus Rift DK1, we created through the Epic Games Launcher a blank Blueprints project with version 4.6.1, since in later versions it has been personally verified that this does not work correctly Oculus DK version. Once created, we can access it quickly from the launcher.

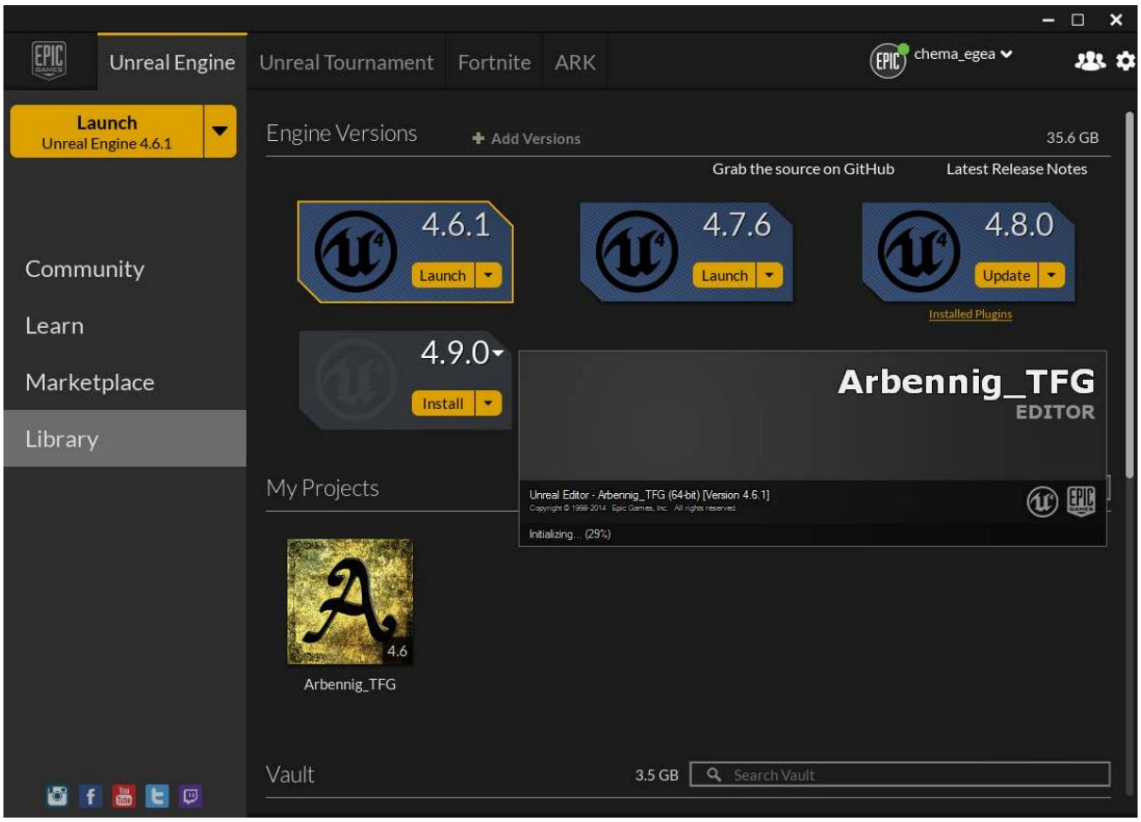


Figure 82. Game launch screenshot from the Epic Games launcher.

Source: self made.

Once the project has been created and having accessed it, it is necessary to configure it, we create from the editor the classes that we will need to be able to setup the project later.

The basic classes that we are going to create are:

- A GameMode class, where we will establish the rules of the game.
- A HUD type class (which is different from the HUD that we will have visual, since this class HUD works with code, and is where we will render the target).
- A PlayerController class that will help the player to own the character principal.
- A Character class, which will have the PlayerController (although not by default, so that it is only during the game when we control the character, and meanwhile, we have a free view that does not react to player events, only to those that we indicate with the mouse to the menus). We create Character and not a basic Pawn because with Character we already have humanoid character functionalities that will facilitate our development.
- A PlayerCameraManager class, which we will assign to the PlayerController to have the control of a camera created by us.

Once these classes have been created, we access the Project Settings of the editor, and we modify the values that we consider necessary. In our case, we have modified (we must take into account Note that we already have levels and characters created, which in this implementation guide is defined its implementation in later points):

Description, adding the pertinent information about the creator, project data.

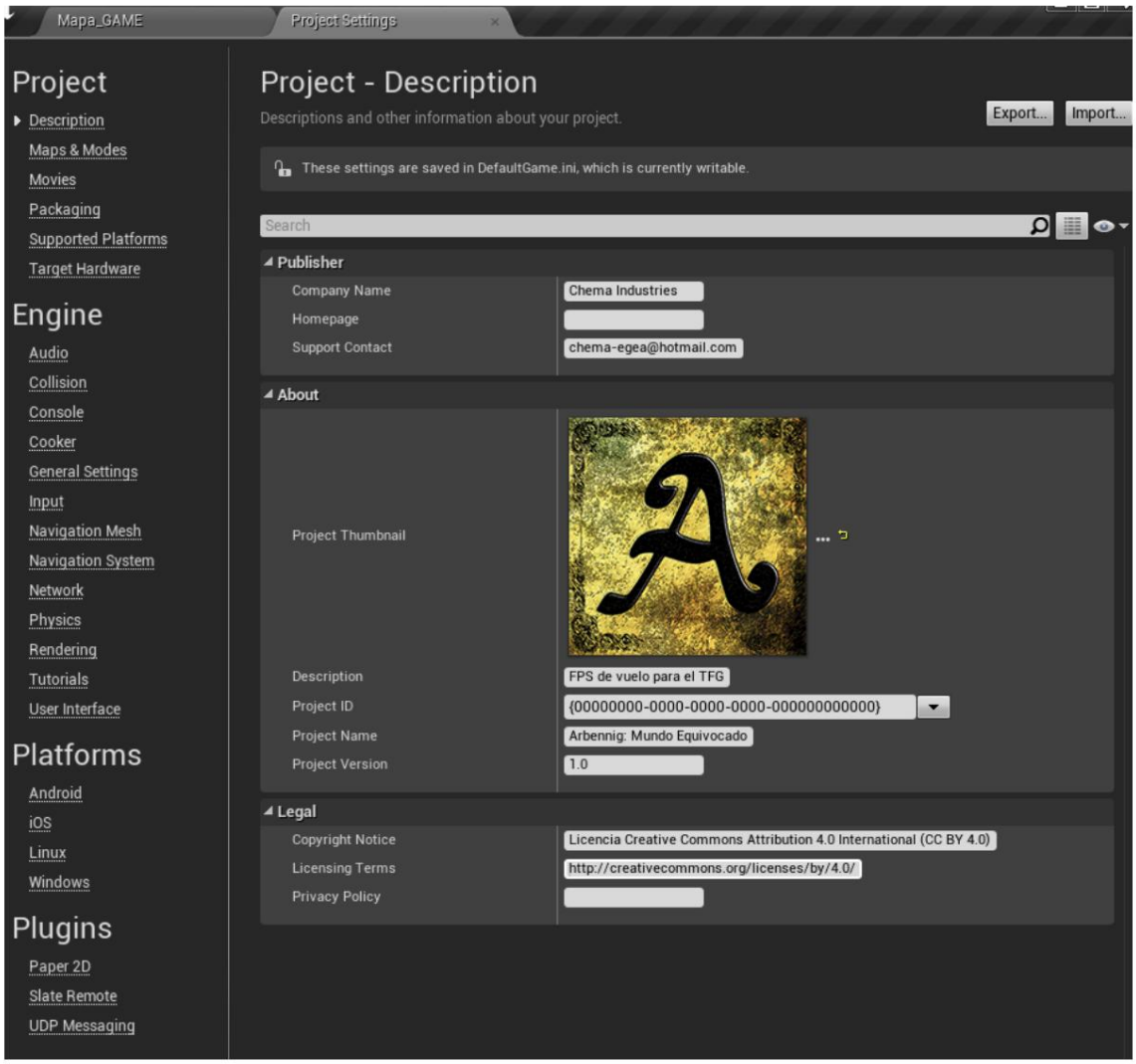


Figure 83. Capture of the modifications added in the description section for Arbennig.

Source: self made

Maps & Modes, selecting the map that is loaded by default when loading the game (game map) and the editor (main menu map), and the game modes, described in the theoretical framework or state of the art section (2.2 .4), in which we indicate our GameMode class, if we have some default Pawn class (in our case we have not assigned, and ingame we have a Character), the HUD that we have created, the PlayerController class, and finally, the class GameState.

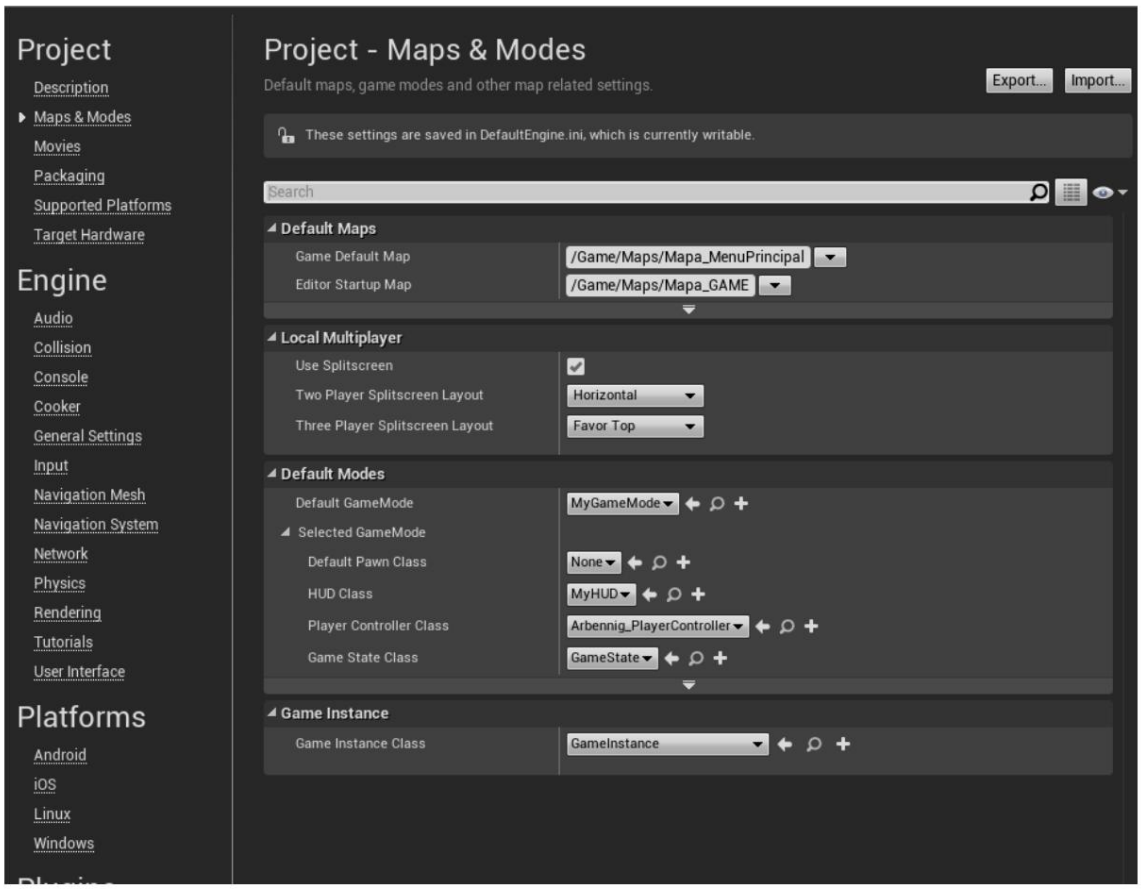


Figure 84. Capture of the Maps & Nodes configuration for the Arbennig project.

Source: self made.

Supported Platforms, selecting that we want to produce the game only for Windows.

Audio, where we choose the default sound class that we are going to use, in our case we have created the class BP_Master_Sound (5.2.4).

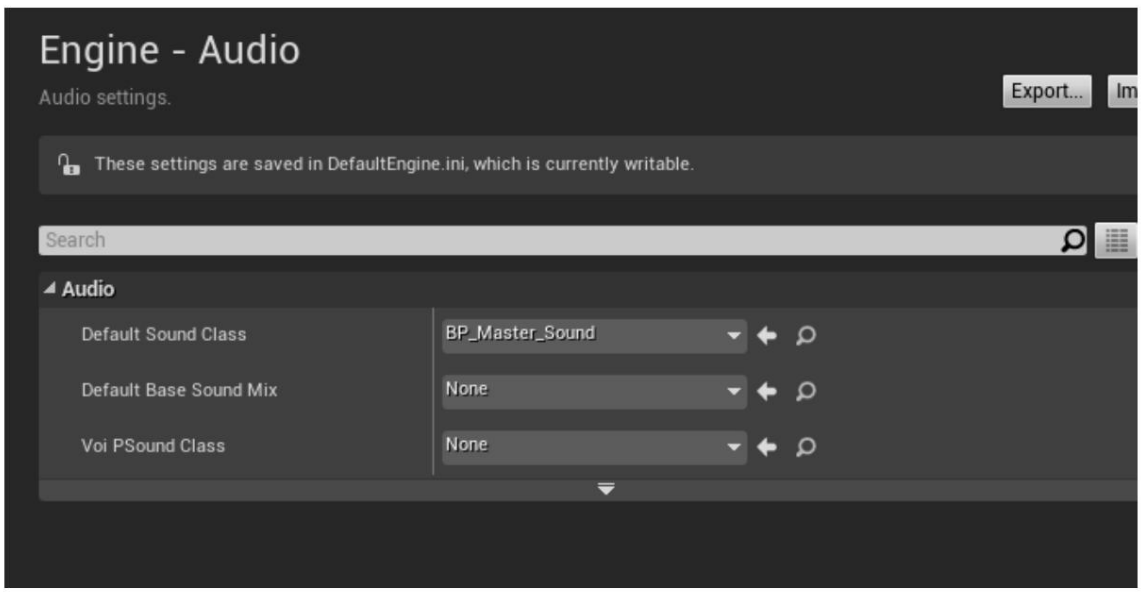


Figure 85. Screenshot indicating where to select the game sound class.

Source: self made.

Input, where identifying events have been added for the game controls, that is, it has been added a list of definitions for controls so we can access them later
from the Graph Editor of blueprints by the name that we indicate. It has been decided to include in this set aside the controls for an xbox control, anticipating possible future improvements that allow its use.

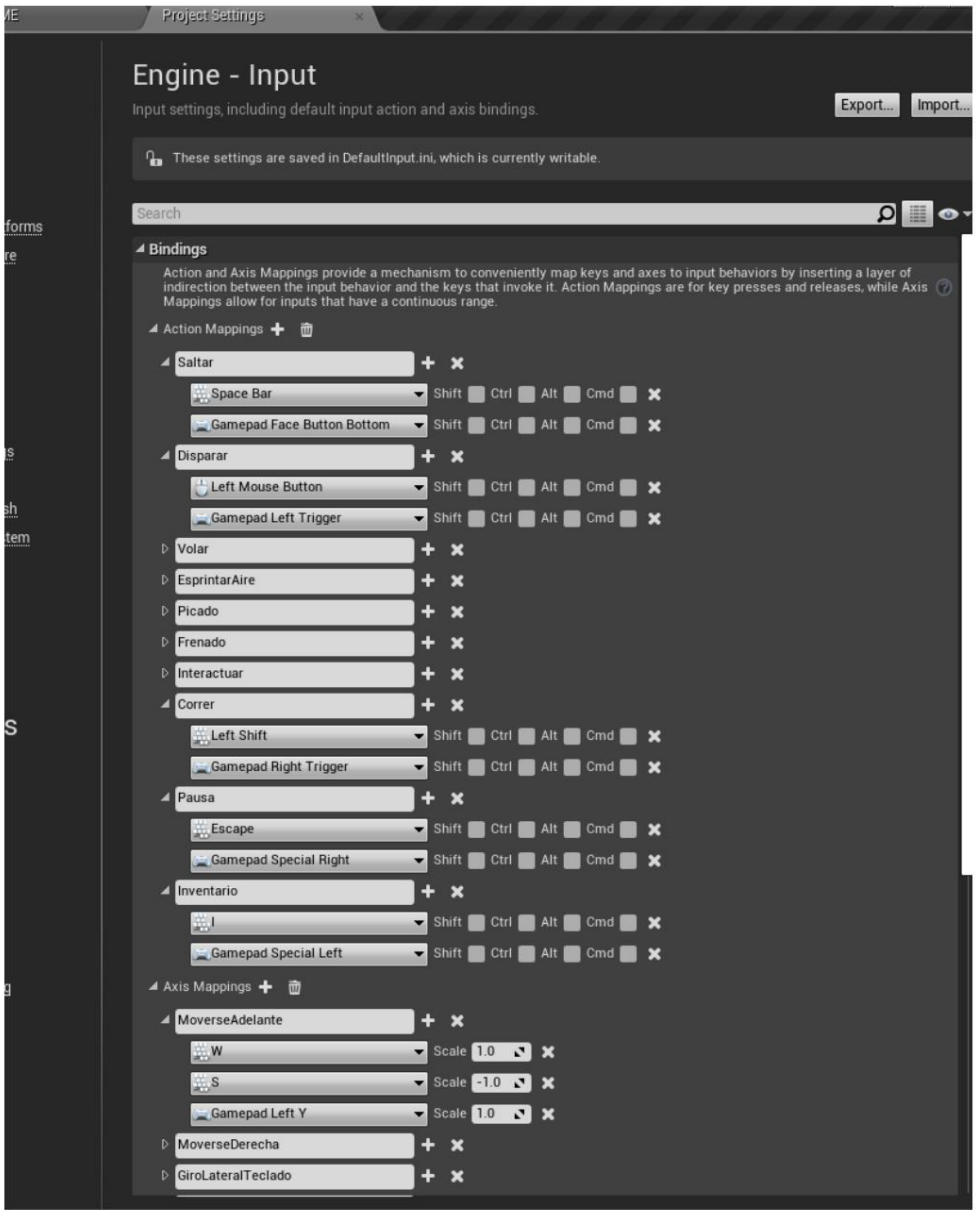


Figure 86. Capture of the Inputs configuration of the project.

Source: self made.

It is worth mentioning for the input section, that we have Action Mappings values, and Axis Mappings values. The difference between these is that with Axis Mappings you save also a value of type float, since they are for movement, which makes it easier to save if we want that as long as the player presses a key, is, for example, ascending, we simply go adding the float to the player's position.

Also, it should finally be noted that for the values of axis mappings, it is not convenient make a definition for each of the character's movements, but we can just set your float to negative. For example, if the character moves forward with W, and this has a float of scale 1.0 assigned, we add to this action of moving forward, that if you press the S key, its float will be -1.0, and with this, we already have defined the control towards front and back.

Figure 87. Motion control definition capture forward and backward.

Source: self made.

It is also necessary to assign our PlayerController a PlayerCameraManager class that allows us to allow you to have control over the game camera.

Simply in the PlayerController definitions, we choose the created class.

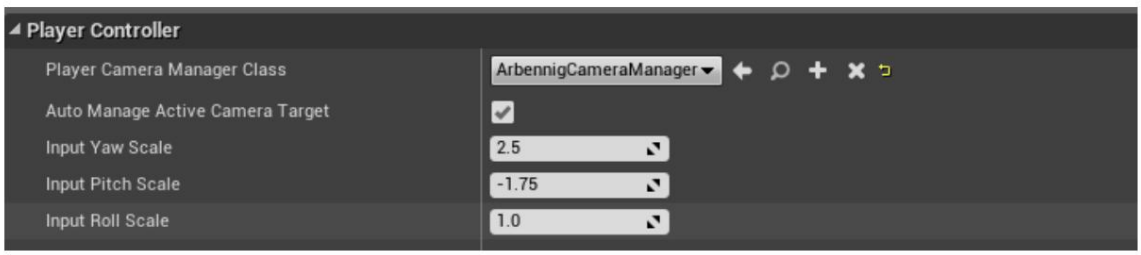
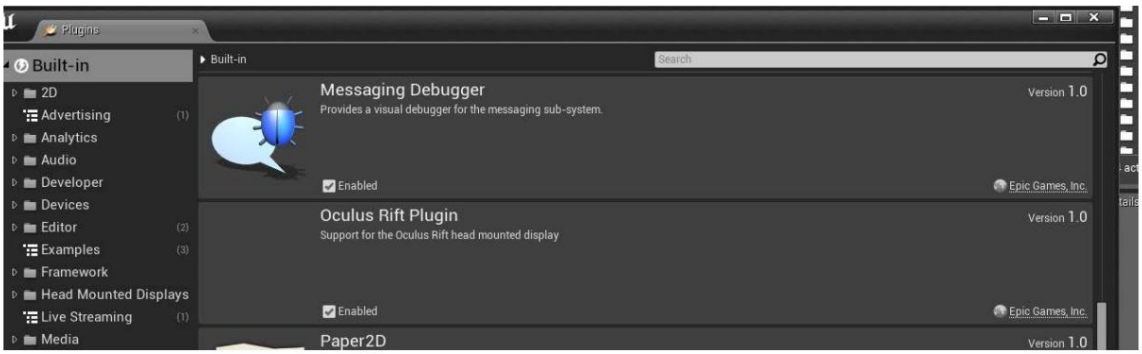


Figure 88. Capture of the created PlayerCameraManager class assigned to the PlayerController.

Source: self made.

Finally, in the plugins window that we find in Window, we activate the oculus plugin rift, which is included by default in the engine.



5.2.2. Implement the actions of the character.

Next, we proceed to explain the blueprint for creating the movement control of the character based on their status changes (Ground, Flight, and Racing). The control blueprint we have chosen to do it in the Character class, in our case, Arbennig_Character, and it is this class probably the most extensive in the entire game along with the Level Blueprint, it is so extensive that by zooming out from the Graph Editor, we cannot see more than a small part of this blueprint, so it has been decided to explain the most important steps relevant in terms of its creation, but not in full detail, since the extent of this memory would almost double.

That said, we should know that since the Character class provides an enum with different states, including Walking (the state we will use for ground), Flying (the state that we will use for flight), and Falling (we will use it for when the player decides to do the plummeting). But to help us with the development, we are also going to create a couple of listed, one for states of ground, which will contain Walking and Falling, and one for states of air, which will contain Flying, and Racing (and we add those listed as variables to the character to be able to use them). The flow of change of these states is as follows:

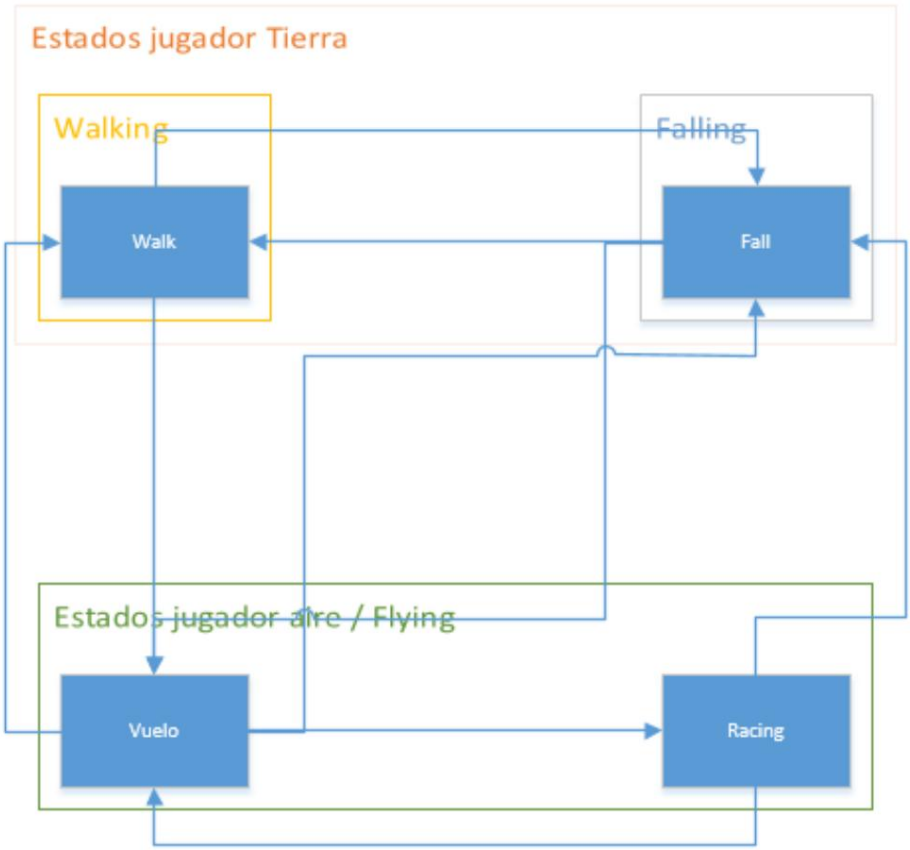


Figure 89. State change diagram.

Source: self made.

As you can appreciate:

- From Walk we can go to the Flight and Fall states.
- From Flight we can go to the Walk, Fall, and Racing states.
- From Racing we can go to Fall and Flight.
- From Fall we can go to the Walk state and the Flight state.

It is necessary to know that so that we can have functionality of the different states of the class Character, what we must do is in its initialization parameters, check the boxes of Can Fly, Can Jump, y Can Walk, en su Movement Component.

Figure 90. Capture of the variables that allow the player to switch between the default movement states.

Source: self made.

5.2.1.1. Camera rotation.

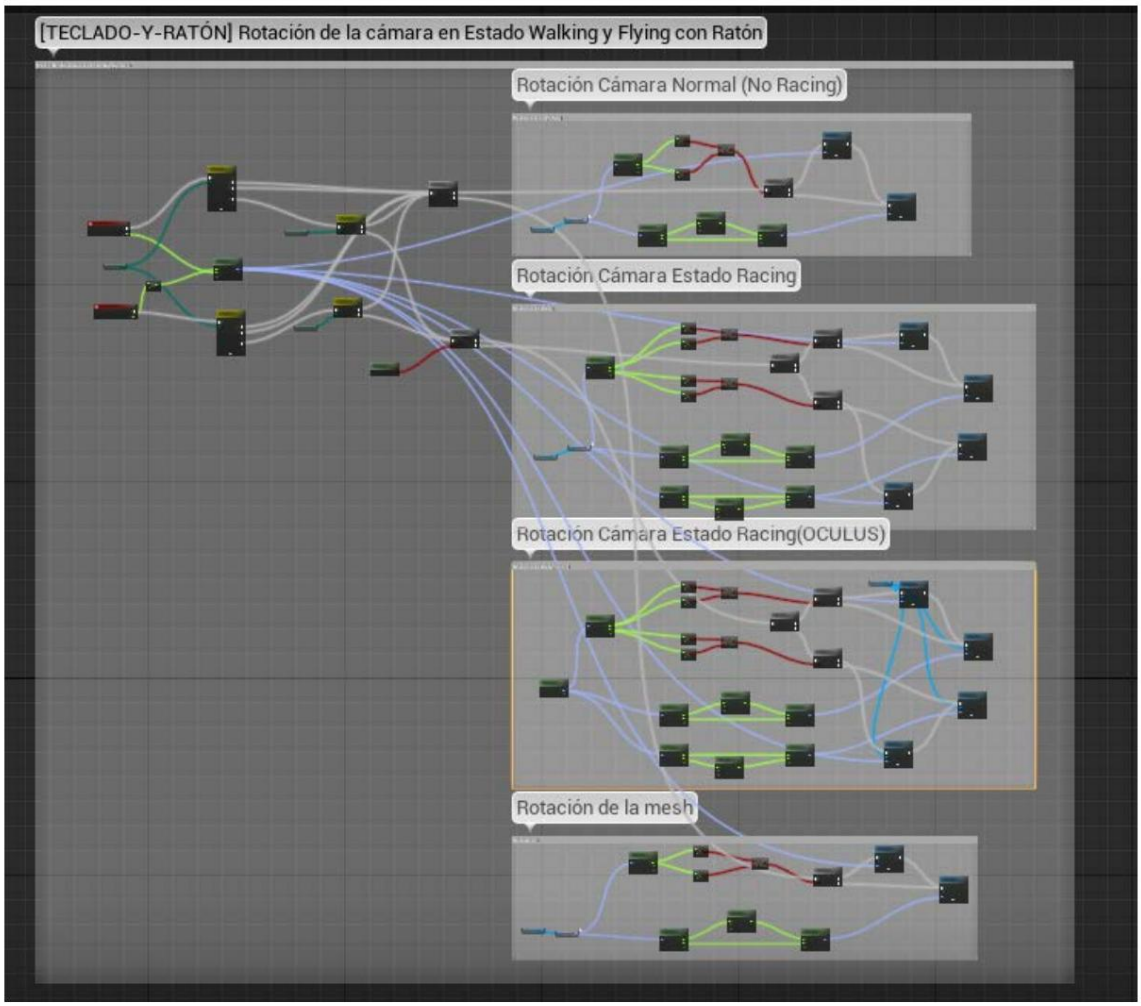


Figure 91. Overview of the portion of the camera rotation graph.

Source: Own elaboration.

The player's rotation has been divided into two parts, on the one hand, the camera, and on the other. another, the mesh it has. This has been decided by the Racing control, where you can move freely with respect to the body.

For the implementation of the camera rotation, we first make a series of checks initials to determine the player's movement state, and also, whether or not they are in an oculus connected. Also, we create a rotator (vector of rotations) with the mouse input inputs, so that we can add them later to the camera and mesh rotation.

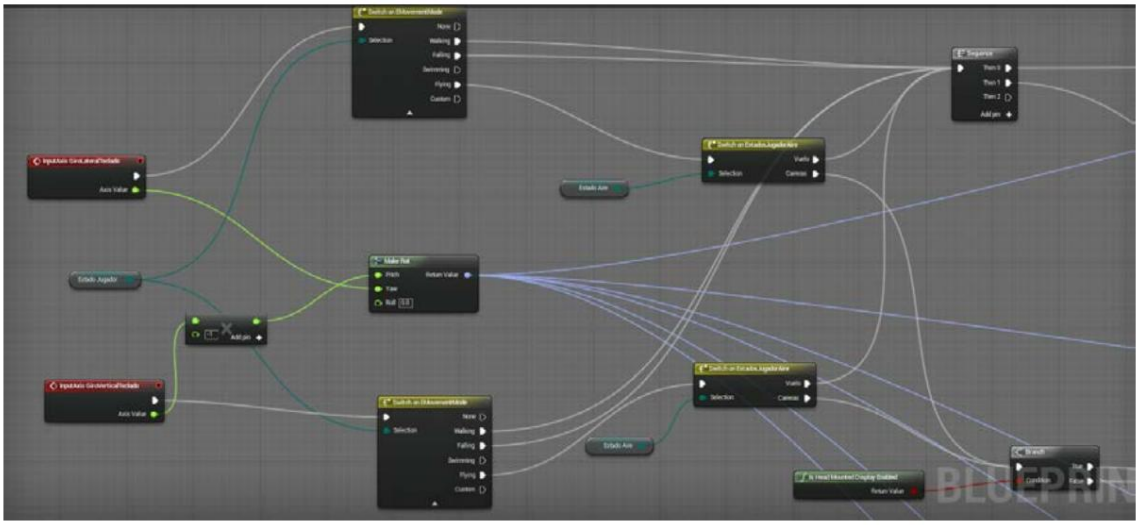


Figure 92. Capture of the portion of the graph of initial checks before proceeding to rotate.

Source: self made.

After this, if we are not in Racing mode, the camera rotation is applied considering a clamp of the angle we are in so as not to exceed a certain amount and rotation problems arise when trying to exceed the 90º limits. And finally, the rotation is added if we are not exceeding any limits, and after that the rotation is set again to avoid a camera crash issue that arises in Unreal due to trying to apply pitch and yaw rotations at the same time, this way we fixed it.

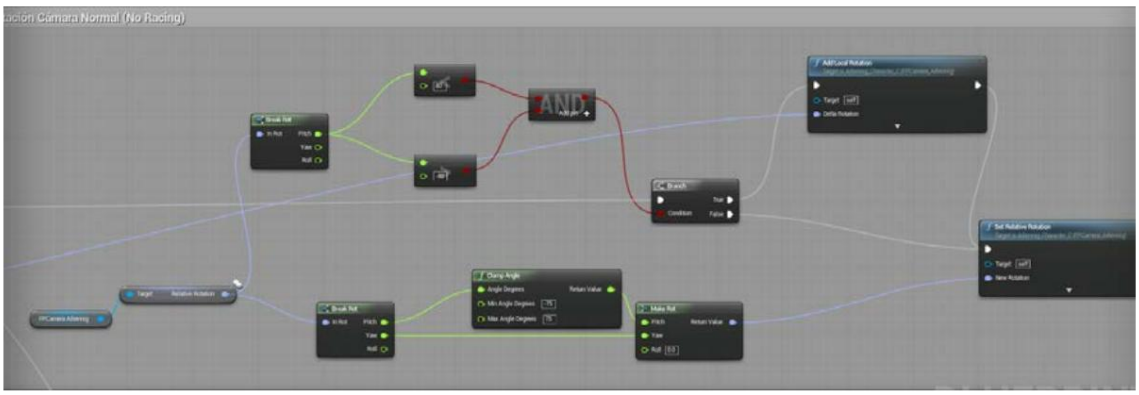


Figure 93. Camera rotation capture when we are not in Racing mode.

Source: self made

As for the rotation of the camera in the Racing state, it is a bit peculiar, since it is designed for compatibility with oculus is correct. When we enter Racing mode, the camera is detached from the player's body, being free so that we can visualize the environment while flying in a particular direction. Despite this, the difference with the camera What is not Racing, is that here we limit both the angle of the pitch (up/down) and the yaw (to the sides), because the character's head cannot be turned backwards.

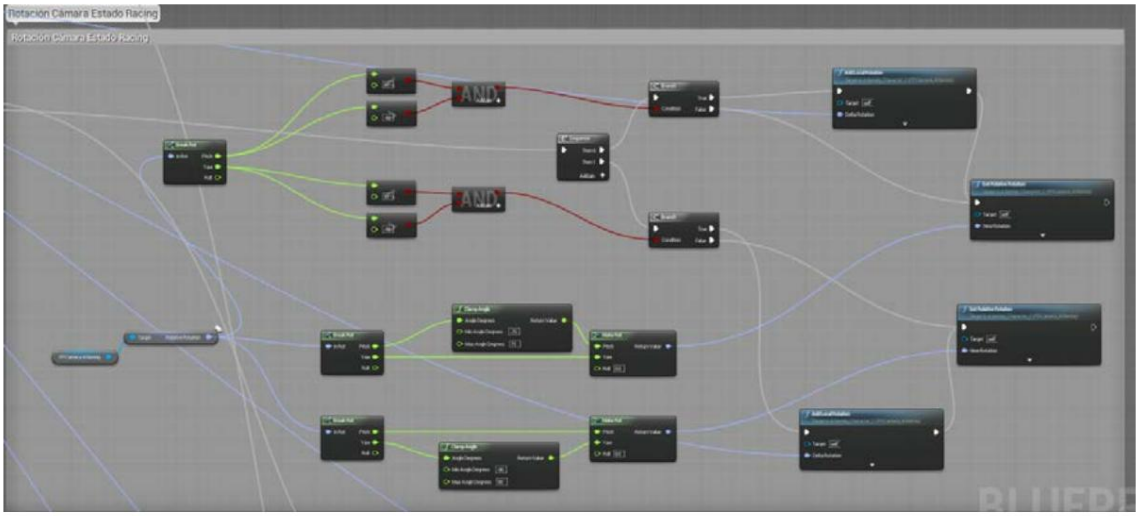


Figure 94. Camera rotation capture when in racing mode

Source: self made

For the oculus camera rotation in racing, the main difference is that we get the orientation from the peripheral instead of directly from the camera we have, the rest of steps are the same.

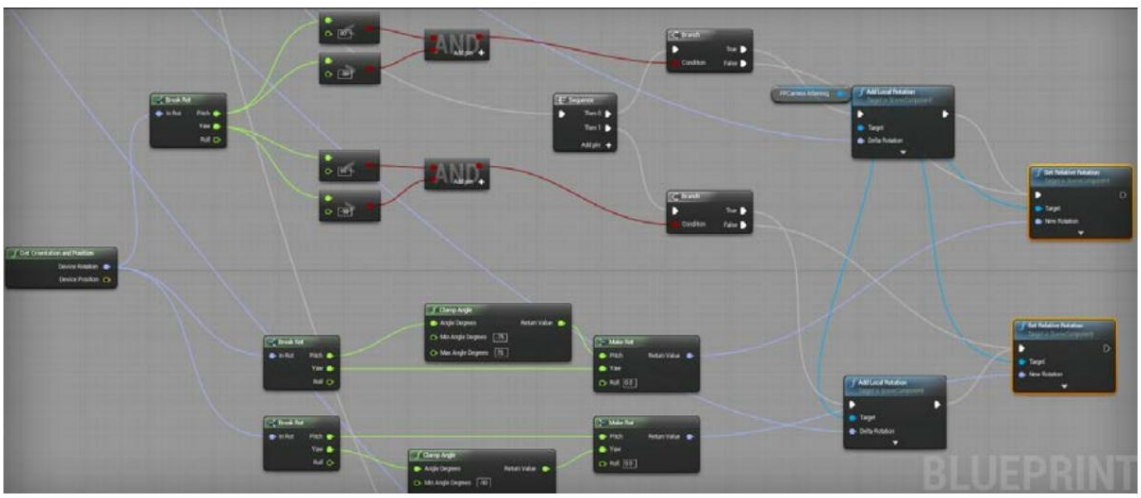


Figure 95. Camera rotation capture when we are in Racing mode and oculus activated.

Source: self made

Finally, regarding the rotation of the character's mesh, it will rotate the same as the camera rotates when we are in a state other than Racing. Since in Racing it will turn with the character itself (remember that the character is a set of a mesh and a camera, so we have a character entity, which can rotate and move all its components, and on the other hand, we can access each of its elements separately and make the necessary modifications).

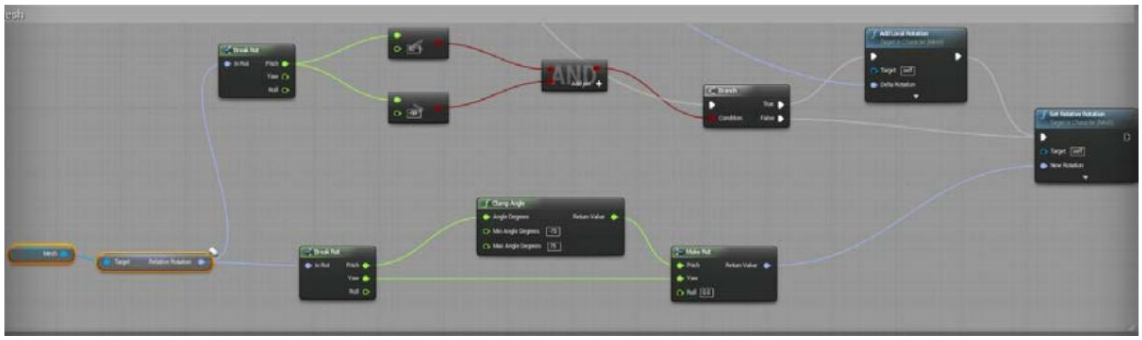


Figure 96. Capture of mesh rotation code portion.

Source: self made

5.2.1.2. Character movement.

The movement of the character is determined depending on the state in which we find ourselves, since the operating logic of the Racing state is considerably different from that of the rest of states, using in the rest the WASD keys to move, while in Racing there is an outside that pushes us forward and with WASD what we do is turn the character.

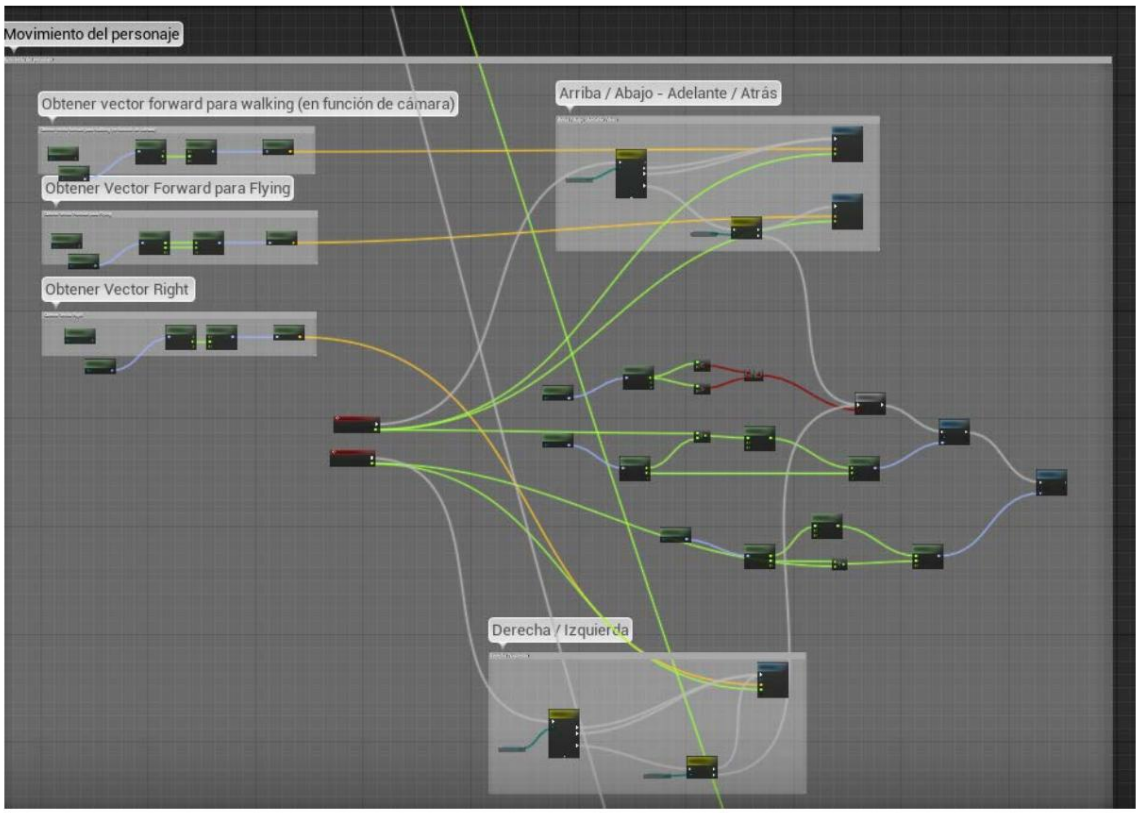


Figure 97. General capture of the character's movement in all its states.

Source: self made

The first thing we do to be able to establish the movement as long as we are not in Racing mode, is to obtain its Forward (for forward and backward movement) and Right (for lateral movement) vectors. These vectors can be obtained as a function of the rotation of the character in the world automatically, since the engine provides us with a function to it to which you only have to pass a rotator.

Since in walking we can only move, let's say, by land, that is, in a space "two-dimensional", while in flight we can also move freely up and down. down, we calculate two different forward vectors.

For the case of the Right vector, we do not need this distinction, since it is the same.

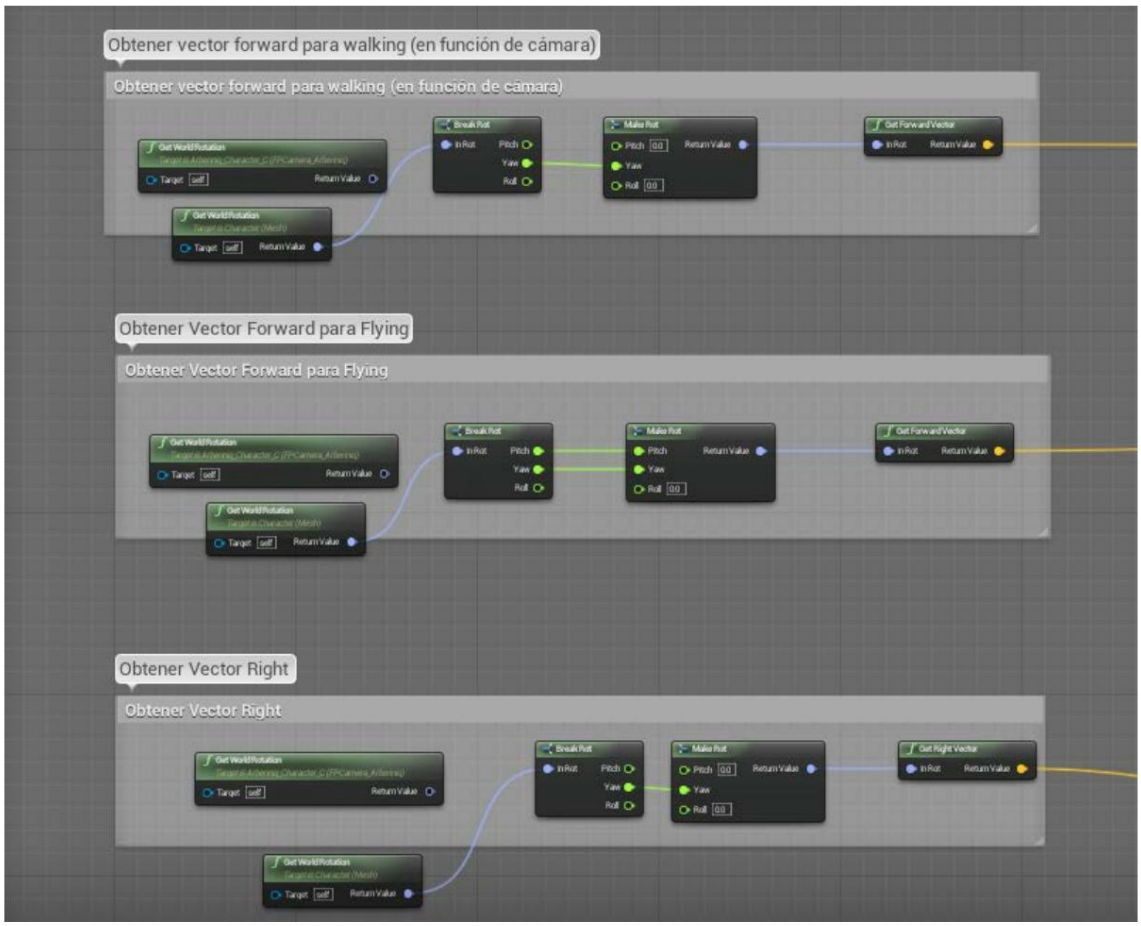


Figure 98. Capture of obtaining the vectors that point forward and to the right to determine the movement.

Source: self made

Once the vectors are obtained, it is only necessary to add to our Pawn the movement in the direction of these vectors, both for movement forwards/backwards, and for lateral movements.

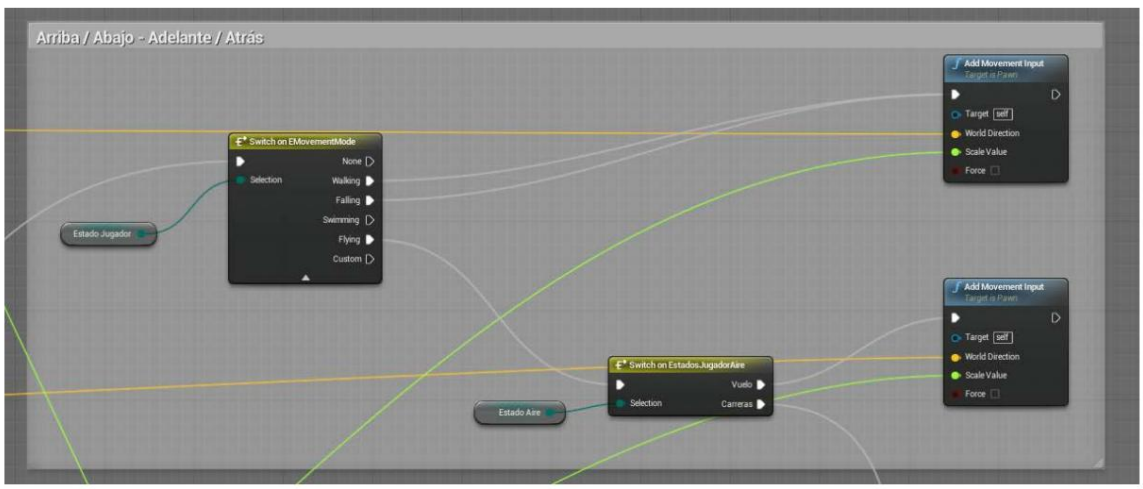


Figure 99. Capture of the forward and backward movement logic in Walking, Falling and Flying(flight).

Source: self made

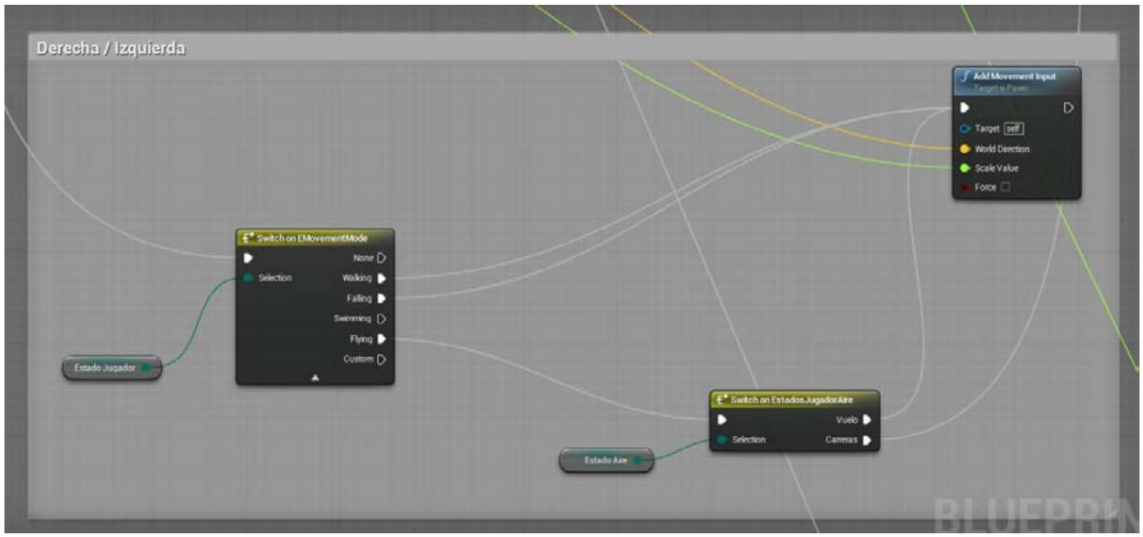


Figure 100. Capture of the movement logic to move to the right and left in Walking, falling, and flying.

Source: self made

On the other hand, for the movement in the Racing state, as described above, it is already applied constantly when entering said state, so that when the player presses any of the WASD keys, what it does is turn the Actor (establishing, yes, as with the turns described above, some angle controls so that they do not exceed certain amounts that give trouble).

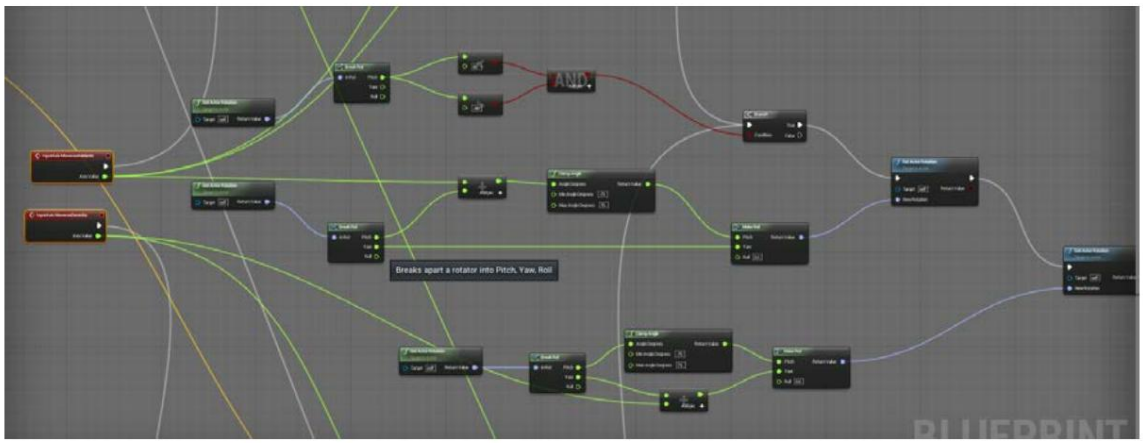


Figure 101. Screenshot for movement in Racing.

Source: self made

5.2.1.3. Ascend/ Descend

The vertical ascent and descent of the character, which occurs only in the flight state, is applied simply adding movement in the Z axis to our character using its Ascent event.

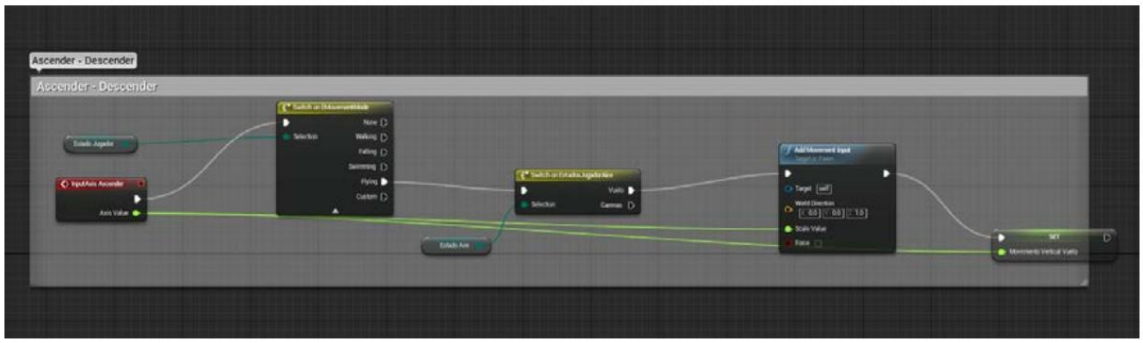


Figure 102. Capture of the logic to ascend and descend in Flight (not Racing).

Source: self made

5.2.1.4. Skip

The jump logic is offered by the character class already implemented, but to apply it to our character, we control that the state in which it is is Walking.

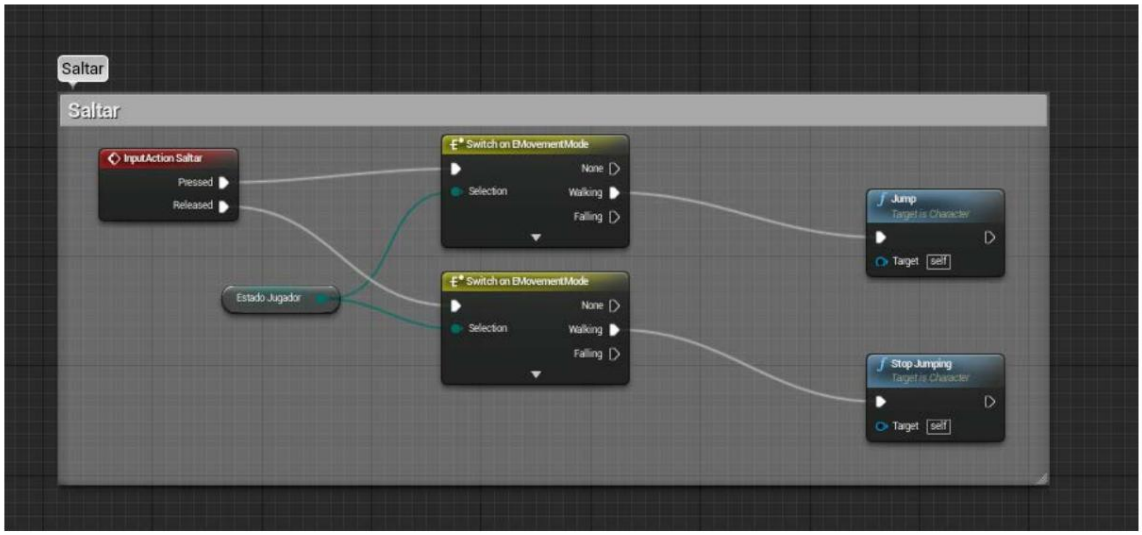


Figure 103. Screenshot for jumping (only in Walking)

Source: self made

5.2.1.5. dry braking

Dry braking, which can only be applied in the flight state, is performed by accessing
to the Character Movement component of the character, which contains properties and functions
related to movement details of the pawn, and we set its speed to 0,0,0.

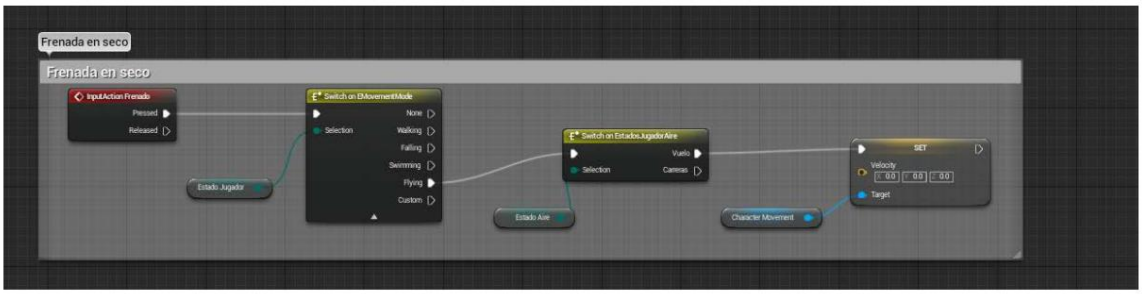


Figure 104. Capture with the logic of performing a dry braking in flight state.

Source: self made

5.2.1.6. sprint correcting

In order for the player to run, we access their character movement component and
We changed his maximum movement speed a little more, from 600 to 900.

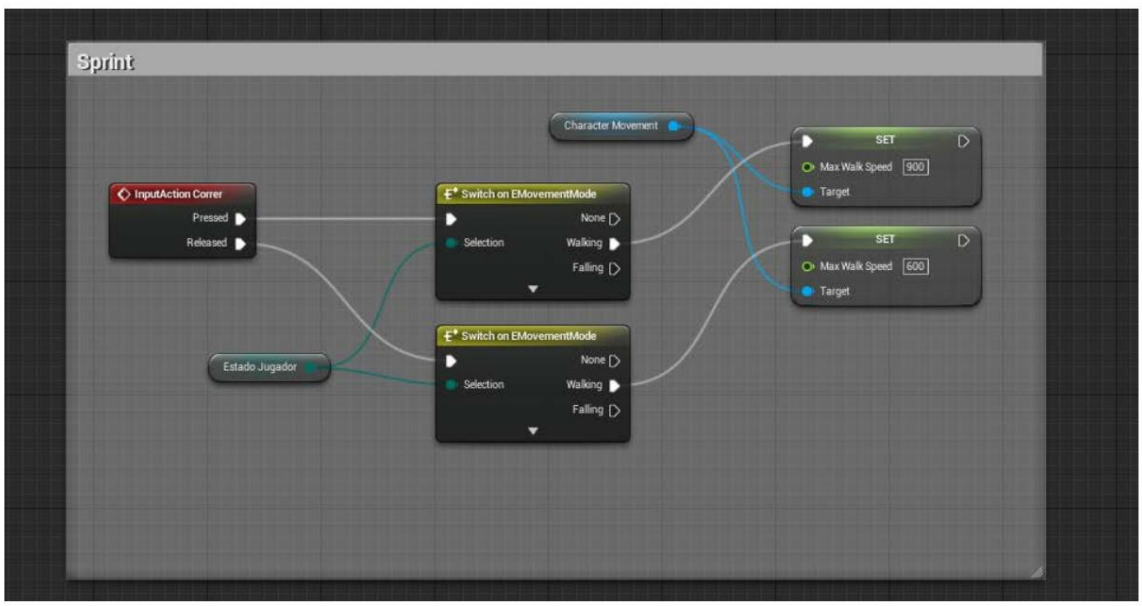


Figure 105. Screenshot with the logic to run in Walking.

Source: self made

5.2.1.7. Turbo in Racing

The logic of the turbo is divided into two parts, on the one hand, the logic when you press the button of activate the turbo, which is only available when we are in the Racing state, and where what we do is that when we press the turbo key and also our energy is not 0, we changed the maximum movement speed to double what the Racing state allows, and we activate a boolean that allows us to manage energy consumption. when the player release the turbo button, it resets the maximum speed of movement of flight and the boolean.

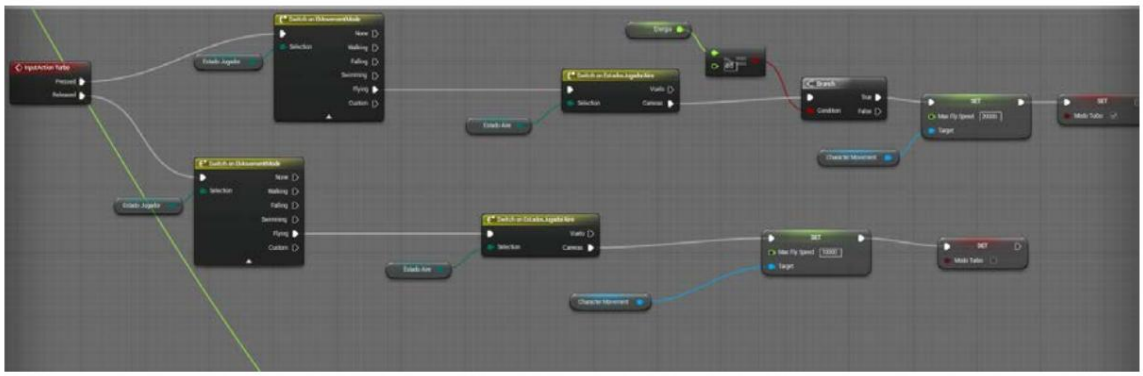


Figure 106. Capture of the logic of performing a turbo in Racing.

Source: self made

In the second part, the turbo, where every moment we control the energy management, we check if the turbo is activated or not, and based on this, we are going to consume energy or recover it every second little by little.

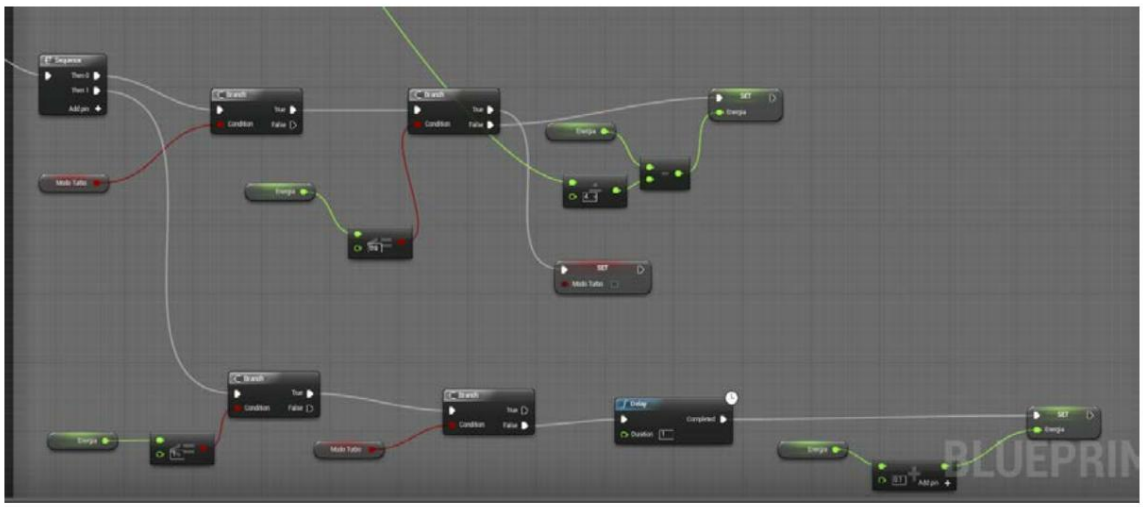


Figure 107. Capture of checks to decrease/increase the player's energy in depending on whether it is in use or not.

Source: self made

5.2.1.8. State change management.

The state changes that the player can make can be manual and automatic,
so if you are walking and want to fly, you can go into that state by pressing a key, but
when he is running and suddenly falls from an island, for example, he automatically switches from
estado Walking a Falling.

5.2.1.8.1. Manual state changes.

These changes, as we have said, are the changes that the player can make by pressing the
buttons or keys enabled for it.

5.2.1.8.1.1. Falling Manual State Changes

The passage from manual state to Falling is possible from the flight state flying and also
from Racing.

To do this, we simply detect that the player is actually in one of those states, and
then we make the change of state and place the gravity to which the object is subjected
character to 1 (it is a normalized value from 0 to 1). When we change to the Falling state while in
a flight state, we can return to it once we release the dive button.

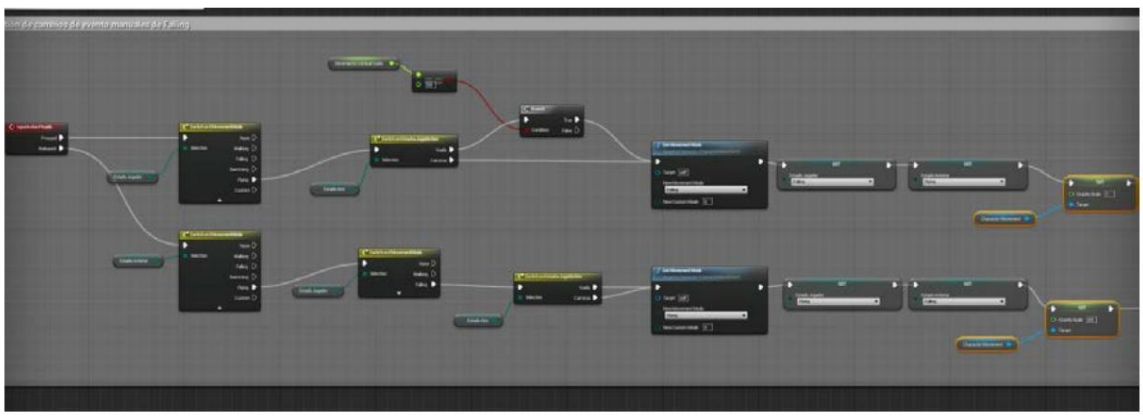


Figure 108. Capture of the manual state changes of the Falling state.

Source: self made.

5.2.1.8.1.2. Manual flight state changes

To make the change to flight status, what we do first of all, the same as in the change
from previous state, is to check if we are in a state that can make the change to fly, and
if so, we do it, and change the gravity the player is under to 0.

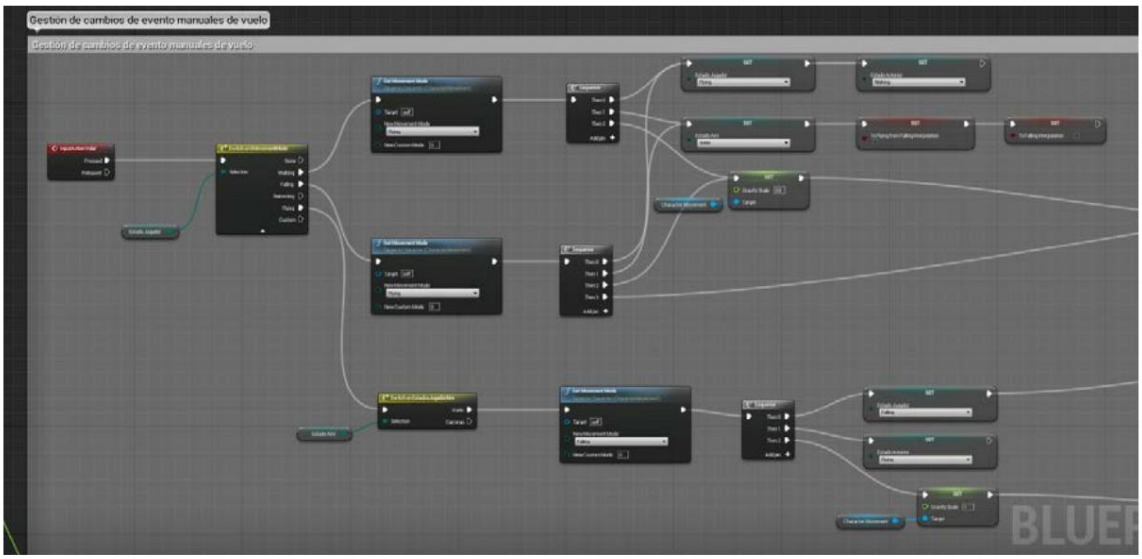


Figure 109. Capture of manual status changes to flight status.

Source: self made.

When we press the change to flight key again, if we are already flying, we go to state of Falling, which will automatically become Walking if we are on the ground, as already we will see in point 5.2.1.8.2.

5.2.1.8.1.3. Racing Manual Status Changes

To make the change of status to Racing, what we do first is, as in the rest status changes, check the status we are in, and if in this case it is flight, we can change

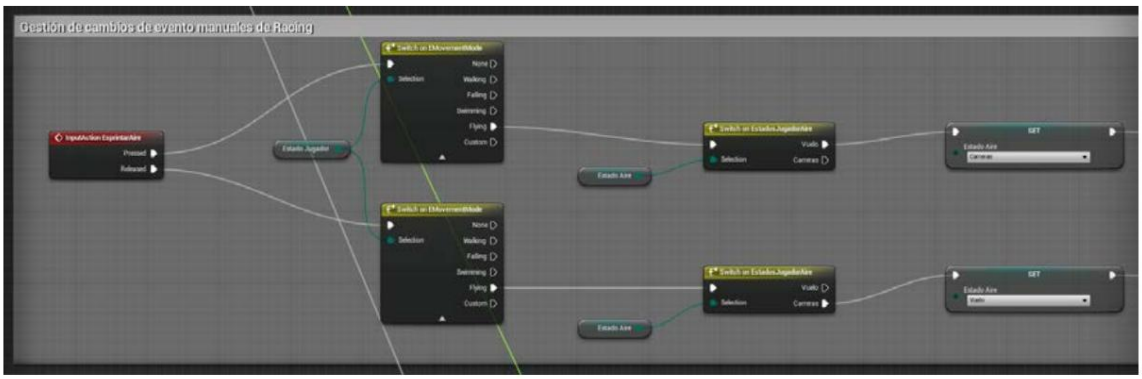


Figure 110. Screenshot of manual event change checks to determine if it can go to Racing state.

Source: self made.

What we do to change to this state is:

1. Rotate the character (the entire actor) to where the camera is facing.
2. Since we will have rotated the entire actor, the camera will also have rotated with it, so we must save the rotation that it had before rotating the actor, and apply it once rotated.
3. Give him a flight speed boost to make him go faster when moving.

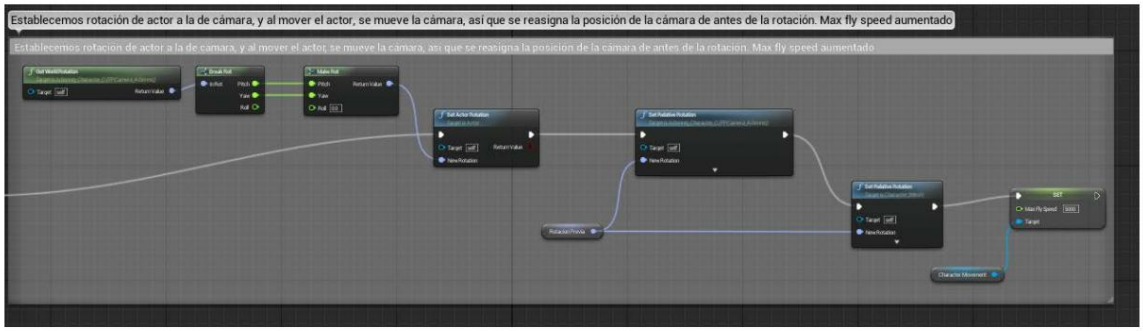


Figure 111. Capture of the logical change when passing to the Racing state.

Source: self made.

4. Constantly moving forward while in this state
   applying a velocity with a certain friction (if we do not include friction, when let's stop being in this state, it will continue to fly always with the same speed driven and will never stop).

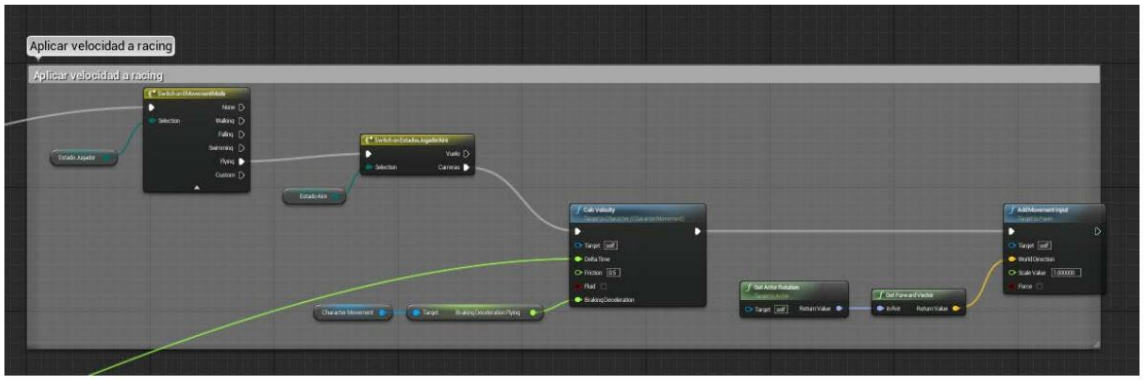


Figure 112. Screenshot of applying a constant speed to the player when entering Racing.

Source: self made.

5.2.1.8.2. Management of automatic status changes

As for the automatic state changes, which, as we have said, are the ones that the player does not you can control directly with keys on your controllers.

5.2.1.8.2.1. Automatic state change management between Walking and Falling

When the character jumps off a cliff, he goes into the falling state, where we turn him 90 degrees and becomes unable to perform actions such as jumping or running. In order to do this, we check if the character is falling by measuring its speed on the z axis, in addition, we check that it is not ascending in the same way (we control in this way that it is not jumping), and also by Finally, we check the state in which we find ourselves.

If the character is walking, we add a delay to determine that there is simply no jumped, and we do the same check again, which, if it meets it, will go to the falling state.

If the character is in the falling state, we check that he is no longer falling anywhere (if has hit the ground), and we make it enter the walking state.

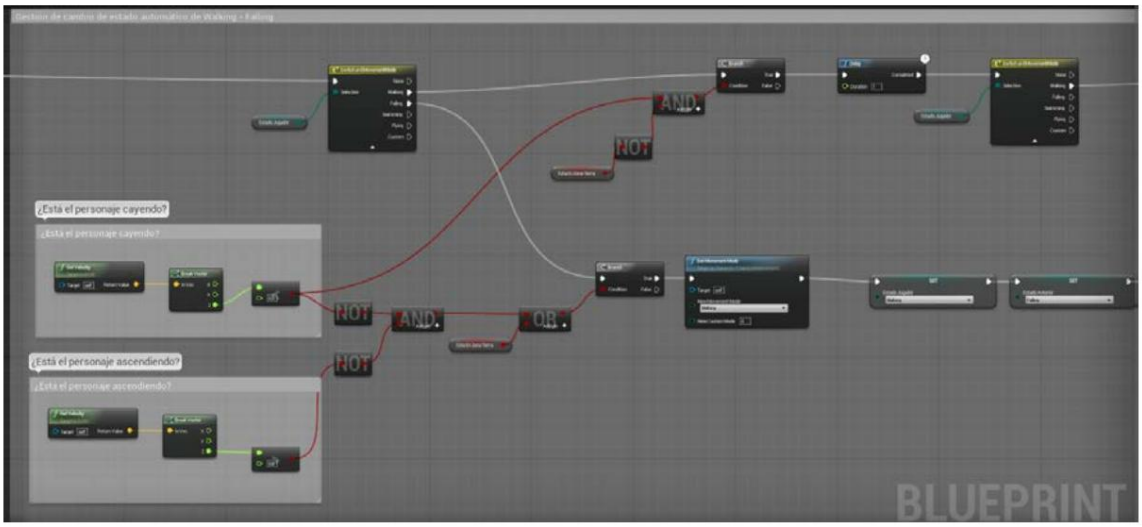


Figure 113. Capture of the management of the automatic change of state between Walking and Falling (part 1).

Source: self made.

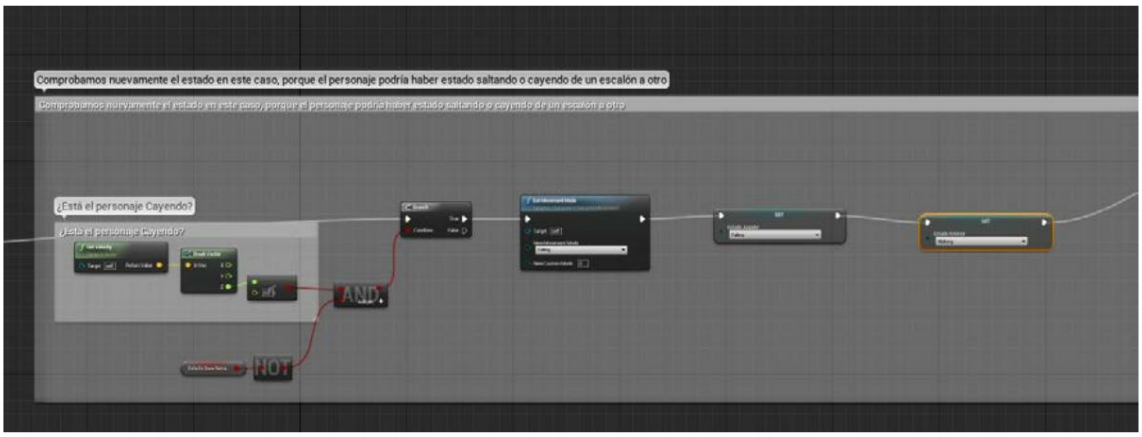


Figure 114. Capture of the automatic status change management between Walking and Falling where we check again that the character is really falling.

Source: self made.

5.2.1.8.2.2. State change tweens

Tweens accompany each state change, and allow us to make a smoother transition between state changes, so that for example, if the player is falling and hits the ground, they meet their actor rotated 90 degrees (the capsule is rotated 90 degrees) and this change made has to be reset (so that the capsule is straight again and thus appears to get up), with an interpolation, little by little the rotation that we want is established.

We have tweens mainly tied to event changes, which are:

Figure 115. Capture of the boolean variables that allow us to pass to the different character motion tweens.

Source: self made.

5.2.1.8.2.3. Automatic interpolation from flying to falling

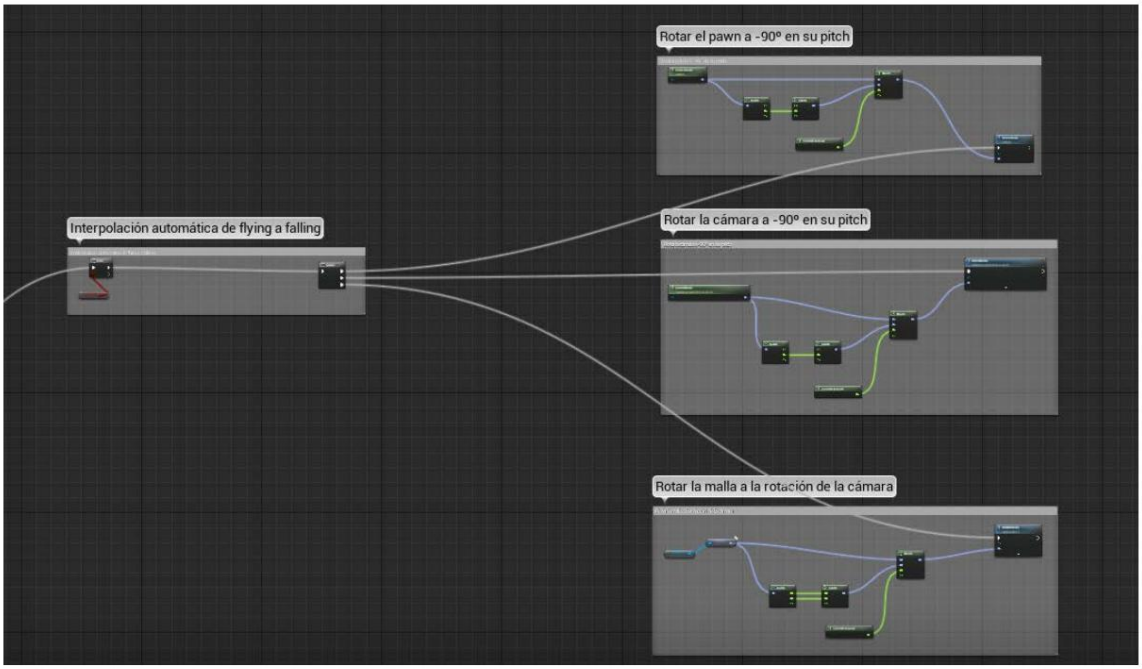


Figure 116. Capture of the general logic of the interpolation to go from the flying state to the falling.

Source: self made.

The interpolation from flying to falling basically consists of the following steps:

1. Check what must be done:

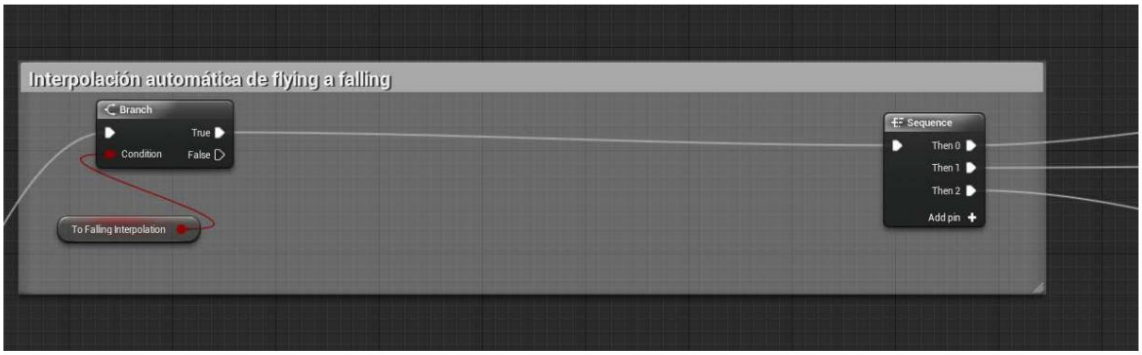


Figure 117. Screenshot of the initial check to determine whether or not to apply the interpolation from flying to falling.

Source: self made.

2. Rotate the Character -90º in its pitch adding interpolation.

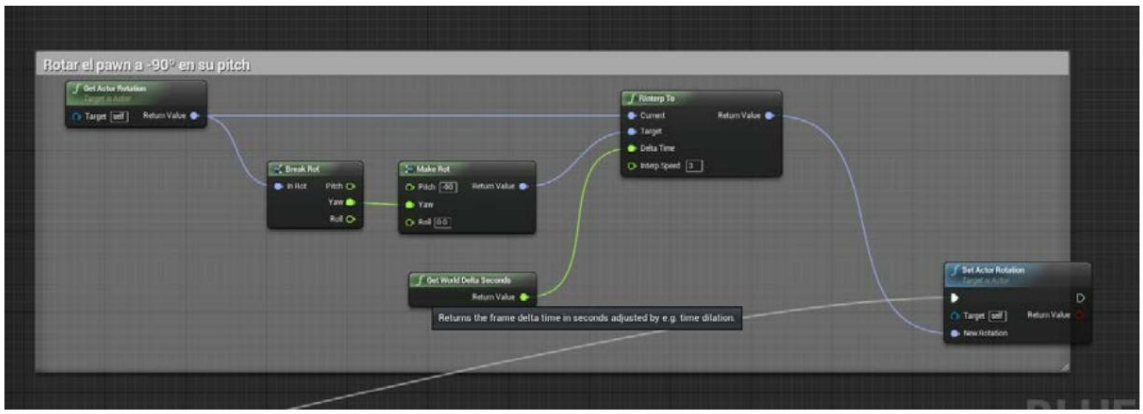


Figure 118. Pawn rotation step capture to simulate falling down.

Source: self made.

3. Rotate the camera -90º in its pitch also adding interpolation (if the actor rotates -90º, we want the camera to rotate -90º its point of view, that even if we turn the pawn, it will follow the camera focusing on the same place)

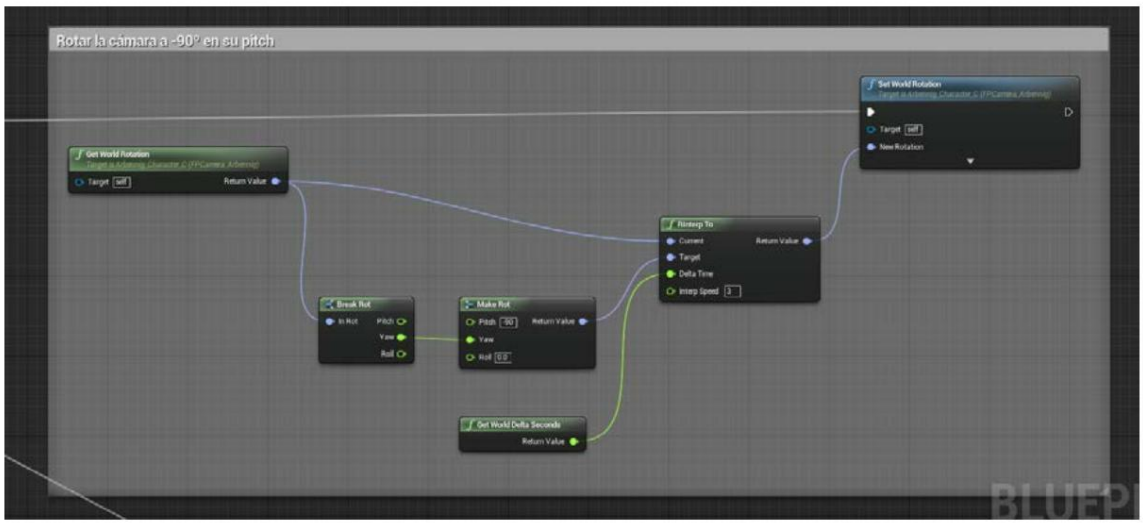


Figure 119. Capture of the camera rotation to simulate that the pawn is falling towards down.

Source: self made.

4. Rotate the mesh to the camera's rotation also using an interpolation.

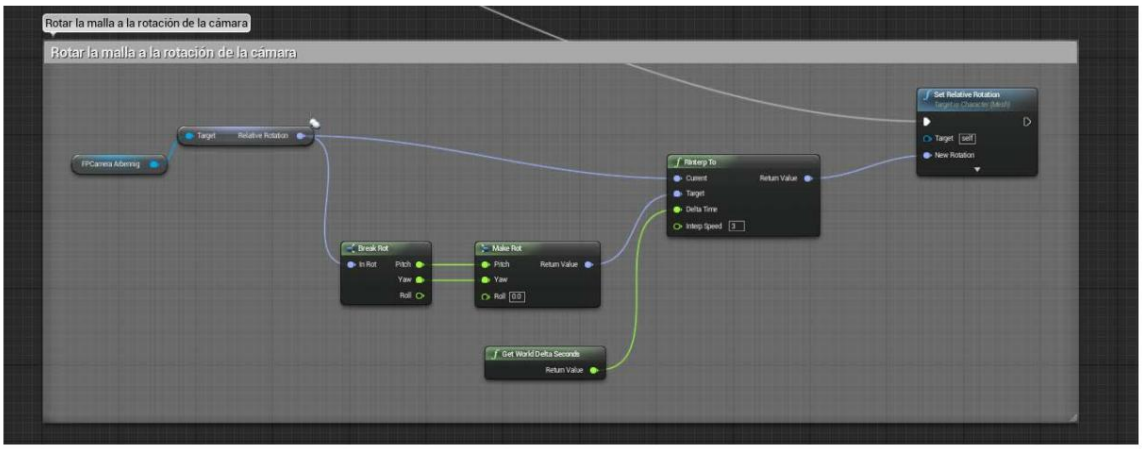


Figure 120. Rotation capture of the rotate mesh to mesh rotation portion of code.

Source: self made.

5.2.1.8.2.4. Automatic interpolation from falling to flying

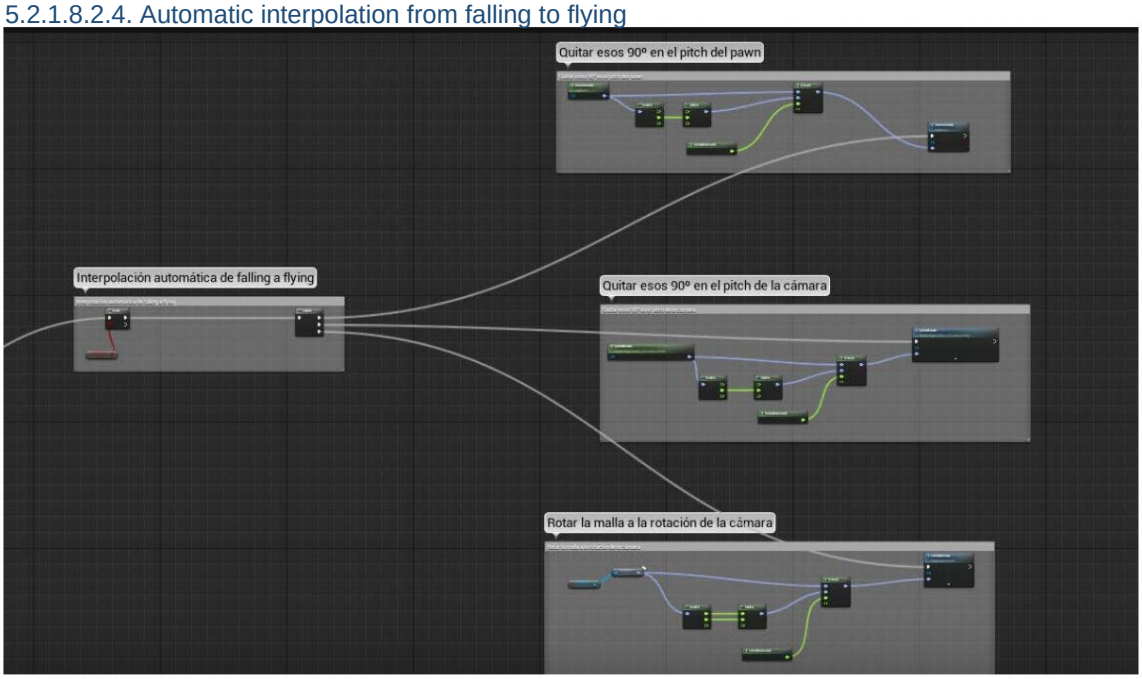


Figure 121. Capture of the general logic of the interpolation of the step from falling to flying

Source: self made.

For interpolation from falling to flying, the steps are exactly the same as from flying to falling, but with the difference that this time we remove the -90º that we had added, leaving them at 0.

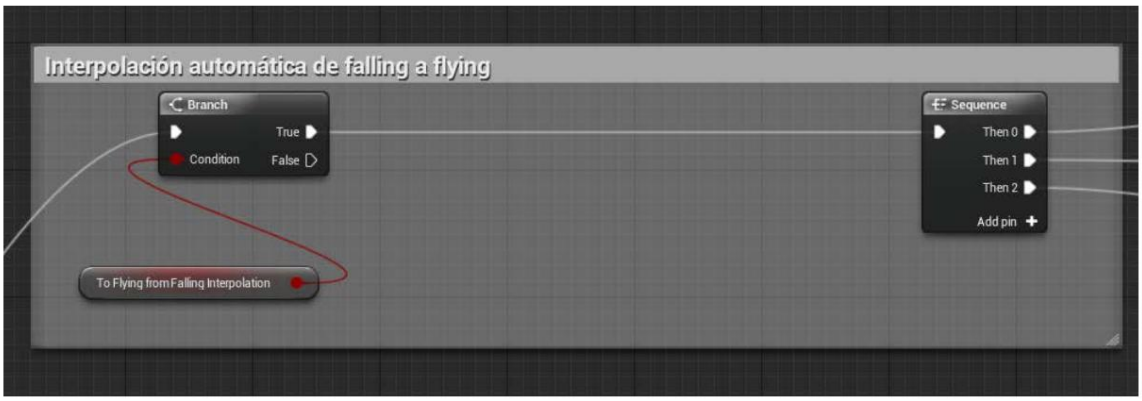


Figure 122. Check capture if it is necessary to interpolate from flying to falling

Source: self made.

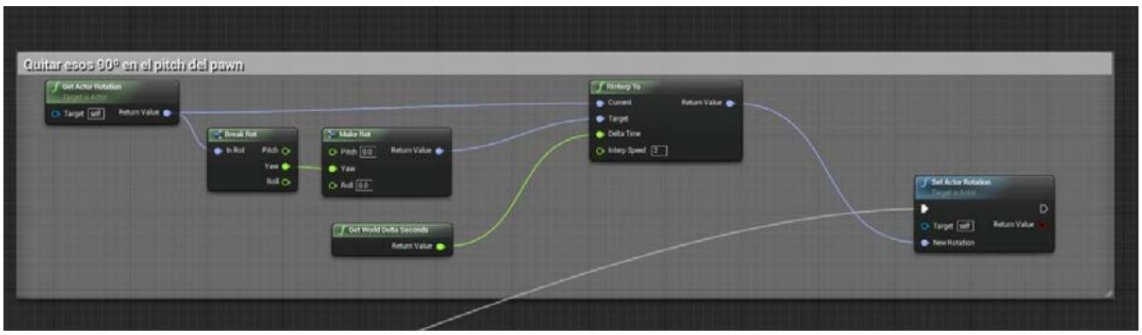


Figure 123. Screenshot of restoring the rotation prior to the falling state of the character.

Source: self made.

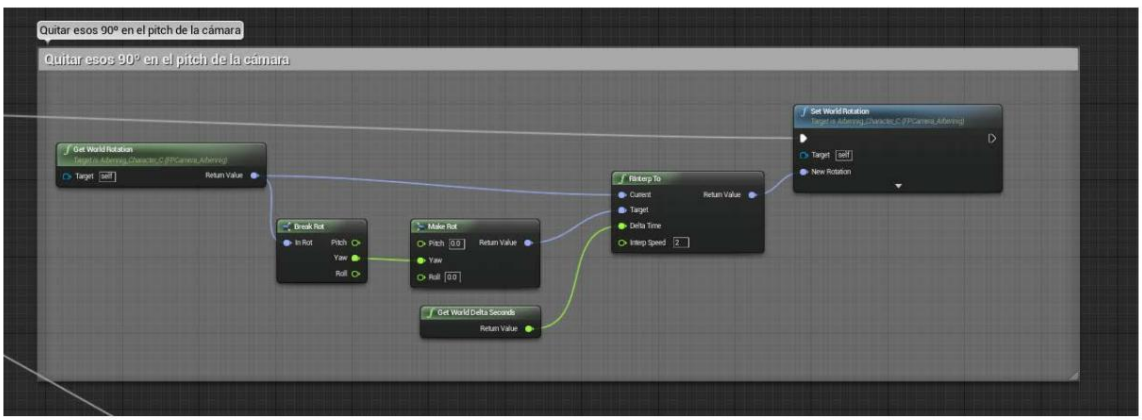


Figure 124. Capture of restoring the rotation prior to the falling state of the camera.

Source: self made.

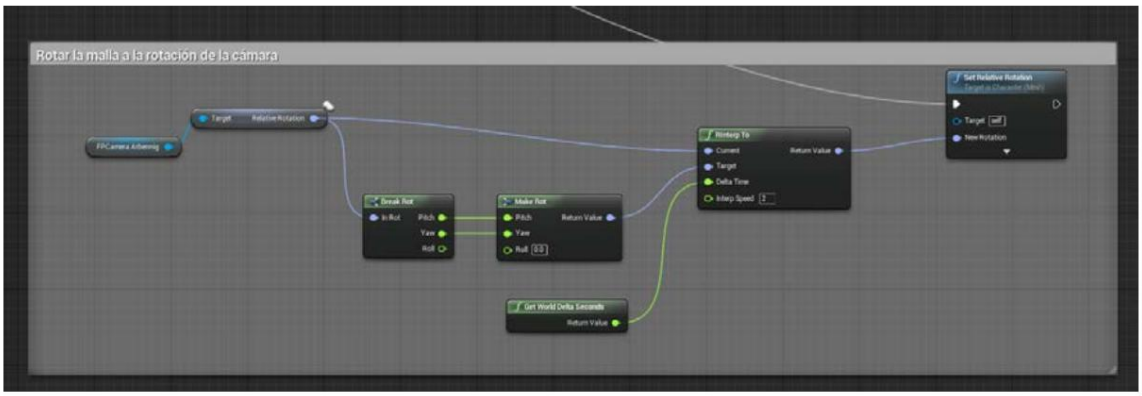


Figure 125. Mesh Rotation Capture to Camera Rotation

Source: self made.

5.2.1.8.2.5. Automatic interpolation from Racing to Flying

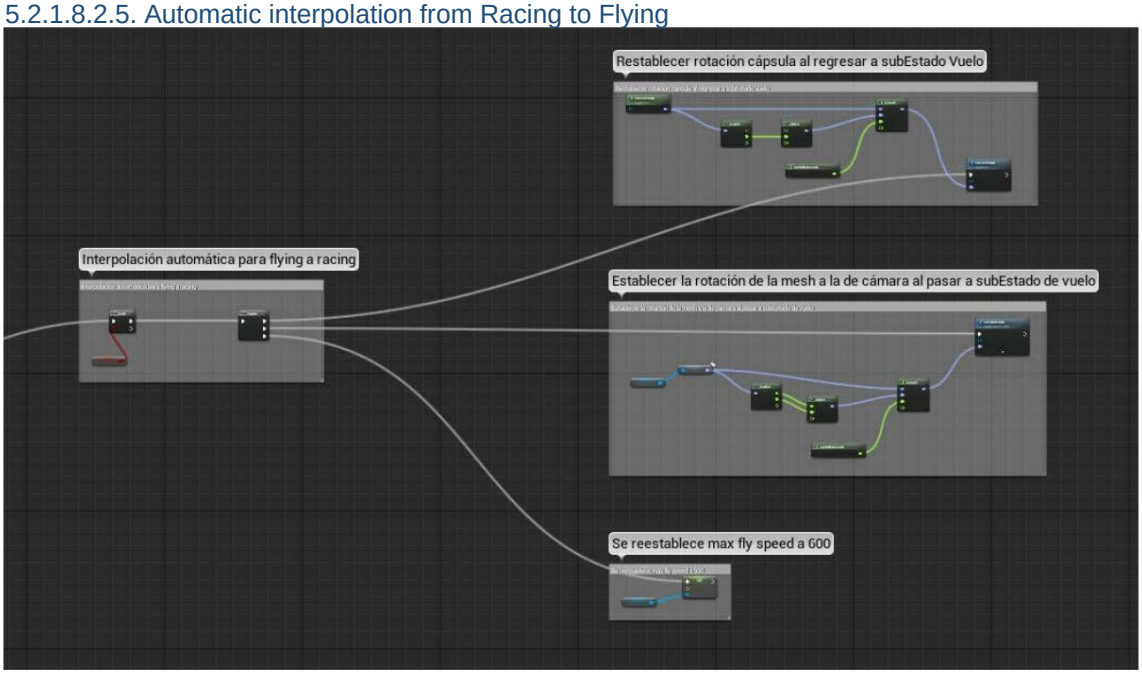


Figure 126. Capture of the general logic of the automatic interpolation from flying to Racing.

Source: self made.

For the interpolation from Flying to Racing, the process we perform is to reset the rotation of the capsule (the character) to 0 in its pitch and its roll, since in Racing these values change. After this, we make the rotation of the mesh be assigned to that of the camera (when we are in Racing, the mesh rotates with the actor, while the camera is free), and finally, we lower the maximum flight speed to 600.

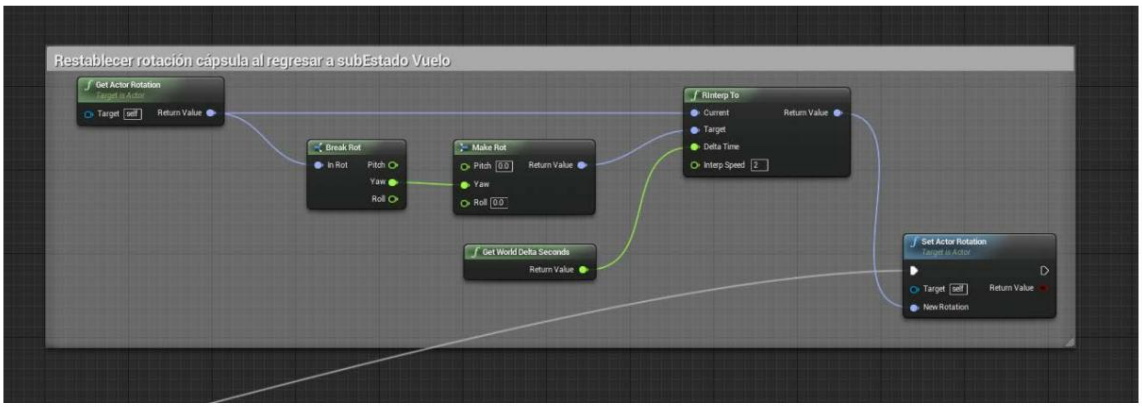


Figure 127. Capture to reset the rotation of the capsule when returning to flight.

Source: self made.

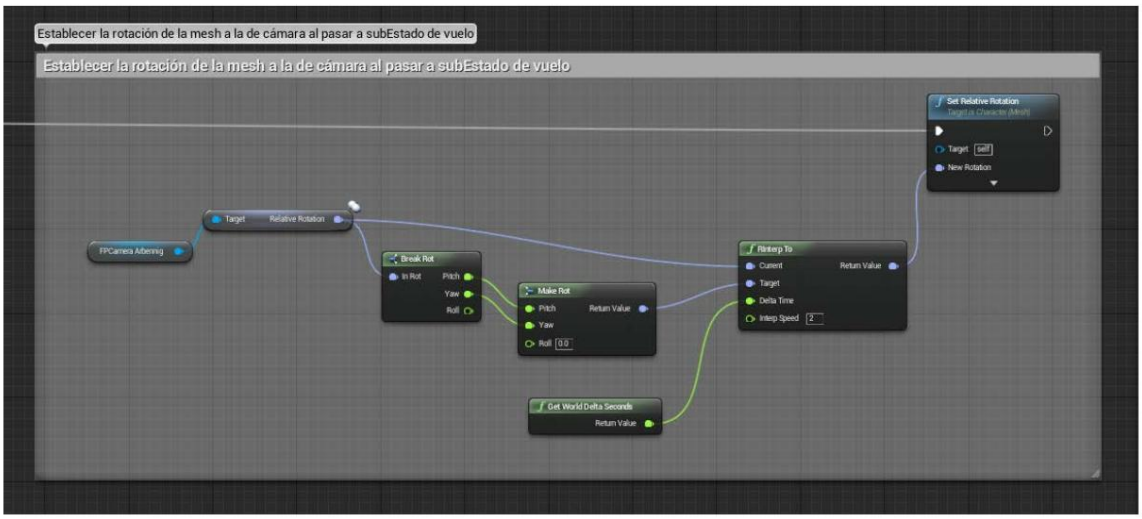


Figure 128. Screenshot of the logic of setting the rotation of the mesh to the camera when hovering (no racing).

Source: self made.

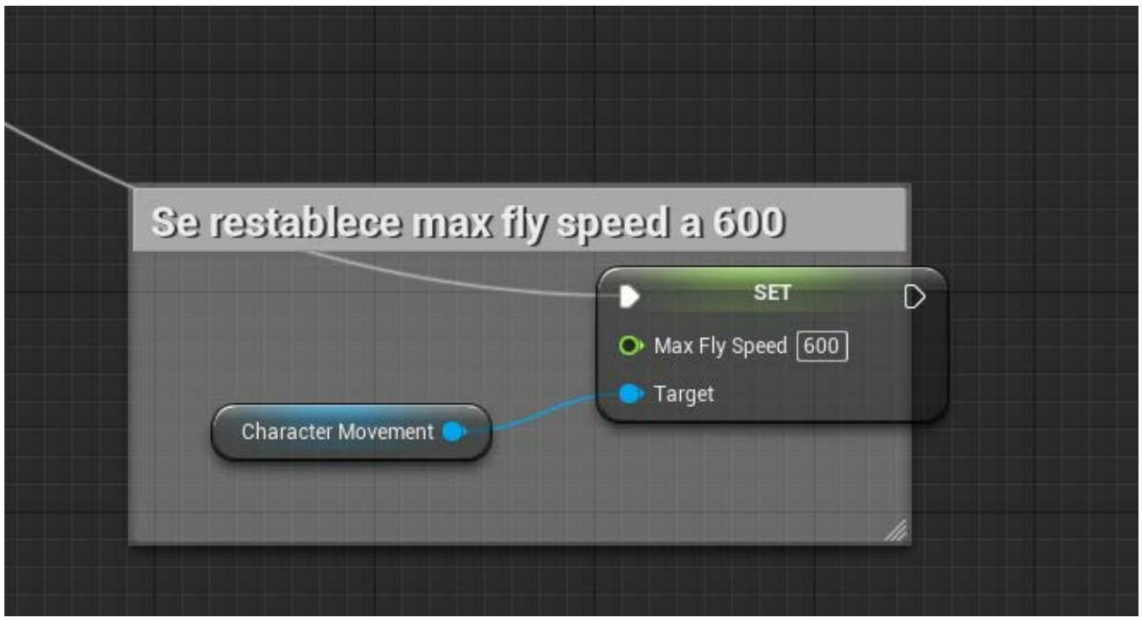


Figure 129. Screenshot of resetting max movement speed to 600 instead of 10000 as we have in Racing.

Source: self made.

5.2.1.8.2.6. Activate tweens

As a last detail to add in terms of interpolations, we activate them to perform the previously named changes in both automatic and manual state changes, and we do this by setting one of these variables to true, and the rest to false.

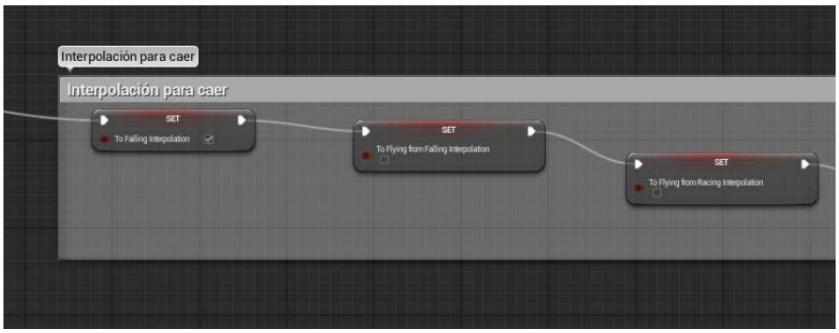


Figure 130. Fall interpolation activation capture (going from walking to falling)

Source: self made.

5.2.1.9. Compatibility controls with Oculus Rift DK1.

In order to finish making game controllers compatible with the Oculus Rift camera (remember that for the Racing state we already made some changes in the previous section, and that we have activated the plugin that allows us to use it), in our particular case, we have decided enter the player controller, and from there, manage that the states of our character in Walking and Flight, so that the mesh moves according to the camera of the peripheral, which will be assigned to the PlayerCameraManager.

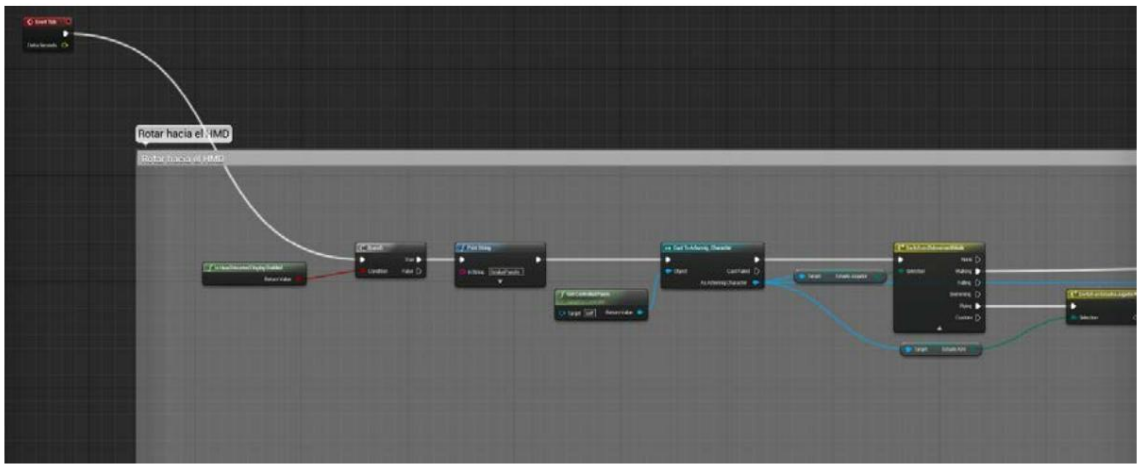


Figure 131. Capture of the necessary checks in the PlayerController so that the camera rotate to where the Oculus Rift device rotates.

Source: self made.

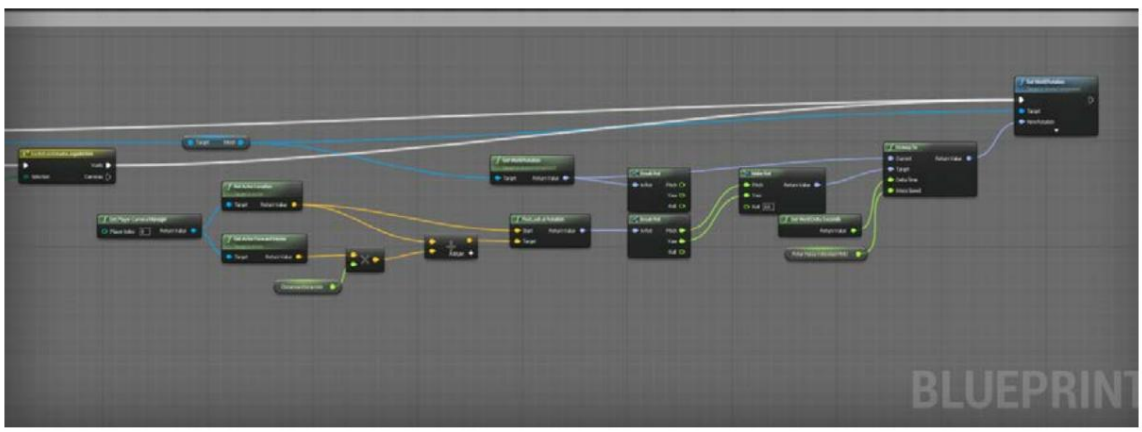


Figure 132. Capture of logic so that the mesh rotates in the direction indicated by Oculus in Walking and Flight.

Source: self made.

5.2.3. Game level development.

Figure 133. Screenshot of Unreal Editor with the Arbennig project open.

Source: self made.

This section aims to explain the realization of all the components of the game that are not they carry implementation logic that interferes with the flow of the game. That is, import and asset positioning, creating terrain and particle effects, and lighting.

5.2.3.1. Asset creation and import pipeline.

In this project we include both assets brought from other templates and materials offered by unreal, as our own assets that we have created in a 3D modeling program previously.

The following explains how both tasks have been carried out.

5.2.3.1.1. Asset import from 3ds max

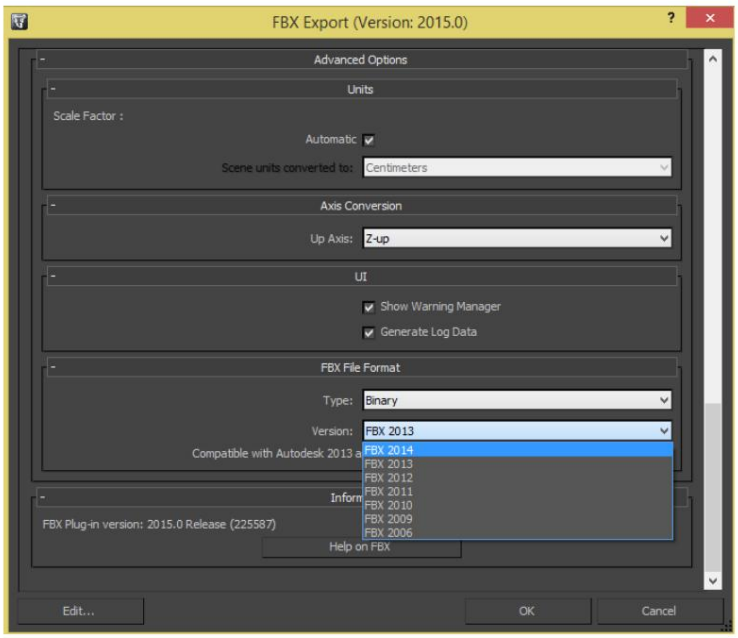


Figure 134. Export screenshot from 3ds max selecting FBX 2014.

Source: self made.

To import meshes to the engine, what we must do is export them to the .fbx format in its version of 2014[26], and after having the exported file, drag it to the content browser of the engine, or if prefer, click the import button and select the file.

The engine will recognize it, and will show us an import message, where we simply have to click on Import, since by default it is how we will initially be interested in having all the imported meshes.

If later, for example, we are not interested in having a mesh in particular collision, we can access to the already imported static mesh and choose remove collision.

When we import an fbx, the materials that it had assigned are automatically assigned in 3ds max, being created automatically in the engine.

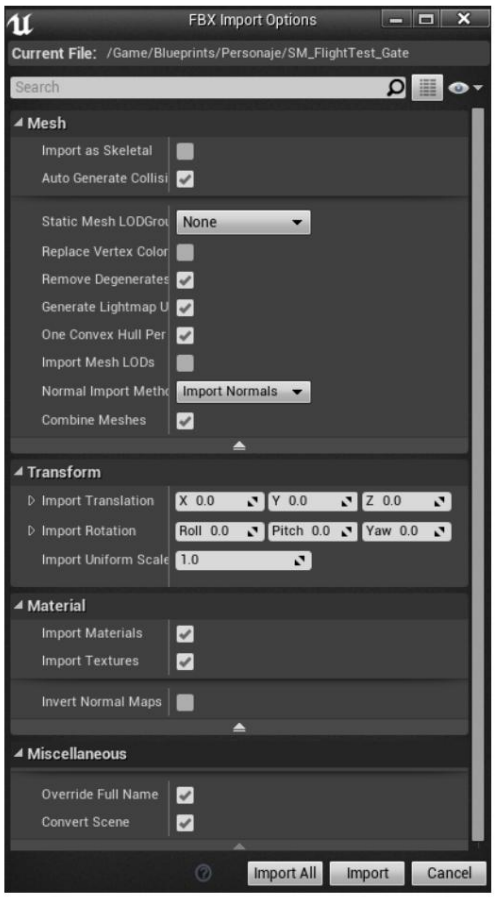


Figure 135. Screenshot of the message that appears after trying to import a .fbx mesh

Source: self made.

After we have imported the mesh, we can freely place them around our stage
dragging or spawning them from code.

5.2.3.1.2. Importing assets from another UE4 project

If we want to import assets from other Unreal Engine 4 projects, for example from templates that
are offered for learning and sample engine features, which are free of charge,
we can do it by clicking on the asset, or set of assets, and choosing between its options
the Migrate option, which will ask us for a directory where to migrate them.

In order to do this, we need to migrate assets from the same version of the engine, since
assets from later versions may not work and cause problems.

5.2.3.2. Creation of land (landscape) of the project.

The creation of a terrain with the engine can be done using the landscape tool included in the editor.

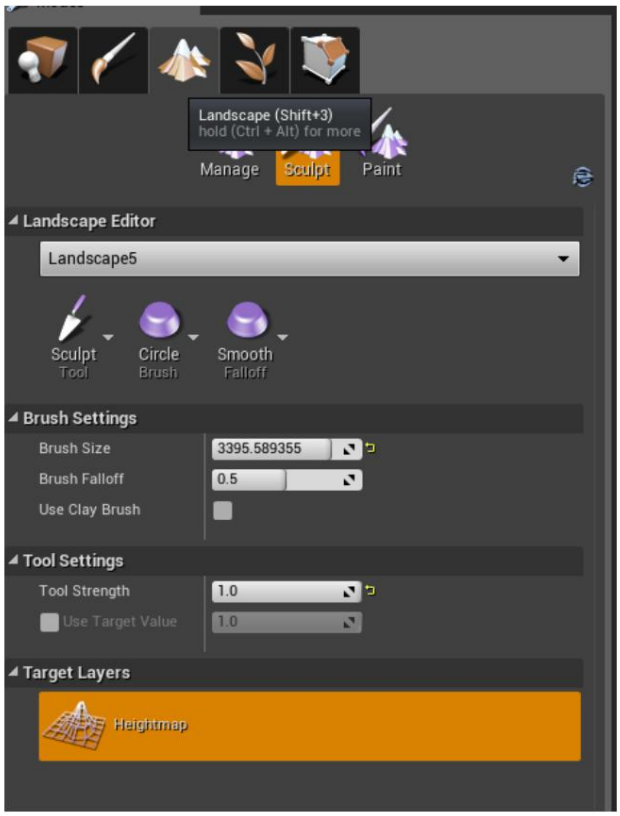


Figure 136. Capture of the engine's landscape tool.

Source: self made.

Its operation is simple, using the manage option, we can create new landscape, where we see different options for creating and selecting the material to be used in it.

Using the Sculpt tool we can "model" this landscape, which is initially a plane, which we can do freely with the brushes it offers, or load a height map and make the desired shape automatically.

Finally, with Paint, we can paint the landscape with materials.

In this project, we have used the landscape tool to create the different islands not floats that make up the world of Delfrydoland. To the labyrinth of the island where the Totem 5, a basic height map has been created and it has been retouched after having created it in the engine.

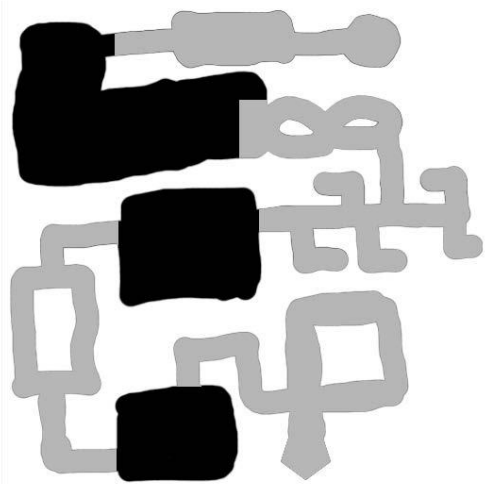


Figure 137. Map of heights of the labyrinth of island 5.

Source: self made.

In this height map, which should be in grayscale, black means maximum depth, while white means that no depth is desired.

Figure 138. Landscape of the maze once it has been created with the height map tool of landscape.

Source: self made.

The rest of the islands have been created freely until they end up with a suitable shape.

5.2.3.3. Create a particle effect of fire sparks.

We want to use a fire spark particle system as a decorative element in the level. These sparkles give a more realistic feel to volcano-shaped decorations than usually appears next to each totem.

5.2.3.3.1. Anatomy of a particle system.

Before starting to create particle systems, it is necessary that we know the different parts of a particle system in Unreal Engine 4.

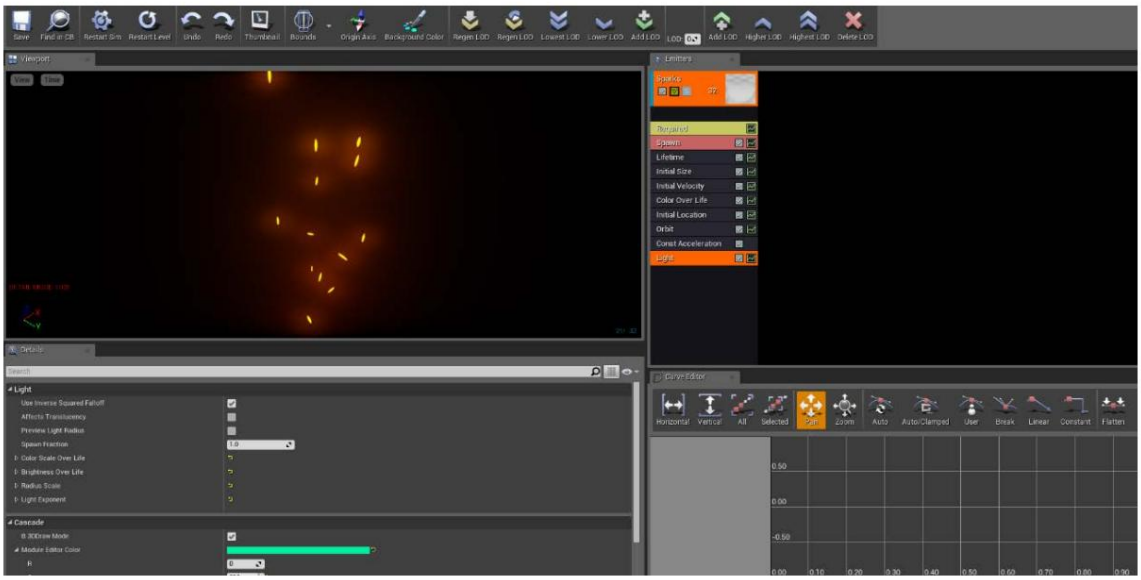


Figure 139. Capture of the particle system that has been created.

Source: self made.

That said, a particle system in this engine is made up of emitters, and a emitter is made up of an emit block, and a list of modules.

The emit block contains different properties that help us create different components of the particle system. These properties are the visibility of the block (to show or not the block), the form of emission to see in the preview window (it can be the normal form, only with points, only with ticks, as it would be seen on stage (lit), and not show anything), and the "Solo" button (when we have multiple emitters, it allows us to only visualize the one that we mark with Solo).

Regarding the list of modules, we have Required (it will allow us to change properties such as the emission material, you can also change the direction and rotation in which it is emitted), Spawn (allows us to configure the amount of particles that are going to be emitted (in the drop-down Rate->Distribution->Constant)), Lifetime, Initial Size, Initial Velocity, y Color Over Life, como modules by default, but it is possible to add or remove modules by right clicking on the module list (all modules except Required can be removed from the emitter). The

modules allow us to "expose" each of them, which allows us to change their properties over time in an edit curve. The workflow regarding each module is first access the properties, and, after this, if it is necessary to touch the editing curve, pass to her then.

A particle system can be composed of several emitters. To create more emitters of particles, it can be done with right click on the emitters window, and it allows us that option (create sprite emitter), to remove it, right click on the emitter, and in the tab of emitters, allows us different options, among which we find Delete emitter. Each emitter is separated into columns, which are called emitter columns.

Now that the operation of the particle systems in the engine has been explained, we proceed to the creation of the particle system that we will use in the game, we are going to create some decorative sparks, which are of the sprite emitter type.

5.2.3.3.2. Implementing the spark system.

Before starting to create the particle system, keep in mind that it is actually It consists of two parts, on the one hand we have the visual (how we see the effect itself), and on the other, its behavior (the logic behind it, composed of what was described above from its anatomy).

The first step in creating a particle system is to create it in the content browser, to This can be done just as you create blueprints (right click -> Particle System), and give it a Name.

The next thing is to set its basic parameters, that is, through the Required module.

Sparks, remember, need an emission base material. We created this base material us, in our particular case, it will be a simple material:

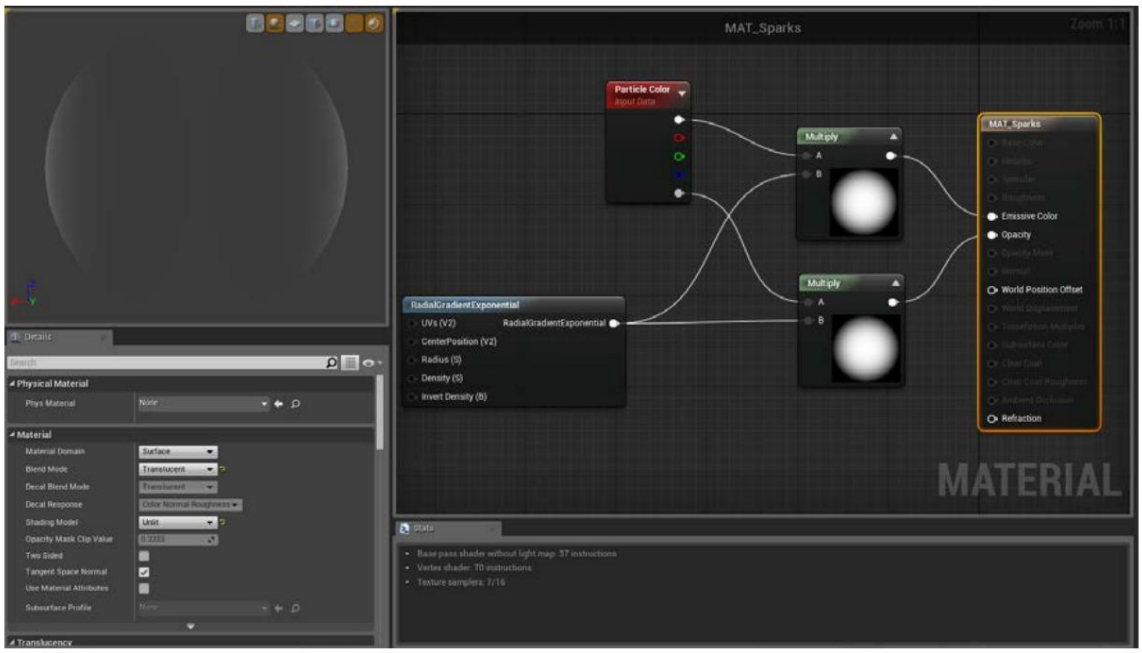


Figure 140. Capture of the material that uses the created particle system.

Source: self made.

What is special about this material compared to a normal material is that it receives data with an InputData event of type ParticleColor. On the other hand, to be able to create the material correctly, we need to set in the properties of the material, that its blend mode is translucent (this is how enables us the opacity property), and the shading mode we place it in unlit, so that our particle system to be as non-aggressive as possible with the CPU.

Once we have the material, going back to our spark particle system, we change it in the Required module.

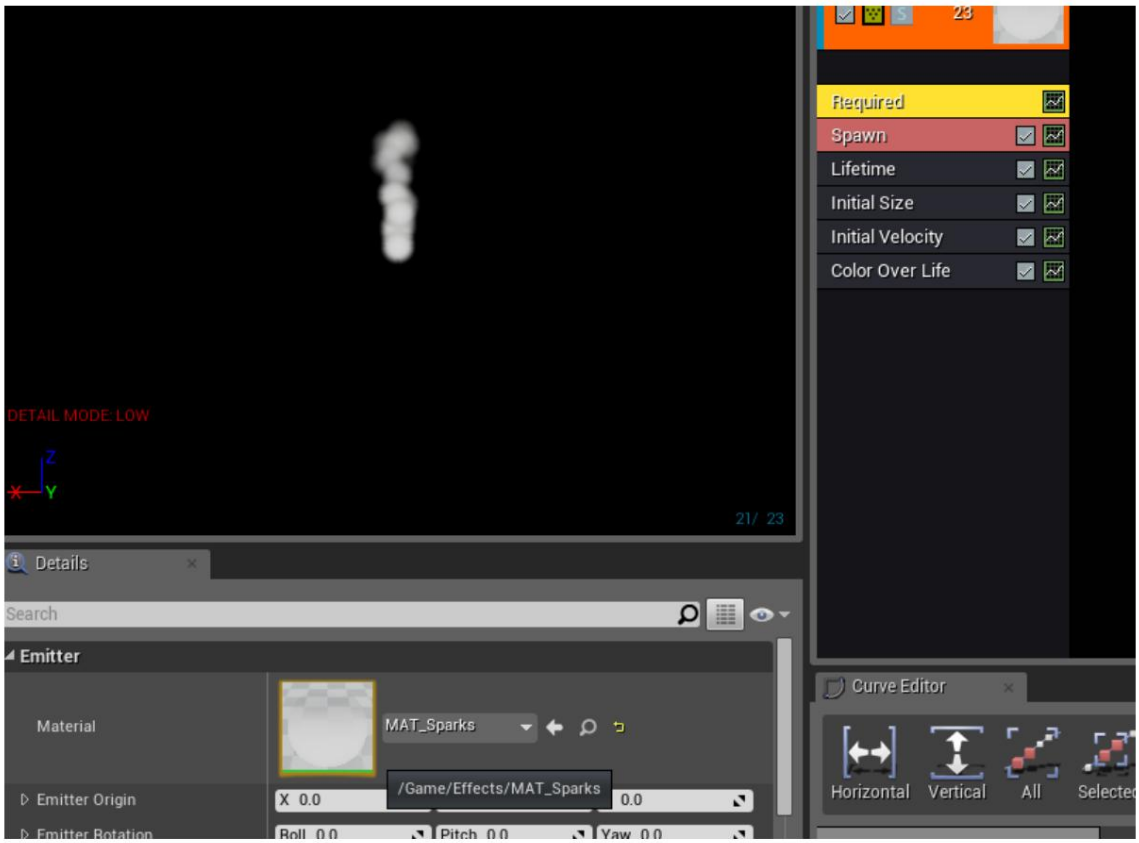


Figure 141. Capture of the particle system after applying the created material.

Source: self made.

In our case, we want the emission to be facing up, so we're not going to modify the emitter's rotation, and we also don't want to change its origin.

We set Screen Alignment to PSA Velocity. Screen Alignment is used to change the capacities of each emitted particle. By default it is on PSA Square, which issues a single square polygon for each emitted particle, with PSA Velocity we want the particles change, and change with speed if necessary.

Figure 142. Screen Alignment option screenshot

Source: self made.

We go to the Spawn module, since we do not need to have the default particles (20), but that less is enough, we change it to 10.

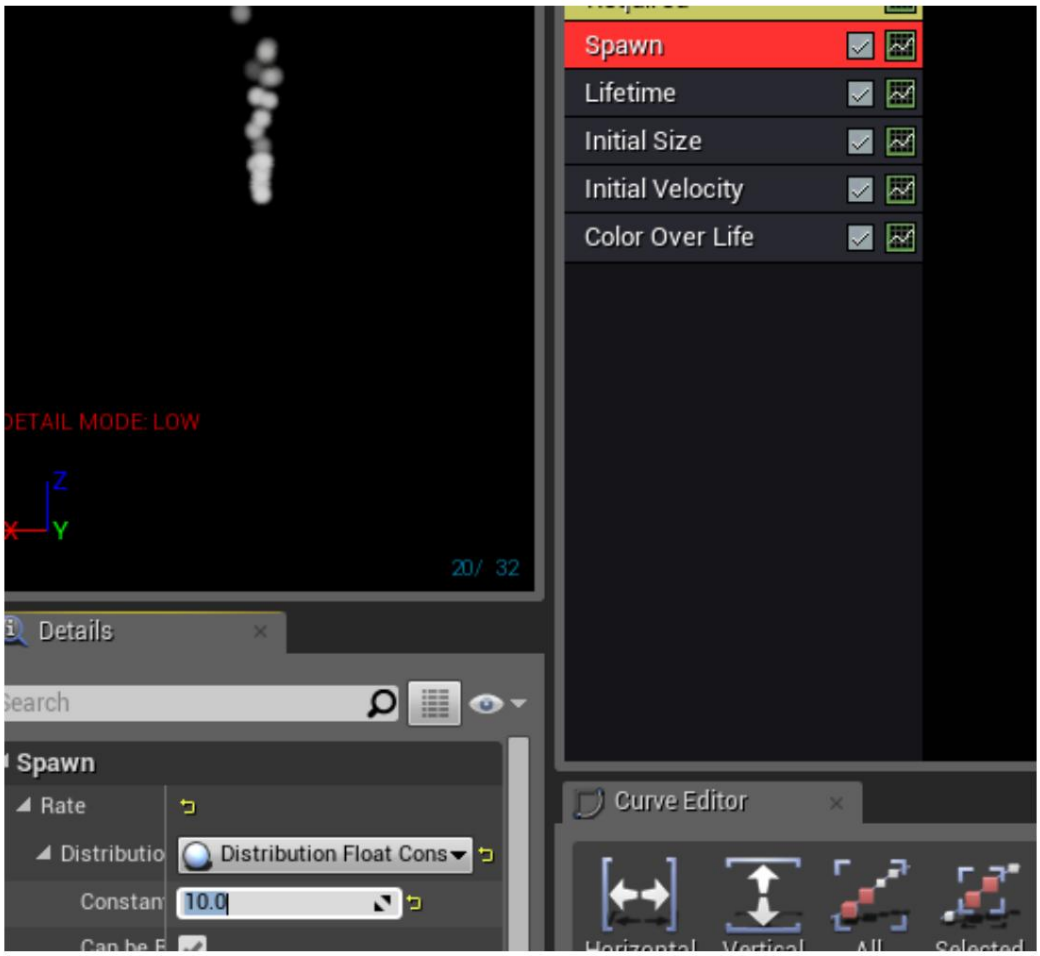


Figure 143. Capture of the particle system after reducing its amount to half by default.

Source: self made.

We continue to the Lifetime module, in this module we have the minimum and maximum duration time, by default both appear at 1. We change the maximum time to 3, so that some last more time and others less, and not be so uniform.

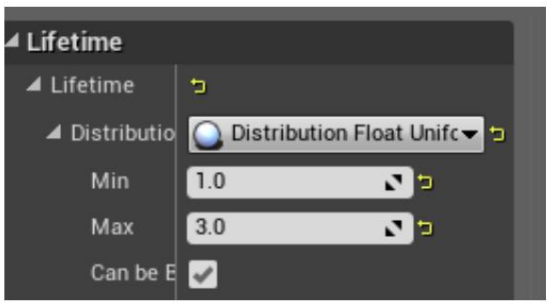


Figure 144. Capture of the minimum and maximum lifetime that we have decided to assign.

Source: self made.

In the Initial Size module, which is used to change the size of the particles, we are going to give them an equal minimum and maximum size, and values (2,10,2), which makes it look more like sparks.

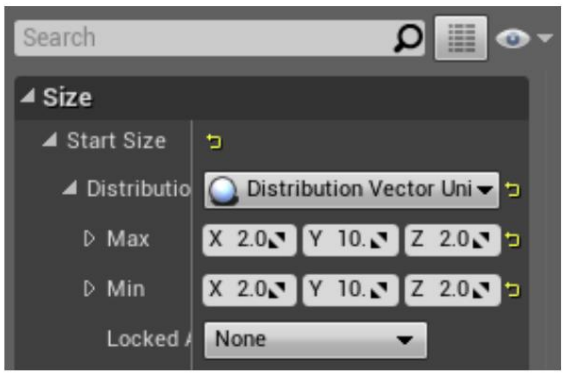


Figure 145. Capture of the initial size of the particles modified to one that best suits the wanted.

Source: self made.

Now we move on to configuring the Color Over Life module, which is used to modify the color of the particles throughout their lifetime. To do this, we have a vector with two points, 0 and 1. The start value of the color is 0, and the end value is 1.

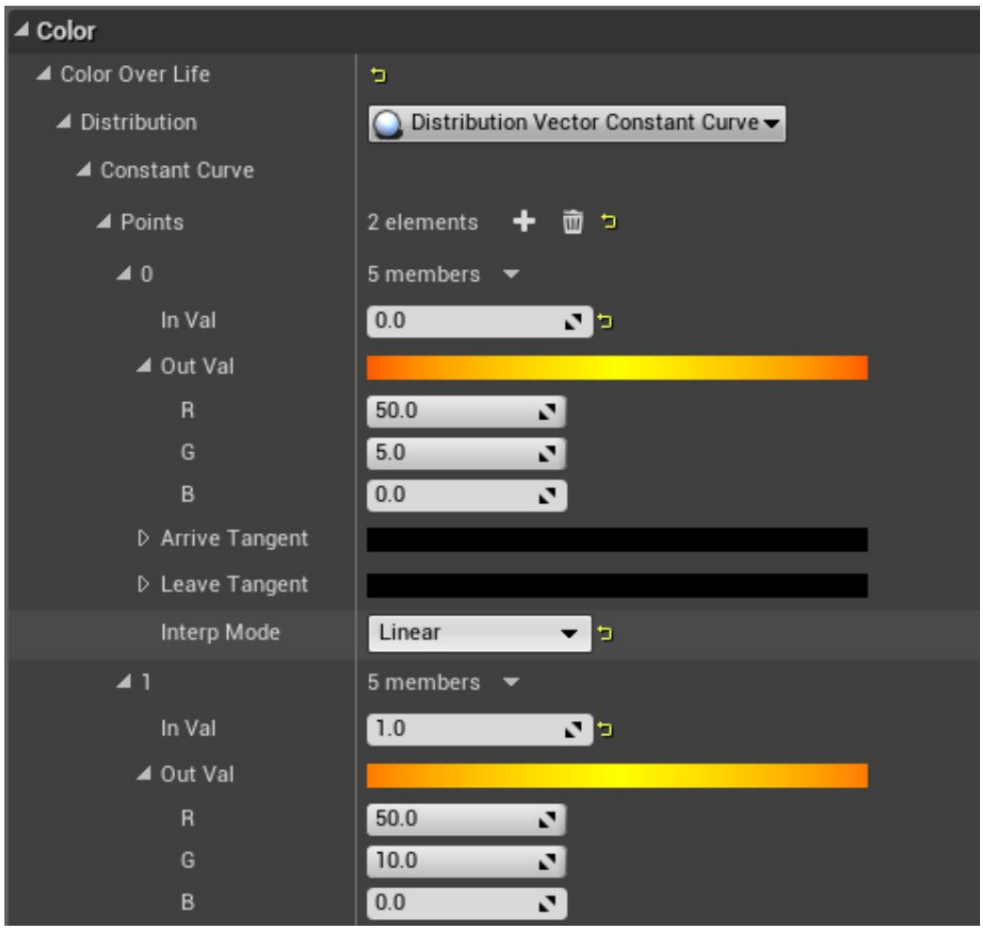


Figure 146. Capture of the chosen color that the particles should have.

Source: self made.

We place as output value (out val) in 0 the following (50,5,0), and in point 1 we place (50,10,0). When we go from 1 in each position of the RGB vector that appears to us, it begins to create a glow effect.

Figure 147. Capture of the result after overloading the value, the Glow effect can be seen.

Source: self made.

The next step is to add one more module to the emit block, the Initial Value module (what we add with right click in the list of modules). With this module we are going to make the origin of the sparks appears “out”, so the particles start to look more distributed.
We do this in our case by changing the maximum distribution to (30,30,0) and the minimum to (-30,-30,0).

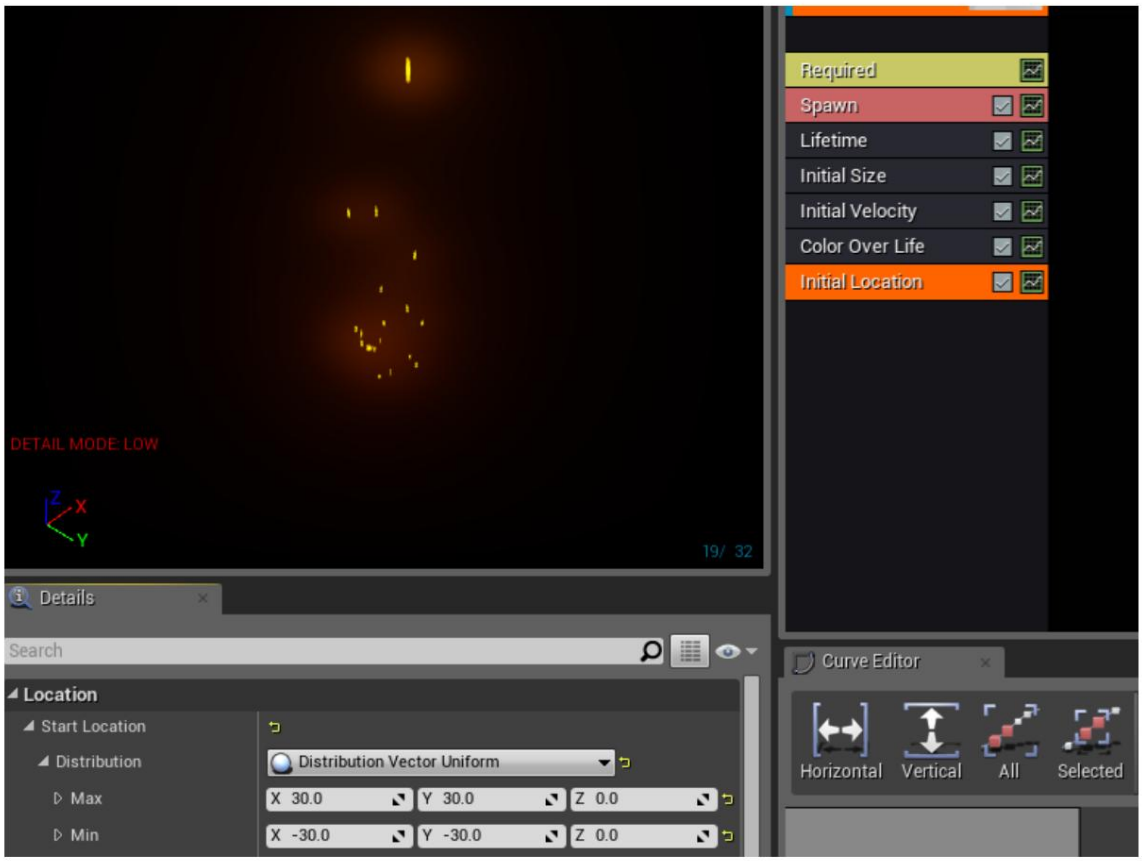


Figure 148. Capture of the particle system after modifying its initial location.

Source: self made.

After this, the goal is to make them look more like sparks, with a slightly more chaotic. For this, we are going to add the orbit module. In addition, we want it to give the feeling that they go up by increasing the speed up, we do this by adding the module const acceleration, and indicating in the acceleration in Z a value of 100, thus, during its time of life, the particles will constantly increase their speed in Z until they disappear.

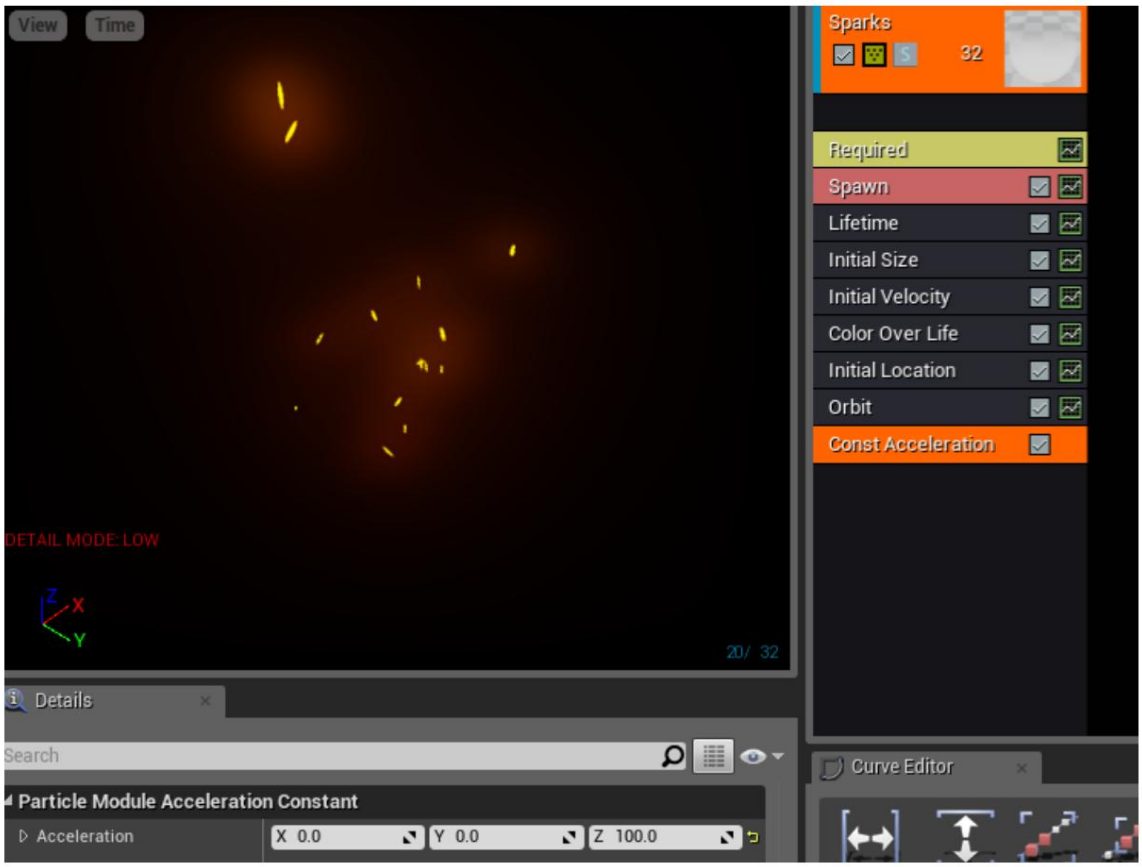


Figure 149. Capture of the particle system after applying the Orbit and acceleration modules constant.

Source: self made.

And finally, we are going to add a Light module, so that each particle has a light source. Nope parameters need to be changed. And now we have our particle effect. Now I just
We drag the stage to see the result.

Figure 150. Capture of the particle system within the game.

Source: self made.

5.2.3.4. Add the lighting.

In the game it has been decided to use two types of lighting offered by the engine, point lights, and Light Source.

While the point lights are point lights that have been placed around the map to give greater illumination to some assets, and to help the player understand that they can interact with some NPC or trigger, the light source has been used as the global light of the level, which keeps all the stage lit with an acceptable light intensity.

To perform light calculations correctly, a component called volume lightmass, which calculates all photons within a given closed environment.

The lights used do not contain special modification, in some the intensity has been modified to produce more light, while in others, the default intensity has been considered the ideal.

Its way of adding is like all the actors in the game, it has been added to the stage by dragging with the mouse, or has been added as a component of some trigger or NPC blueprints.

5.2.4. Game sound.

The sound of the game in Unreal Engine 4 is treated in a considerably simple way.
To be able to play a sound in the game, it is only necessary:

1. Import the audio into the project. The Unreal engine only supports the .WAV format for audio, which, when imported, converts to Sound Wave. For this project, conversions from other formats to .WAV have been done with the Audacity program.
2. Call an audio playback node, which may well be Play Sound at Location, Play Sound, Play (audio component), among others.

The way to indicate that we want an audio to play in a loop until we order it to stop, is defined in the properties of the same imported audio, in a checkbox called loop that we must call if we want it to play constantly.

5.2.5. Implementation of the User Interface

The User Interface section includes both menu implementation, HUD implementation and an Inventory.

For the creation of user interfaces in Unreal Engine, we make use of Unreal Motion Graphics (UMG), whose core is Widgets. Widgets are a series of predefined functions (such as buttons, sliders, progress bars, etc.) that can be used to create
our interfaces. These Widgets are edited in a special Blueprint type called Widget Blueprint, which contains two tabs for creating interfaces:

The Designer tab, which is the visual layer of the interface.

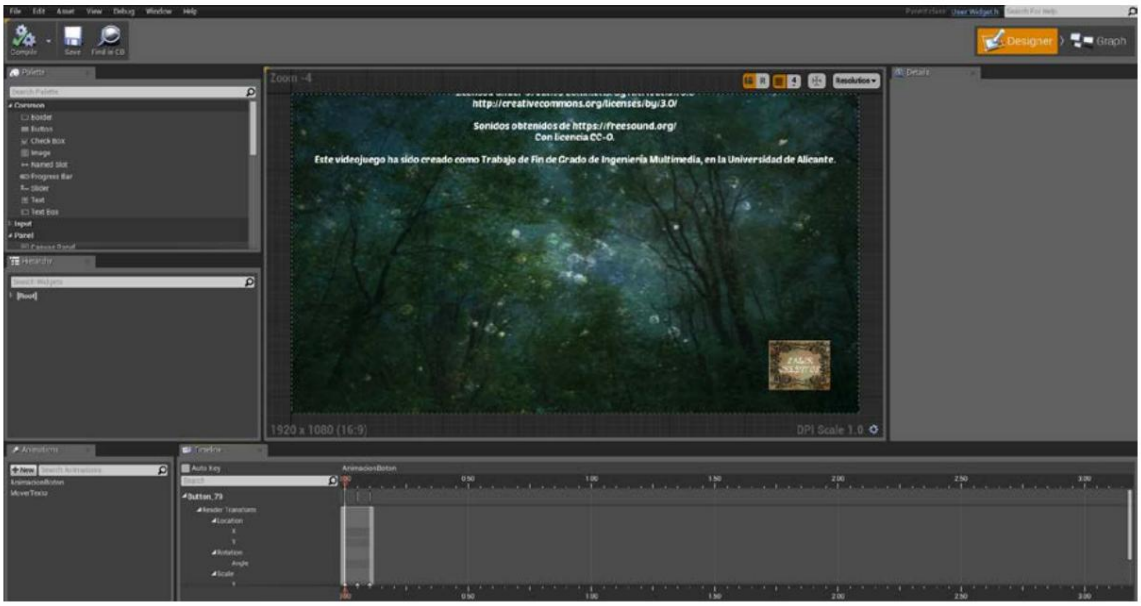


Figure 151. Screenshot of the Designer screen of a blueprint widget.

Source: self made.

The Graph tab (graph), which will provide the functionality to the widgets used.

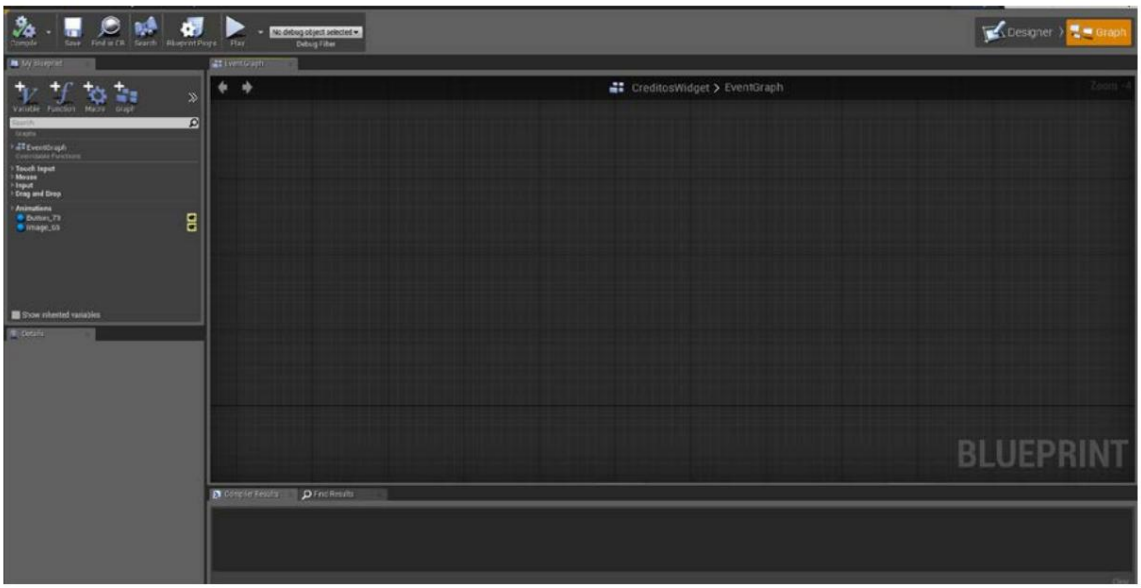


Figure 152. Screenshot of the Graph tab of a Blueprint Widget.

Source: self made.

In other words, with Designer we add elements, create animations and customizations of the generic elements on a canvas, and through the Graph they are given a logic that, for example, allows a level to be loaded when a new menu button is pressed. departure.

In addition to the above, just as the positioning of the elements in a Widget Blueprint works in other video game engines such as Unity, we have some Anchors, to be able to place our menu elements as we wish depending on the type of screen.
Given that in our particular case, we are not going to find screens that are not in 16:9 or 4:3 format, we have chosen to make relative anchors on the screen. That is, whether it is an 800x600 screen, or a 1920x1080 resolution screen, the elements on the screen will always occupy the same position within the screen frame, scaling.

5.2.5.1. Pipeline of creating user interfaces with Widget Blueprints

Regardless of whether it's a Menu, a HUD, or even an Inventory, the basic pipeline for using each UI is the same. Which basically consists of the following.

1. Creating a Blueprint Widget.

2. In an existing blueprint, create from your Event Graph a Widget of the Widget class created in the previous step.

3. Create a variable to store the created Widget. The fastest way to do this, although not the only one, is from the created Widget, which has the created object as an output parameter, dragging through it until a connecting line appears, and with Promote to variable, we create a variable for it. We save the variable for

greater comfort, since if it is not saved, every time you want to access this object, you must draw a line from the node of the created Widget.

4. When you want to add the Widget to the game screen, the variable that we created in step 3 is placed as Get, and through it, the Add to Viewport node is called, which will show the Interface Widget.

5. On the other hand, when you want to remove the Widget from the game screen, as in the previous step, the variable that we created in step 3 is placed as Get, and through it, the Remove node is called From Parent.

5.2.5.2. In-game use

As reflected in the Game Design Document, for Arbennig we built several game menus (main menu, options, credits, confirmation, pause, help), and a HUD interface, which includes both a status bar of character (life, energy, and keys obtained), as an inventory.

It should be noted that the maps where we find menus are the main menu map, where the player's interaction is fully with the UI, and the game map, where the player has a HUD and the pause menu, in addition to other 3D menus. displayed when interacting with elements of the stage.

In addition, each button has been provided with a scaling animation, so that when they are pressed, the button is scaled 0.8 on both the abscissa and ordinate axes, and after this, it returns to normal (it is scaled back to 1). The animation is created in the designer, and assigned to each button through the graph, in which when the button is pressed, we create a Play Animation node, to which we give the created animation as an input parameter. This node is followed by a Delay node that delays any other functionality after the button, so that the animation can be seen.

The buttons also have a sound for when they are pressed, and when the cursor is placed on them, although this logical aspect of the game is only a matter for the designer, since from the graph it is not necessary to add any functionality.

Next, we proceed to detail points that have been considered relevant when creating the different user interfaces.

5.2.5.3. Main menu

The main menu allows access to the game level, as well as moving between menus and exiting the game.

The functionality to highlight corresponds to access to the game level, and how to exit the game.

To access a game level, it is necessary to call the Open Level node when pressing the new game or continue game button, to which we indicate the name that has been given to the level in the editor.

On the other hand, to exit the game, it is achieved by executing a console command, for which the Execute Console Command node must be called, and in the command to call, write quit.

5.2.5.4. Option menu

As for the options menu, through it you can configure both the resolution at which to view the game (by default in 1280x720), and its general volume.

5.2.5.4.1. Modify screen resolution.

For the resolution, 6 different screen configurations have been established, 3 for the 4:3 format, and another 3 for the 16:9 format.

Modifying the screen resolution, just like quitting the game, is done using a console command. In this case, r.setRes is used followed by the desired resolution in WidthxHeight (the x is included), for example, 800x600, and optionally a w is added at the end to indicate if we want the screen in windowed mode. For reasons of final aesthetics, it has been decided to run the game only in full screen.

It should be added that for version 4.6.1 of Unreal Engine 4, used for this project for direct compatibility with the Oculus Rift DK1 (it will be lost in future versions), there is a bug when making the change of resolution, that allocates all screen resolutions at once as they are chosen. After several personal checks, a trick has been found to fix it, which is to first set the resolution in windowed mode, and followed by this, the resolution in full screen.

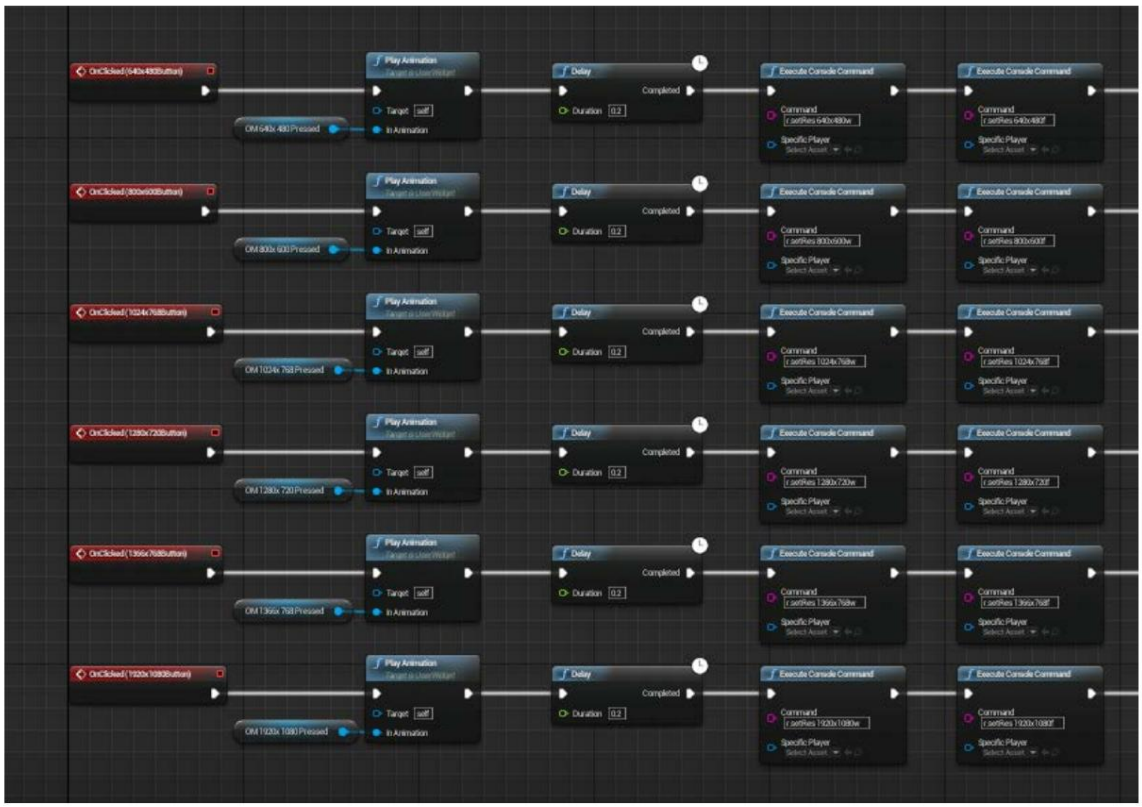


Figure 153. Screenshot of the logic of changing the screen resolution.

Source: self made.

For usability issues, once a resolution is selected, the button to choose it is disabled. The deactivation of buttons is done through the Set Is Enabled node, to which the button to deactivate is passed as an input parameter.

5.2.5.4.2. Modify the overall game volume.

Modifying the general volume of the game is easily done through an audio hierarchy. So a generic audio blueprints class has been created, called BP_Master_Sound, along with three other classes, which are UI, SFX, and Music. In addition to this, the Victory plugin is used, which incorporates extra functions to the engine, and that we download from the Unreal website for free.

Through the BP_Master_Sound class, three child variables are created, which are the three named in the previous paragraph, and, in the graph of this parent class, nodes are linked to indicate that they are child classes.

When we import audio, what we do is indicate the sound class to which it belongs, setting BP_Master_Sound by default.

Thus, if we modify the volume of the BP_Master_Sound class, the volume of the child classes is modified. In addition, this allows that in future changes it is possible to add more personalized audio modification options, such as allowing the volume of each type to be modified independently, and also, this hierarchical structure allows the creation of new child subclasses. For this initial project, despite being hierarchical, it is

regardless of the type of class to which it belongs, since the volume change applies to all, but in possible extensions, this detail facilitates any addition of audio functionality.

Finally, to end the options, at a logical level, we use a slider with a value from 0 to 1. In its OnChange event, we call to update the general value of the parent audio class with “Victory Sound Volume Change”, and on the other hand, regarding the default value of the slider, we create a bind function for this property, which will allow us to have the value initialized to the general one, and in turn, have it controlled at a logical level.

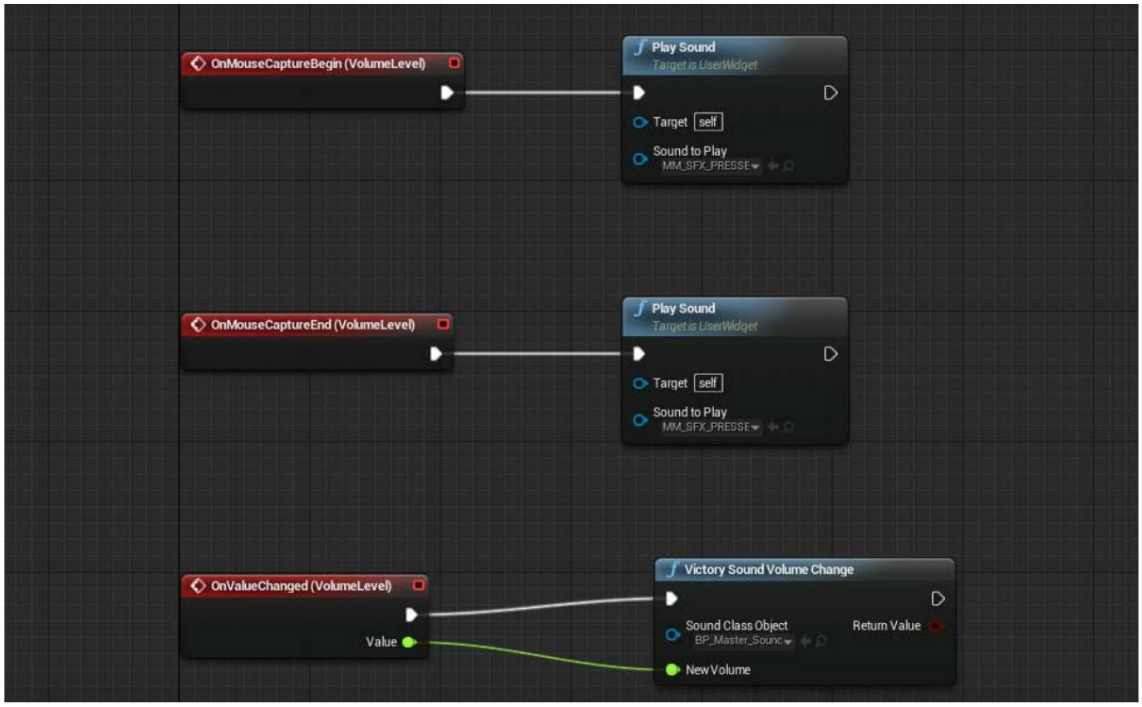


Figure 154. Capture of the logic of modifying the general volume of the game, and an example of audio reproduction.

Source: self made.

5.2.5.5. Pause Menu

The functional detail to take into account in the pause menu is, strictly speaking, the possibility of pausing the game.

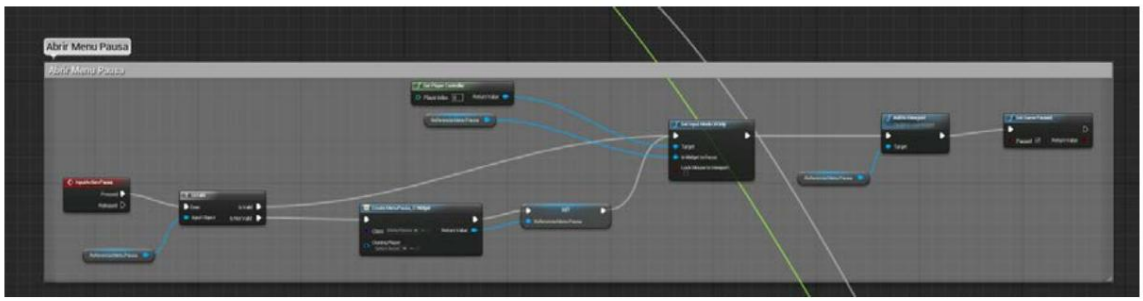


Figure 155. Capture of the pause logic.

To pause the game in Unreal Engine 4, there is a special node for it, this node is Set Game Paused, which has a logical variable that we set to true or false, depending on whether

we want to pause or resume the game. The Unreal engine automatically handles pausing the game.

5.2.5.6. HUD

The functionality to highlight with the HUD is that it communicates at all times with our character's blueprint, so that if he loses life, for example, the HUD must reflect that loss in the health bar. Or if the player picks up an item, this item should appear in the inventory (when the player opens it).

To achieve this direct communication between both blueprints, we have chosen to create the HUD from the blueprint of our character Arbennig, so that a reference to the HUD is saved at all times. Also, from the HUD, we get a reference to the character using the Get Player Character node (with input parameter player index set to 0, since we only have one), and after this, we cast it to the specific class of our character (Arbennig_Character), with node Cast to Arbennig_Character. Finally, we assign this cast to a variable that we can comfortably use at any time, quite quickly with promote to variable from the output of the cast result object.

Once the communication channel between blueprints is established, those values that you want to modify are "binded", so that our variables are always pending changes, and thus our HUD can reflect any change that occurs to the player, such as losing life or energy , or pick up a key. These bind functions are assigned from the Blueprint Widget Designer, and given a logical behavior from the graph.

For keys, for example, 8 variables of type ESlate Visibility (the type of data structure that UE4 uses to perform visibility changes on elements) are created in the HUD, and 8 variables of type bool in the character Arbennig. What we do in the bind function is check with a branch if any key happens to be true, and if so, its visibility will go from hidden to visible. In this way, if the player obtains a key, the HUD will listen for that change automatically, and will proceed by making the obtained key visible.

With energy and health, that we have progress bars, instead of binding on visibility, we bind on the value of the fill percentage of the bar. Having a normalized value from 0 to 1 in the player, that value is simply read and the percentage is displayed on the bar.

5.2.5.6.1. Inventory

The extra function of the inventory is, in this case, the UI that contains the most extra functionality.

Its logical operation allows us that, being in the game, if we open the inventory, it is shown on the screen, and if we close it, it disappears.

In addition, the inventory is made up of two parts, a sampler part, where 10 initially empty boxes can be seen, in which objects will appear as they are collected, and, on the other hand, an action part, which is will make it visible if we select a collected object that appears in the inventory. This action submenu allows you to perform three actions, use/deliver the item, drop the item (drops it to the stage), and cancel action (if you don't want to do anything

with the object). Both use/deliver (if it is possible to perform this action with said object) and drop, imply that the object disappears from the inventory.

The inventory consists of an array of elements, so that, unlike the case of keys, which already have their position and only changes if they are visible or not, in this case, when an object is picked up, it is added to the array, when an object is used or thrown away, it is removed from the array, and the objects in the array are optimally placed, thus leaving no gaps in between, for example, once an object is dropped.

In addition, the inventory must not only interact with the player, it is also necessary that the elements that can be picked up, call the inventory so that it is updated if they are picked up.

As an extra, when we approach an object that can be picked up, a text will appear on it, indicating its name.

That said, for the creation of the inventory, it has been used, apart from a hole in the Widget HUD Blueprint to display inventory, two more Widget Blueprints:

1. A Widget Blueprint for each slot of the inventory.

   In this Blueprint, which works like a button with an image, what we do is mainly two things, make a bind of the image that will have the slot of the Blueprint Widget, so that when the player picks up an object, this image passes to be that of the object, and, on the other hand, that when the player clicks on this slot, it calls an Event Dispatcher that sends the slot on which the click was made as a parameter and will activate an event on the HUD indicating that what slot has been selected to treat.

   The buttons are initially set to disabled, until an image other than null is assigned (with the Is Valid node to which we pass the image variable of our Widget), which means that until we collect an object , and is placed in the slot, we will not be able to interact with it. Since it works dynamically and must listen for changes, we bind the behavior of the button to enable it (Is Enabled?), which will be set to true when it has a null image.

   no

   At design levels, in the Designer, we eliminate the canvas that appears in it, since we don't need it, and we add a button, and, within it, an image. At logical levels, to bind the image that the inventory slot will have, since we cannot bind the image property, but we can bind the brush that paints it, by code in our bind function, we create a variable of type Texture2D (the type of variable in which we store the representative images for inventory items), and this variable we convert to Brush by calling the Make Brush from Texture node.

2. A Widget Blueprint to display the text that appears on the objects to pick up.

   This blueprint, like the previous one, does not require a canvas, since we are only going to need a text. Although, in order to position it correctly, we create a horizontal box as a container, and inside it, we add the text. This text, which we want to appear centered, when we create the widget, in its Event Construct, we use the node

Set Alignment in Viewport, and we place it in the center, that is, at 0.5 in x and y.

Here, the logic that we have in the graph allows us to show or not the Widget's text, in addition to positioning it in the Viewport (game screen), which we will do by obtaining the position of the actor on which we want to show the text, we add to that position, 100 units in Y so that it does not appear glued to the object, but a little more displaced, and we convert that position in coordinates on the screen based on the player controller (we obtain the player controller with Get Player Controller with index 0), which for To do so, with the player controller node, we get the Convert World Location to Screen Location node, to which we pass as input the position on which to display the text and it returns the position on the screen. To determine the visibility of the text, the node to convert to screen coordinates, in addition to the position on the screen, returns a boolean that indicates whether or not the conversion could be carried out, that is, if, for example, the player is looking towards up, or with its back, 2D coordinates will not be assigned, and depending on whether this happens or not, we create a boolean select, which, based on its value, allows us to determine the visibility of the Widget (ESlate Visibility) .

And three Blueprints of a different type than the Widget Blueprint:

1. A Blueprint Interface, which allows us to communicate the HUD with the elements that we can collect. Two functions are created in this special blueprint, the use function, and the drop function. For the drop function, in its graph we add an actor variable, which corresponds to the actor that we are going to drop.

2. A Blueprint of type Structure, which is basically a collection of different variables, and we are going to use it as the structure of each slot in the inventory. In this blueprint we add 4 variables, the first, of type actor, which is the item that we will collect, the second, of type Texture2D, which is the image that will represent the item in the inventory, and the other two variables, of type Text, one for the text that will appear floating on the item
when we can pick it up, and another for the text that will appear in the use field, which can either be to deliver the item, if it is from a quest mission, or to be used, if it is an item for healing. We call this blueprint the Inventory Structure.

3. One or more Actor-type Blueprints, which will be the different elements to collect from the stage. This actor must include a mesh and a collision trigger to detect the player, and as variables, a boolean to know if the player is inside the trigger (therefore, the element information message will be displayed), a variable of type Inventory Structure (the Structure type blueprint created previously), and all the components of the structure are filled in the properties tab of this variable, except for the actor, which we indicate by code, and, finally, this blueprint, we add variables (which we must place as public or editable so that they can be modified from the editor) for what the item will do when used, in case we need something, for example, if the character's health increases, we add a float variable for it.

Having explained the above, within the HUD, we add two Vertical Boxes in the Designer, one that will contain the Inventory itself, and another that will contain the action submenu that will be displayed when selecting an item from the inventory. In the Vertical Box of the inventory, we add a Text Block (to indicate to the user that what he sees is the inventory) and a Uniform Grid Panel, and what

What we do is fill it with 10 Inventory Slot (it appears in the User Created tab, since we created it ourselves). While in the Vertical Box of the action submenu, we add a Text Block to indicate to the player that he can perform one of the actions shown in it, and 3 buttons, the use button, the release button, and the cancel button (do nothing).

Once all of the above has been created, what we do is:

1. In the HUD class, where we inherit the interface that we have created, we create a RefreshInventory event, where we create an array with 10 inventory slot type variables. We go through the array with a foreach loop, checking if the inventory is full or not (if it is full we do nothing else), which, if not, we assign Pickup images and after this, we assign with the node assign to each inventory slot a custom event for an item in the inventory to have been clicked, and that what this will do is disable the inventory so that the action menu appears.

   Also, in this class we give functionality to the action menu buttons, making: a. When the cancel action button is pressed, the inventory is reactivated and it is hidden

   the action menu.

   b. When the drop item button is pressed, let's send a message through the interface to the blueprint that we want to drop, and update the inventory by removing the item after sending that message from the inventory, and also call a custom action completed event so that reactivate inventory.
   c. When the use item button is clicked, a message is sent through the blueprint interface to the specific blueprint we want to use, and after that, the inventory is updated and the action completed event is called.
2. In the class that we use to create a collectible element, what we do is inherit, as in the HUD, we must inherit the action blueprint interface that has been created, create the widget that is displayed on the object with which we can interact, and assign the text to display, after this, assign a PickupItem custom Event, which will check if the player is within range to pick up the item, and if it is, pick it up, add it to the inventory, and update it.

   Also, in this class the use action is implemented, which depending on the object, may have one functionality or another.

3. Finally, in the Character class, which will also inherit the Action interface, what we do is control if the player presses the inventory key to show it, and enable mouse controls in the game as well.

   On the other hand, if the player interacts with any items that can be picked up, the pickupItem event found in the classes of the picked up collectibles is called.

   And to finish, with our character we must implement the drop action event, from which we receive a variable with the object that we are going to drop, and, after this, we reactivate its collisions, we make it visible again, and we change its place to a drop location.

5.2.5.6.2. Ring testing timer

For the implementation of this timer and its corresponding logic according to the test,
what has been done is to have in its tick function on the one hand, the update of the time counter,
where it has been decided to calculate the time in a way that adds the value of thousandths of each instant,
and when it reaches the value of 1, one second is subtracted from the remaining time.

On the other hand, in the tick function, we also check if the remaining time is up, or if
the remaining and obtained rings (the obtained ones are updated from the LevelBlueprint),
match, to determine if the challenge is over or not. That we will call a custom event
in the graph that will be in charge of marking that the challenge has ended (and if it is successful or not), so
that from the level blueprint we have proof of it. In addition, the player is teleported to the challenge island,
and this blueprint widget of the hoops and time is made to disappear.

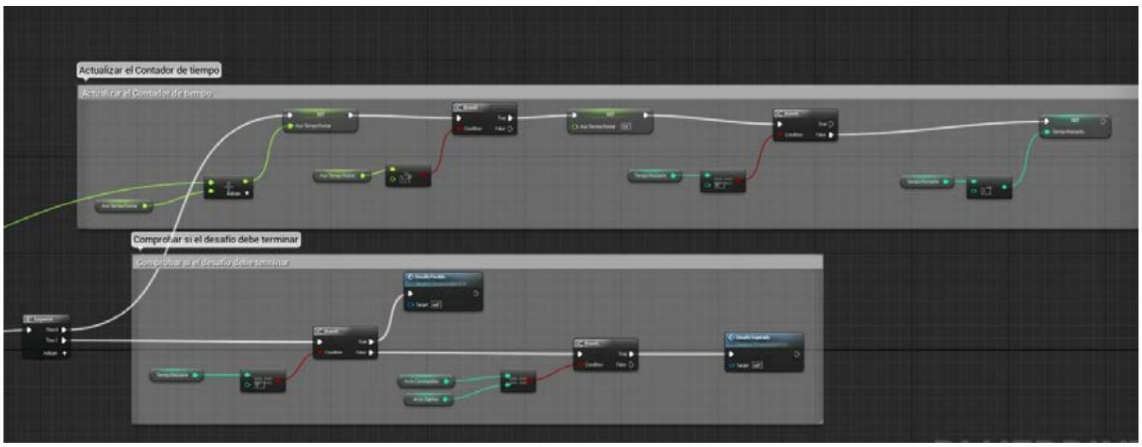


Figure 156. Capture of the game logic of the timer.

Source: self made.

5.2.5.6.3. Prologue, Epilogue, and Credits screen.

The implementation of the prologue, epilogue, and credits screens contain the same logic of
game, without more than punctual variations to decide where these screens point once
its functionality has ended.

These screens have in the designer a background image, animated text that appears below the screen,
scrolling all the way up, and a button to skip the “cutscene”.
Each screen is in a different game level, and loads other levels when finished.
move the text up or the button is pressed.

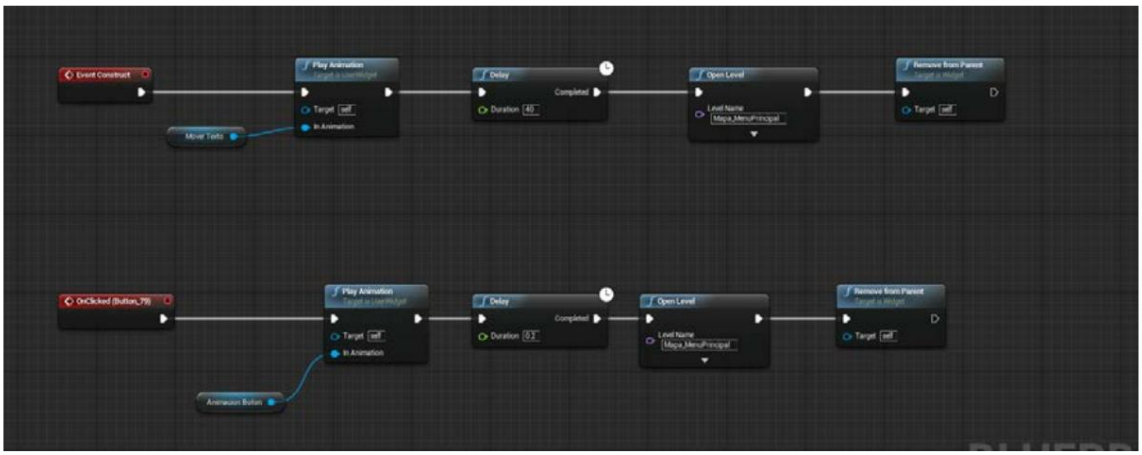


Figure 157. Capture of the logic of the credits class (same for prologue and epilogue).

Source: self made.

The logic that follows so that when the text rises completely, the level is changed is to check the duration in your animation timeline, whatever duration it is, we put it in a delay inside of the event construct, after playing the animation of the text, and, when this ends delay, open another level.

On the other hand, the logic with the button is that when you press it, the level it points to opens.

5.2.5.6.4. Challenge passed and failed screens, saved game, and you can't use the item.

This is a series of Widget Blueprints with a message in their designer. When they are created, they have in their internal logic a delay in their event construct, that when a couple of seconds pass, causes them to self-destruct.

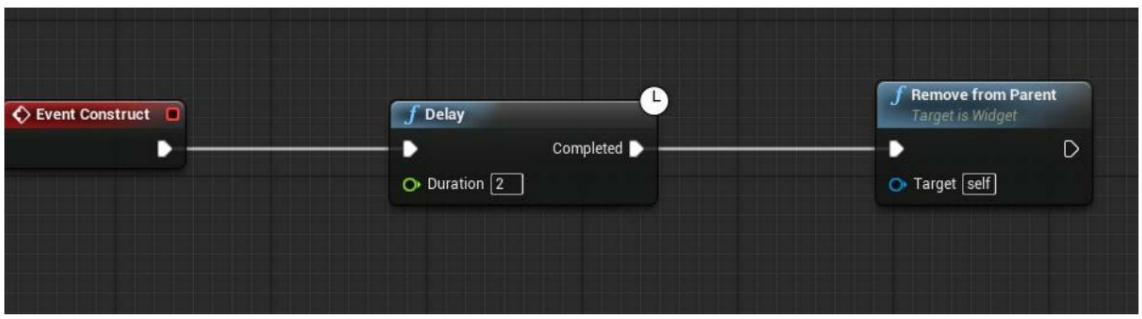


Figure 158. Screenshot of a Blueprint Widget with the indicated functionality.

Source: self made.

5.2.6. Implementation of game mechanics.

In the case of this project, all game mechanics work in a similar way. We have
to take into account that each different mechanic will be a different blueprint, which we can drag
to the stage (or spawn it dynamically).

In any case, for each of the mechanics, what is necessary in essentially any of the
them is what triggers the functionality for which it is programmed, and in this project,
all the mechanics activate their functionality through triggers with which the player collides.
As an extra, it is possible to accompany these triggers with a mesh that allows the player to recognize
that he is facing an element that is not a simple ornament.

That said, the most detailed and specific operation of each mechanic included in the project is
presented below:

5.2.6.1. teleporter

The teleporters are found in the forest of the island that is further south in the
game map, and in the maze to the north of the map.

The logic of this mechanic is to move the character's actor to another point on the map.

It has been decided to implement this functionality with the creation of two blueprints. a blueprint
teleporter, which if the player collides with, they will teleport, and a destination blueprint of
teleport, in which the player will appear if he collides with the previous blueprint. through the
teleporter blueprint, we indicate to which blueprint actor on the map the teleporter should be teleported.
player.

Described its functionality, through blueprints, once both have been created as blueprints of type
actor, what we need to do, on the one hand, in the teleporter blueprint:

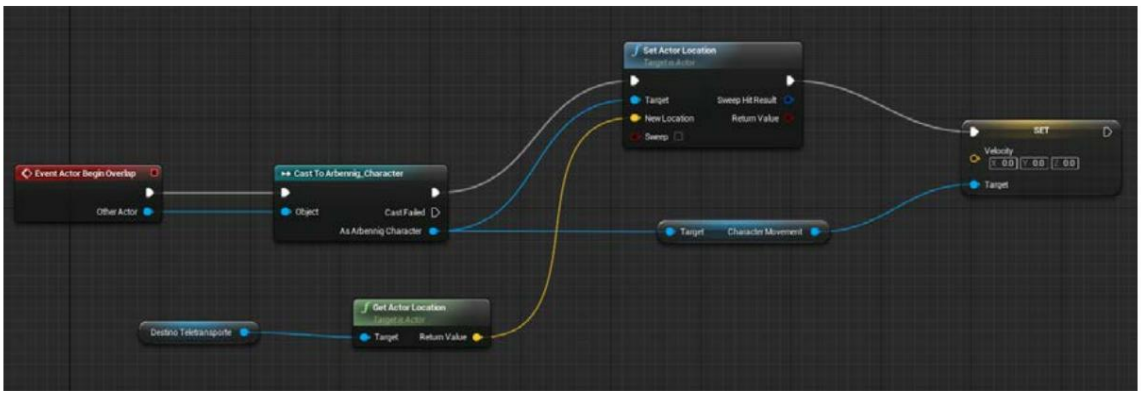


Figure 159. Blueprint capture of the teleport class logic.

Source: self made.

1. Add an event when the player interacts with the teleporter trigger.
2. From this event, we get the colliding actor, and cast it to
   Arbennig_Character, that is, our character.
3. From this cast, from which we get our character, we click on the variable pin
   and we call the Set Actor Location node, to set the new position.
4. To get the destination position, we create a blueprint variable of the destination class
   of teleportation (the other blueprint) through the variable details tab, and make it editable
   by checking the editable checkbox (when we make an editable variable, it is possible to
   access it from the level editor) to be able through the editor
   initialize it indicating to which destination the teleportation corresponds. This variable
   we drag to the graph, indicating that we want to obtain it as Get, and from it, we drag its
   pin to obtain a Get Actor Location node (which will give us the Destination position), and
   Finally, we drag the value provided by this node to the New Location input of the
   Set Actor Location node that we called in step 3.
5. Dragging again from our character variable, we get its Character
   Movement, and from this, we call Set Velocity, so that if the player was in
   movement, its speed becomes 0.

In the teleport destination blueprint, for gameplay reasons:

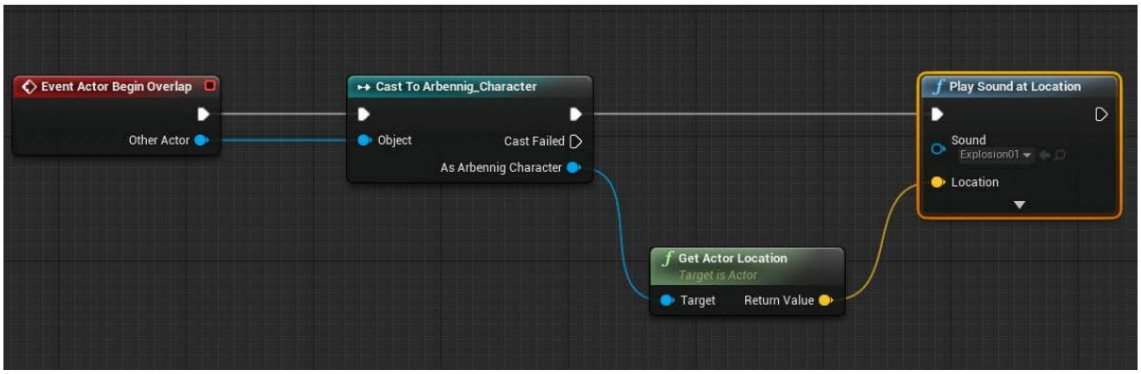


Figure 160. Capture of the Teleport Destination blueprint.

Source: self made.

1. Add an event when the player interacts with the teleporter trigger (when
   the player has teleported to this position, it will activate).
2. From this event, we get the colliding actor, and cast it to
   Arbennig_Character, that is, our character.
3. We call the Play Sound at Location node, and choose a sound from our content
   browser and set the position where the player is using a get node
   actor location taken from the cast.

5.2.6.2. Wait

Rings appear when the player accesses one of the flight challenges in the game after speaking with temple totem 4 or 7 and accept the quest.

The player sees a timer appear and an informative text of the rings obtained in the game. screen, all the hoops are activated and become visible, and you must go through each of these before the time is up. When the player passes through a hoop, the player performs a slight animation (rotates faster) and after a couple of seconds, it disappears. Also, if the time reaches 0, the hoops disappear, and must be restored for a next try if the player wishes. retry. Also, when the flight challenge ends, the player is teleported to the location where the corresponding totem is located.

Ring logic makes use of 4 main blueprints:

1. **Blueprint of type Widget Blueprint** that we are going to create so that it overlaps in the interface given the moment, and thus not have the HUD overloaded with logic. Here we take care to keep the remaining time counter and the collected rings up to date with respect to the rest. In addition to that, through this blueprint we will check if the challenge ends or not, if it has been successful or not, and, depending on this, indicate it so that in the The blueprint of the level will proceed to give the player the rune or not. Finally, it is through of this blueprint where it has been decided to teleport the player to the place where find the totem
2. **Hoop Blueprint.** This blueprint, of which two versions have been established, one more simple, and a more complex one with particle effects, is located on the stage spinning at a certain speed. When the player collides with the trigger inside it, it starts spinning faster, and after two seconds it disappears. In addition, it resets maximum the player's energy, and registers that it has been traversed in a variable so that you can proceed in the blueprint of the level.
3. **Character blueprint.** It is accessed through the different blueprints to the character to modify its rune, energy, position, and state variables.
4. **Game level blueprint.** With this blueprint we are going to start the hoops challenges, thus blocking the completion of any other challenges, and reset their values in case of not be outdone. In addition, we will send information to the Timer Widget Blueprint, to tell it the updated number of hoops crossed, and finally, also we will be checking if the challenge should end, to give the player the rune, and reset the game so that other challenges can be done.

Next, we proceed to explain at a logical level the most relevant steps at the time of having made each of the created blueprints (therefore, not the character, which has already been created previously, and no code is added to it).

Blueprint timer widget.

In this blueprint we have a text widget for the timer, another one for the rings obtained regarding the totals, and a couple more to show a message of challenge completed or not. These texts include a background image to stand out more.

What we do is bind the content of the timer text and the rings obtained. We associate this bind with some variables of type integer, which we will first cast to string to be able to append (it allows us to join strings), and thus add “s” of remaining seconds, or “/” and the number total rings. On the other hand, we bind the visibility of the passed or failed challenge texts, assigning them to a variable of type ESlate Visibility, so that we have control of this when it is necessary to show it.

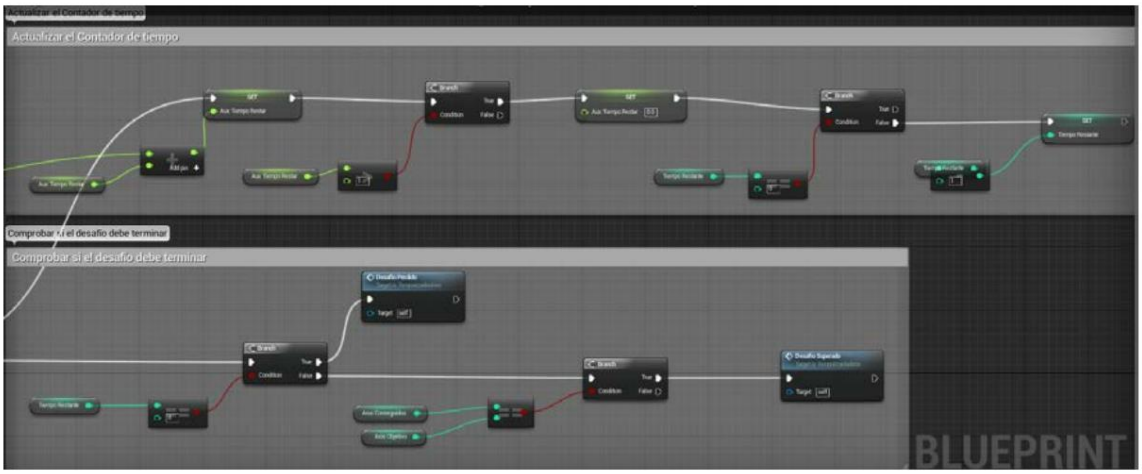


Figure 161. Capture the graph logic of the Blueprint Widget

Source: self made.

As for the main graph of this blueprint, we have starting from a tick function that will go calling a sequence of two pins while the challenge is not finished. On these two pins, the first is responsible for updating the time counter (depending on the delta time, it adds to an auxiliary variable, when the value reaches one, a unit is subtracted from the counter, and the auxiliary is reset to 0), and the second takes care of checking if the remaining time is 0, to call a Lost Challenge Custom Event, and checking in case of not running out of time, if the number of hoops achieved is equal to the number of target hoops, which, if so, would call another Custom Event, but challenge overcome.

The custom events of challenge lost and overcome, the first thing they do is change the visible challenge text passed, and its image background, after this, set the challenge variable to true finished, and a true or false variable indicating whether the challenge has finished successfully. To this a delay of 3 seconds follows, so that the player can appreciate the text that he has passed or not the challenge. Once the delay time has elapsed, the character's state is changed, whatever be (probably flight), to falling, so that it is set automatically to walking (and it seems that we've been teleported and fall), and finally we call remove from parent to remove the blueprint widget.

Blueprint del Aro.

In this blueprint, let's remember, we have two (although the only difference is that one has a mesh, and another has a particle effect around the trigger), what we do is have a Tick function, which is constantly turning the hoop by local rotation, which is applied multiplying the delta time by a value of your choice, creating a rotator with this on one axis, and applying it to the mesh with Add Local Rotation. We include before applying the rotation, a check if the player has gone through the trigger, so that if it is the case, the multiplicative become another larger number.

When the player walks through the trigger, it fires the OnComponentBeginOverlap event, in which we check if the trigger has already been crossed, we activate the bool variable so that the ring rotates faster, and after this, we recharge the player's energy (getting the character with the Get Player Character function and casting Arbennig Character), we set the variable to true indicating that the ring has been crossed, and we make a delay of a couple of seconds (so that the ring has 2 seconds to rotate fast). After the delay, we set to false the variable that indicates make the hoop spin fast, and we hide the actor with Set Actor Hidden in Game.

Game level blueprint.

For this part of the hoops, we do is use the begin play event, and the tick event in this level blueprint, plus use of the event to activate the totem challenge.

In begin play we get references to the hoops that we have placed on the stage, we create a array with them, and we return that array to a variable that we will access in the tick function repeatedly. The rings of each challenge must go in different arrays.

On the other hand, in the tick function, we perform a series of checks, consisting of check if the ring challenge has been activated (we activate it when accepting the challenge by the totem), and whether or not the challenge components have already been initialized. Having said that, we initialize

challenge components first (since we haven't initialized any yet), where in each
ring of the array we change its visibility from hidden to visible, we create the blueprint widget of the
timer and assign it to a variable to have references to it in a comfortable way, and
we add to the viewport, and then accessing this widget, we initialize the rings to 0
achieved, we indicate the target rings by making a length of the array of rings, we assign the
challenge time, and lastly, the teleport destination. At the end of this initialization part,
we set to true a variable that indicates that we have initialized the components, and we start in the tick
function itself, where we are going to update the obtained hoops and check if and how the challenge
has finished.

To update the rings, we have chosen to make a foreach loop of the array of rings, so that
for each element, we check whether its traversed variable is true or not, and the
we add to a variable of rings obtained. When the loop ends, we access the variable
of rings obtained from the blueprint widget, and we set the result, ending up restarting
to 0 our variable of obtained hoops, which being in the tick function, will add values
constantly.

As for checking whether the challenge has ended or not, what we do is access the
blueprint widget and check if the variable bool that we had created to indicate that the challenge
is terminated is true or false. If true, we check if your bool challenge variable
passed with success is also true or false, and regardless of this, we set it to false
the variable indicating that our challenge is active.

If the player has successfully completed the challenge, we access the rune 4 variable to place it
to true, stored in the character of our character (we access it with the function get player
character, and from this, making a cast to Arbennig_Character), and finally, we place our
totem challenge enumerator variable to none, so that more challenges are possible.

In case the player has not successfully finished the challenge, the variable is restored to false
to initialize the components, so that it is possible to reinitialize them if you want to retry
challenge not overcome. In addition, the visibility of all rings is changed to hidden (if we have
overcome the challenge, they would all be hidden), and also, performing a foreach loop of the array
of hoops, we set to false the variable that indicates that the hoops have been traversed. Finally,
we reset the variable of rings achieved to 0, so that when we start the challenge again, the
first value of crossed rings is not the one of the previous game, and we already establish in none the
enumerated variable that indicates the challenge we are in.

Regarding the logic that contains the event of activating the totem challenge, simply
we set the enumerator of the challenge to which we are going to perform (4 or 7), and we set to true the
hoops challenge trigger variable.

For the hoops challenge with the other guy, the procedures are exactly the same.

5.2.6.3. NPC dialogues and totems

Dialogues that appear in the game are shown when the player interacts with any NPC (town villagers, tree, and totems). They consist of a dialog box, where in the part At the top is the message from the NPC, and below it are the available responses of the player.

With this system the player and the NPC engage in a multi-response conversation. In this project, our character can talk to the NPC of the village and the tree infinitely, but with totems, once the challenge is accepted and passed, the totem stops being able to speak, because "His soul has been set free."

To implement this dialog system, it is necessary to create two widgets (a option widget, and a dialog screen widget), three structs (dialog tree, NPC response, and character options), and an actor (dialogue system). Also, you need to do a couple of calls to the Blueprint of our character.

How all these structures and classes fit together is as follows:

- The dialog system actor contains the dialog screen widget (it's only something merely visual, which will serve as a container for the different responses, both from the NPC as of the character), which, in turn, will dynamically contain the widget of responses to the NPC (consists of a button with a text on it).
- The dialog tree struct contains the NPC's response struct, which, in turn, contains the character's options struct.
- When we interact with an NPC or totem, from the character's blueprint we will having to enable controls on the user interface, in addition to spawning the actor that we want to show.

To enable controls over the UI, we create an InitMouse function, in which we create a reference to our player controller, and from it, we set the mode SetInputModeUIOnly, in addition to modifying the bool variable to show or not the red crosshair of the hud, show cursor, and enable mouse events.

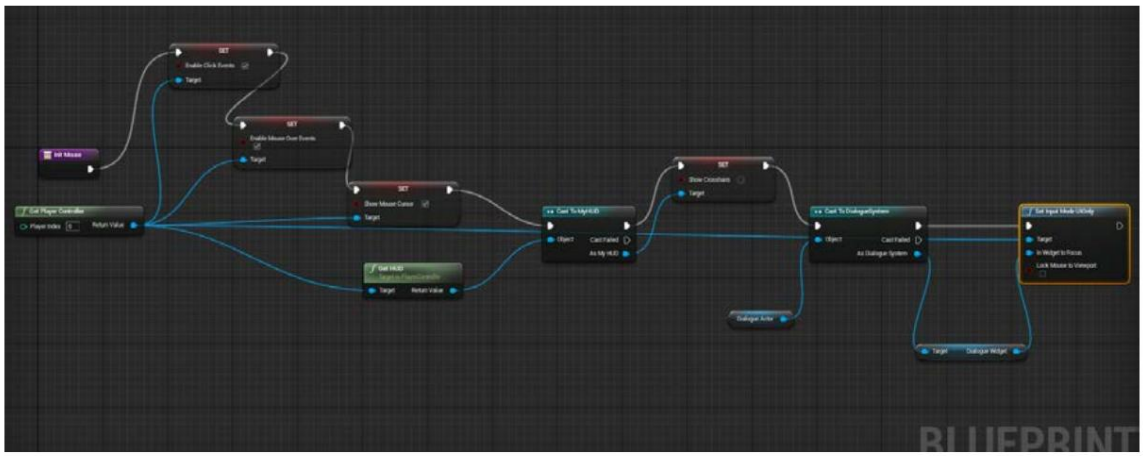


Figure 162. Capture of the initMouse function

Source: self made

On the other hand, to spawn the actor, we create another function, SpawnDialogue, and in it we simply call the SpawnActorFromClass node. As class, we choose the Blueprint class DialogueSystem (dialog system), and finally, we save the variable of this created actor. The position where the actor spawns does not matter, because when it is created, the widget appears in the viewport. Whenever you talk to any NPC, you can easily access it like this to the dialogue actor through our character.

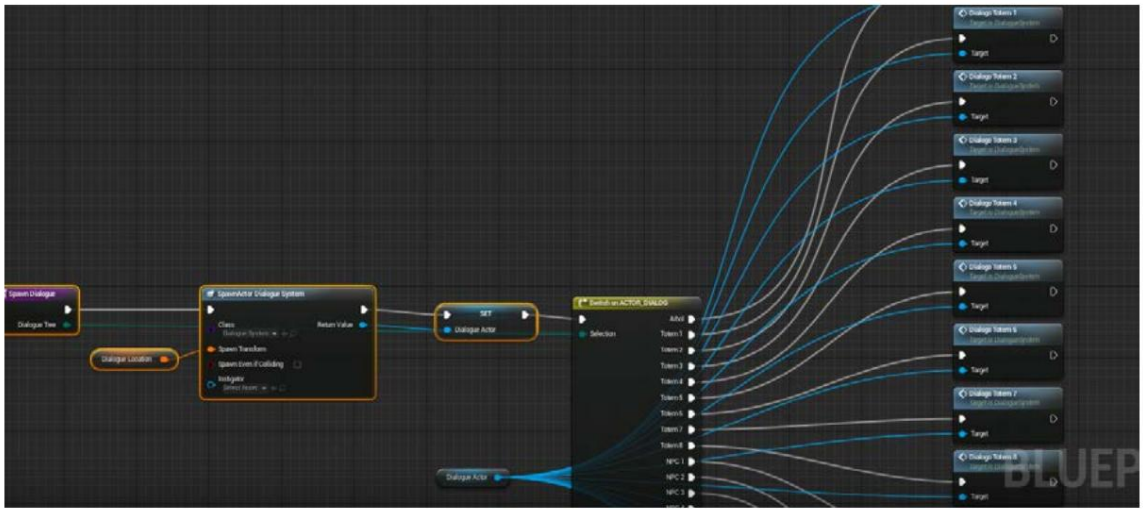


Figure 163. Capture of the SpawnDialogue function

Source: self made.

As for the struct, they contain the following:

- Character Options Struct (PC_Choices). Contains Reply (response), LinkToNode (link to the response node it points to), and exitDialogue (bool to determine if with this option ends the conversation).

- NPC Response Struct (NPC_Reply). Contains NPC_Reply (the reply from the NPC), and an array of PC_Choices.
- Struct of the dialog tree (DialogueTree). Contains a Conversation variable, which is un array de NPC_Reply.

As far as Widgets are concerned, the Dialog System Widget does not have any logic in it. its class, we simply have here a container of type border, which includes a text inside (the response from the NPC), and another border, which in turn has a Vertical Box inside (where the answers available to the player to be added dynamically), but it is necessary indicate that both the text and the vertical box are variables, and make them public so that they can be edited from outside. In the responses Widget we must make the text of the response public variable. button, and, at a logical level, what we do is get the DialogueActor from our Character, and call a function that we will create in this actor, named LoadDialogue, which receives as LinkToNode (the node the option points to), and ExitConversation parameters, and what it will do this function is to load/reload the game's dialog system every time we press a option.

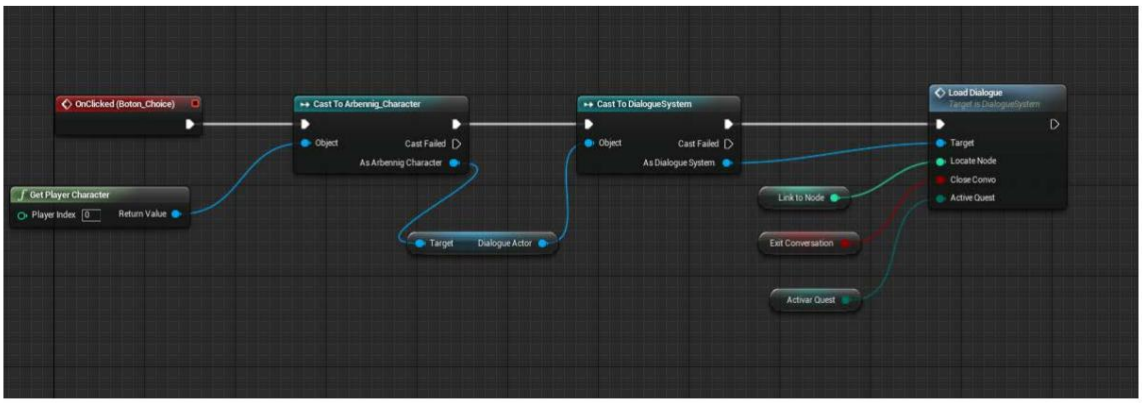


Figure 164. Logic capture of the Dialog Options Widget.

Finally, the DialogueSystem class. In this class we have logic in the main graph, and in two functions we created, one is LoadDialogue, already mentioned, and the other is StartConversation, where all we do is call the LoadDialogue function.

In the main graph of the class we create variables of data structures of the type dialogueTree for each distinct conversation and a general one to assign this value to. Apart from this, on the one hand, we have a series of custom events, which depending on the NPC or totem with the that we talk will skip one or the other, thus assigning a value for a talk tree struct or another. On the other hand, through the begin play event, which will be called once the spawn of this actor, we create a DialogueScreen widget, we store its value in a variable, and we add it to the viewport with its add to viewport node, finally, we call the function StartConversation.

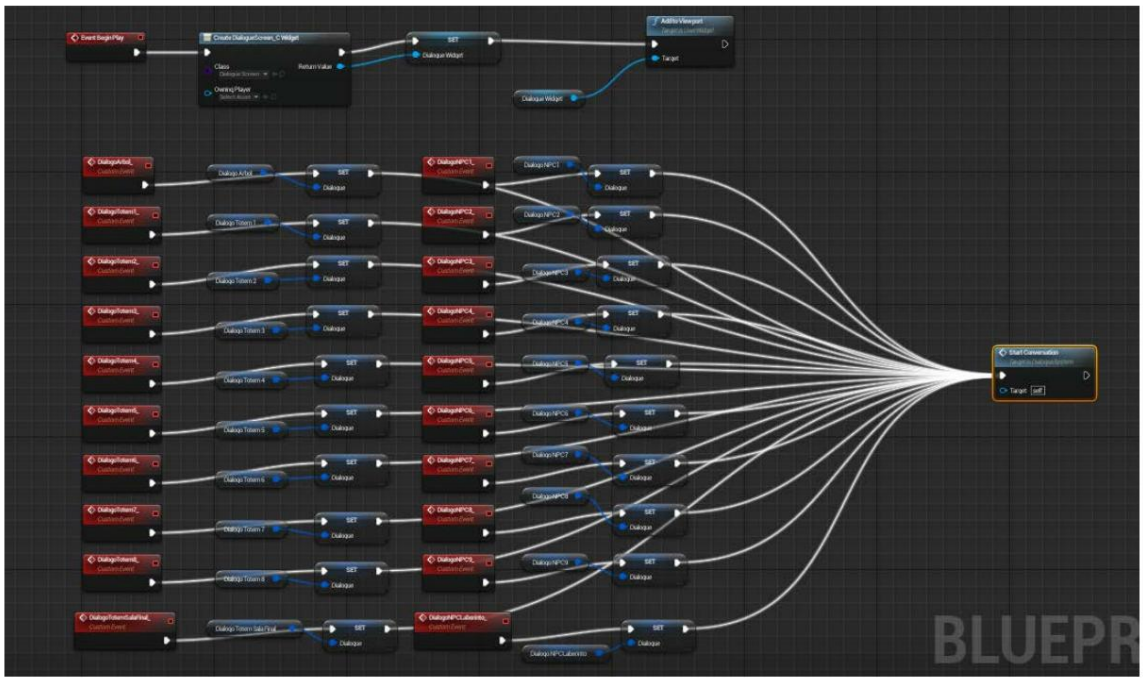


Figure 165. Graph node of the DialogueSystem class.

Source: self made

Finally, in the loadDialogue function, what we do is call a branch with the condition of exit that starts as input parameter of the function. If it is true, it means that the conversation is over, and we'll call setVisibility of the DialogueScreen Widget, in addition to call the character's custom event to reactivate the controls.

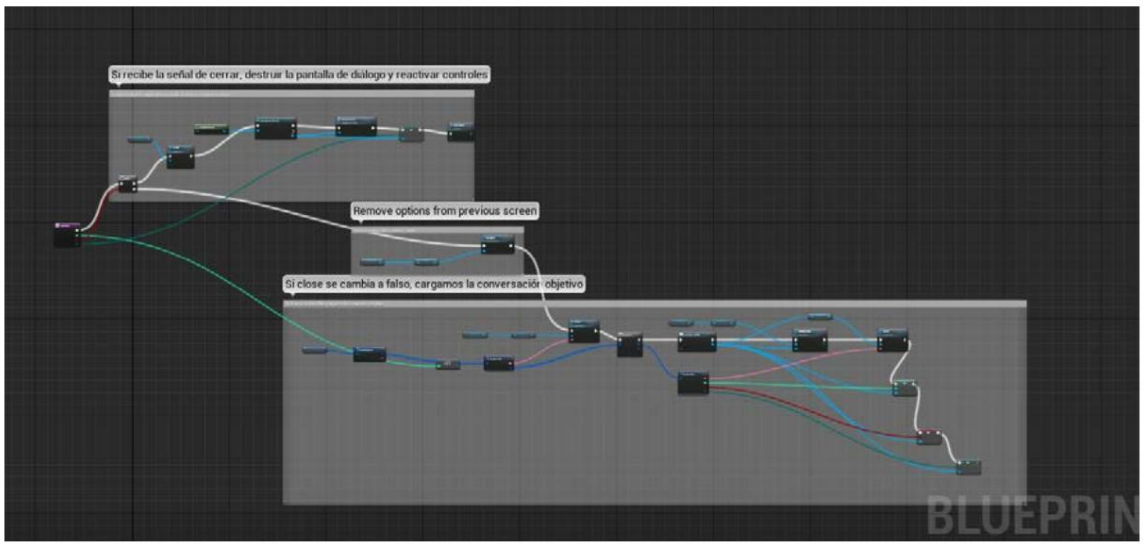


Figure 166. LoadDialogue function complete.

Source: self made.

If the branch is false, what we do is load everything in the DialogueScreen widget that we have, but first, we access the vertical box container of the conversation options, and we clean it with clear children. To load the conversation properly, we use the other input parameter of the function, which corresponds to the conversation node to be read (with node we mean, for example, an NPC response, and the options for this response), and with this, using the conversation struct that we have assigned in the main graph based on from the NPC we talk to, we get the node.of the conversation (we call the get function of the struct, and its index is the value of the node we want). Following this, we break this conversation node, which is a struct of NPC_Reply, and with it, we assign the value of the text of response from the NPC (we use the SetText function for this).

Finally, we have to add the player's response options to the vertical box that we have prepared for it, and we do this by calling a foreach loop of the PC_Choices array that we have (when we have broken the struct NPC_Reply, remember that it contains a NPC_Text and an array of PC_Choices), and for each element of it, we call the Add Child node Vertical Box, and assign from it the correct Text, the conversation node pointed to by the response, and the value of the variable bool of ExitConversation.

5.2.6.4. Door at the end of the game.

The game's end gate mechanic contains its functionality in two parts, a trigger that will send the notice to the class of the level that the door is ready to open, and a trigger of the door, which contains through an animation with timeline, opens it.

Starting with the trigger class, which contains a light that turns on and off based on whether the player is on the trigger or not, what we do is detect first, when it collides with the trigger, that it is possible for the player to open the door (because it is within the radius of action trigger).

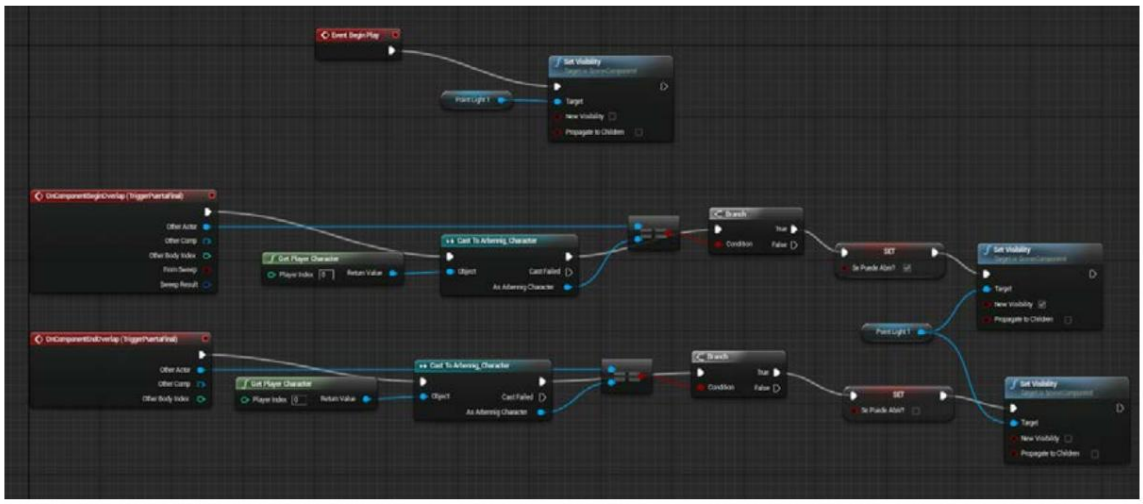


Figure 167. Capture of the initialization and response to the events when the character traverses and stop going through the trigger.

Source: self made.

Knowing if it is inside the trigger or not, inside the tick event we will check it, and also if the player is trying to interact with it, which, if so, we will check if it has all the keys, to send the signal to the level blueprint that the door has to be opened.

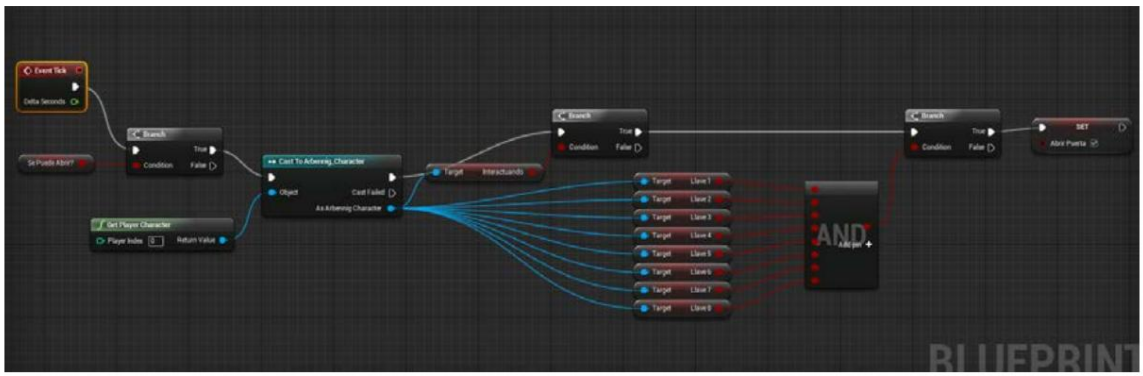


Figure 168. Capture of the logic of the trigger to open the door in the tick function.

Source: self made.

Now, in the LevelBlueprint, what we do is have the tick function check whether We have modified or not the variable that acts as a flag so that the door opens and, if so, we send the signal to the door to open using another flag.

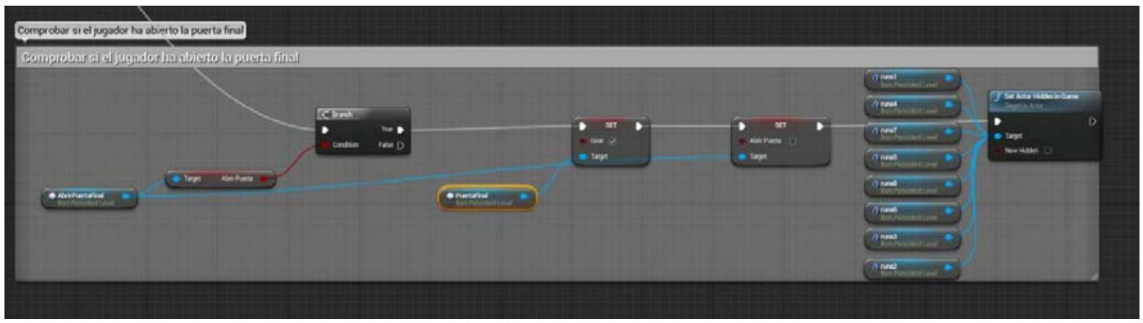


Figure 169. Capture of the logic of opening the door in the level blueprint.

Source: self made.

Lastly, inside the gate blueprint, the first thing we do is its begin event play, save the initial rotation of this, which we need to be able to use the timeline. And finally, within the checks with the tick event, we check if the flag of that is open has been updated to true, which, if so, will use a timeline, which is a component of blueprints to make simple animations, and relying on an interpolation, we will rotate the door until it opens.

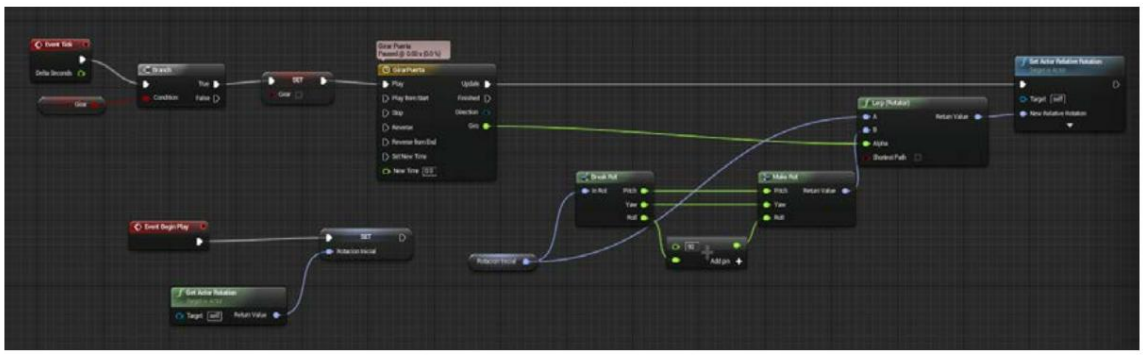


Figure 170. Logic capture of the final gate class.

Source: self made.

As an extra detail about the timelines, their operation consists of an animation curve which can make the value of a variable or a vector vary according to it. In our case, since we only need the door to turn on one axis, with a float variable as a parameter of output that modifies its value is enough for us.

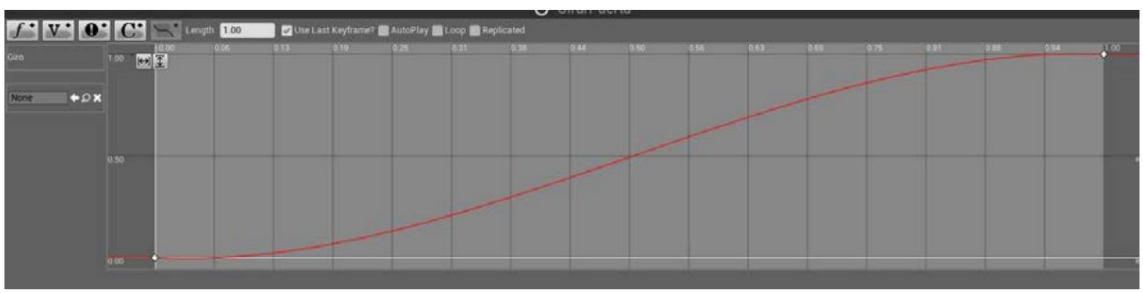


Figure 171. Capture of the timeline of turning the door.

Source: self made.

5.2.7. Game save/load system.

The save system in Unreal Engine 4 is done by a blueprint that inherits from the class Save Game.

Figure 172. Screenshot of how to find the SaveGame class.

Source: self made.

With this class created, what we do is store in it all the data that we want to save, creating variables for it as if it were a normal blueprint. In our case, we we are interested in saving 8 booleans, one for each key, which will be true or false depending on whether the player has achieved them or not, we will also save a vector to store the position of the player in which you saved the game, a boolean to determine if there is a game or not saved, and finally, a string variable and another integer that we use to identify the file save file that is created. All these variables must be left visible for the rest of the classes, so they can access their values.

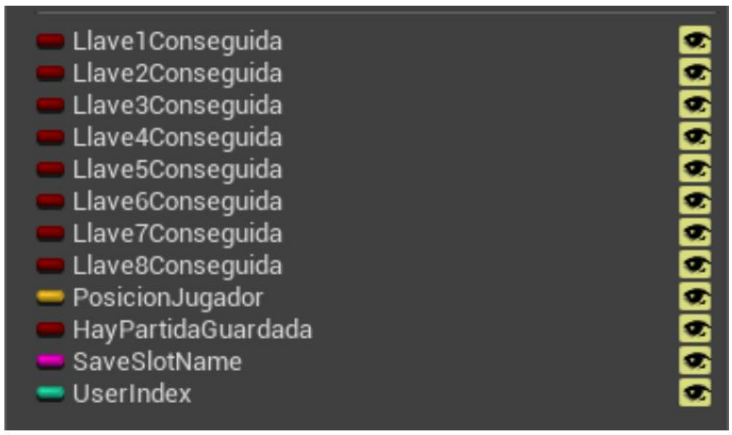


Figure 173. Capture of the variables stored in the blueprint.

Source: self made.

With this class, we can now save the game and load it.

5.2.7.1. Save game.

To save the game, a blueprint has been created consisting of a save point, which contains a position sphere where the player will appear. In it, when the player approaches, will set a flag to true that will indicate in its tick function that it can save the game.

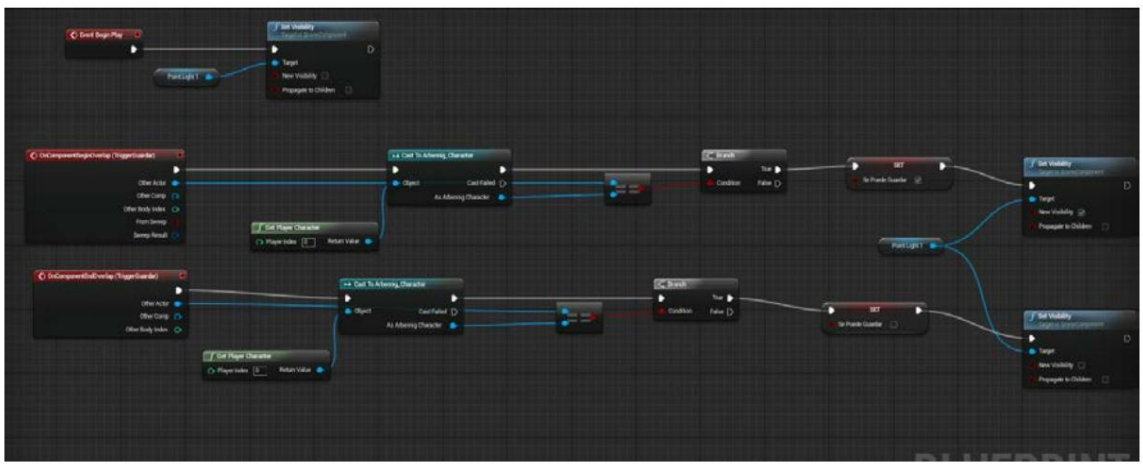


Figure 174. Logic to detect that the player is within the save game trigger.

Source: Personal elaboration.

This done, and after checking that the player is interacting, the logic of the saving system consists of calling the function Create Game Save Object to which we will indicate the blueprint class that we have created, then, from this object of type game knows, we make a cast to our blueprint and we will save that variable result of the cast for now access the properties it contains and modify them. Indicating that there is a saved game, accessing our character, with which, in turn, we will access its boolean keys achieved, saving in the position vector, that of the reappearance position sphere that

we have created in the blueprint, and finally, indicating that we are going to save the game in the index 0, and its name is going to be 0.

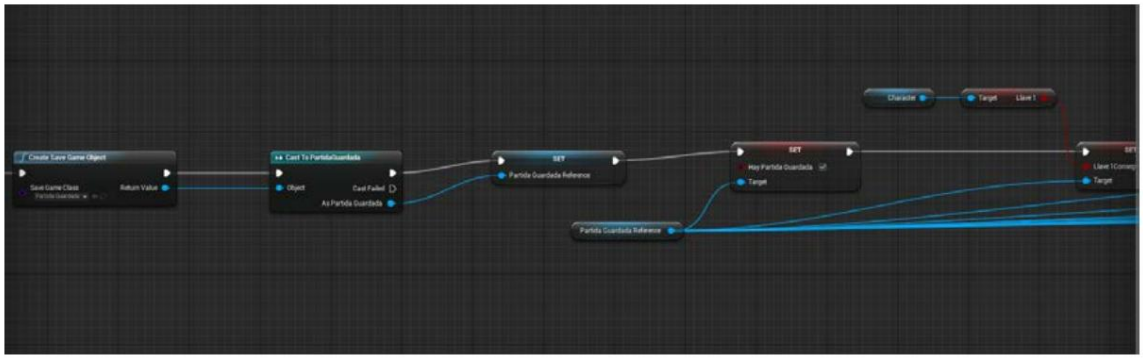


Figure 175. Capture of the creation of the save game object, its cast to saved game, and its start assigning values to your variables.

Source: self made.

When we have finished assigning the variables, we call the function save game to slot, and passing the new object that we have saved game, we already have the game stored.

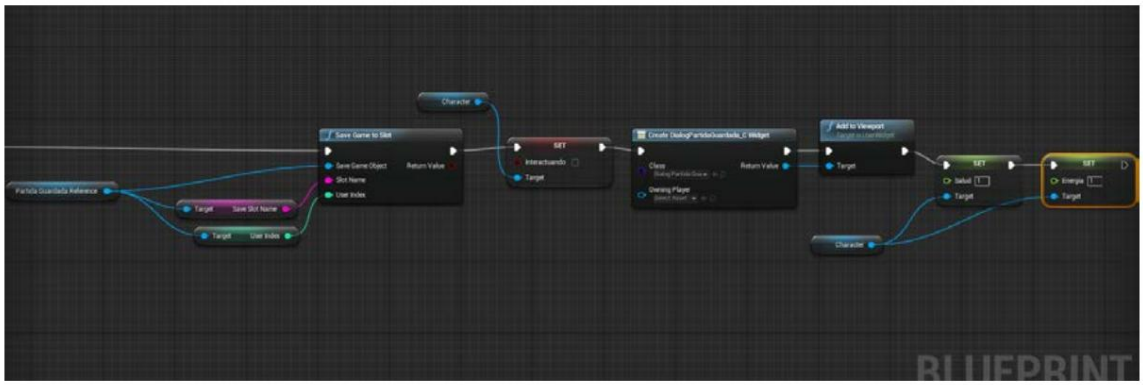


Figure 176. Capture of the call to the save game to slot function.

Source: self made.

After this, what we do in this save point blueprint is call the Informational Save Game WidgetBlueprint, and restore player health and energy to full character.

5.2.7.2. Load game.

The loading process of the game follows a fairly similar process to the save process.

What we do to load a game is create a save game object, cast it to our blueprint, and store this cast (as in save). Once this is done, we call the function load game from slot, where we will indicate 0 in name and 0 in user index (as has been decided

save the game), and the object that will be returned to us, we cast it to the blueprint of our save class game, and assign it to the reference we already had.

After this, we can now access all the variables that we had saved.

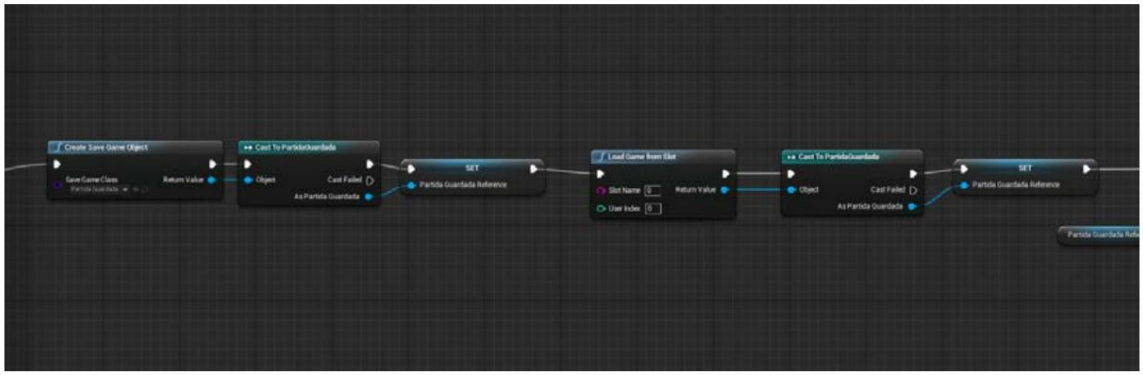


Figure 177. Loading a game.

Source: self made.

5.2.7.3. Delete game.

To delete the game, it is only necessary to call the delete game in slot function, indicating the slot and the user index, which in our case is 0 and 0.

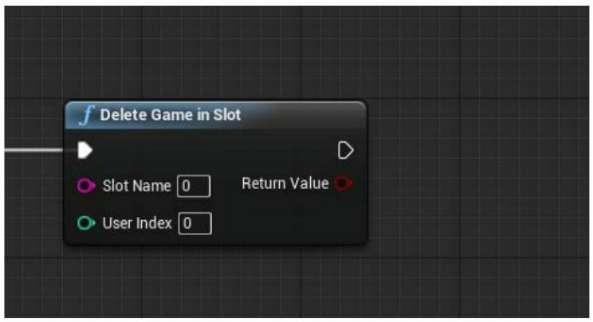


Figure 178. Delete saved game function.

Source: self made.

5.2.8. AI implementation.

The implementation of the AI defined in the GDD requires the creation of a series of components basics in Unreal Engine 4, which we proceed to define and explain their set up basic, so that later we will focus on those particularities that distinguish each of the artificial intelligences described in the design document.

The basic components necessary for the creation of an AI in this engine are 4, which can increase depending on the complexity of the AI, they are the following:

1. A Behavior Tree, which allows us to establish behaviors based on a series of conditions, and is prepared for AI, in a way that allows a management of more efficient and graphic behaviors. This component is found in the section de Miscellaneous.
2. A Blackboard that serves us to store variables that we use with the Behavior Tree. Like the Behavior Tree, this component is also found in the section of Miscellaneous.
3. A Blueprint of type Character. This Actor corresponds to the character that will be possessed by the Artificial Intelligence Controller, following the same Unreal scheme as we use in Arbennig, where our Player Controller has the Character Arbennig.
4. A Blueprint of type AIController that will own the Blueprint of type Character (could be of type actor if it is a very specific AI, but it is not our case).

Once these components are created, we begin to add their game logic. In our case, the logic of the AI game that we are going to use, consists of a character that remains still until sees the player, and then chases after him until he catches up.

For the AI character, what we're going to do is add a trigger that covers the player. To this trigger we're going to give it the functionality that if it collides with the player, then the player changes its position to an indicated one, which corresponds to the area where it will appear for having lost the match.

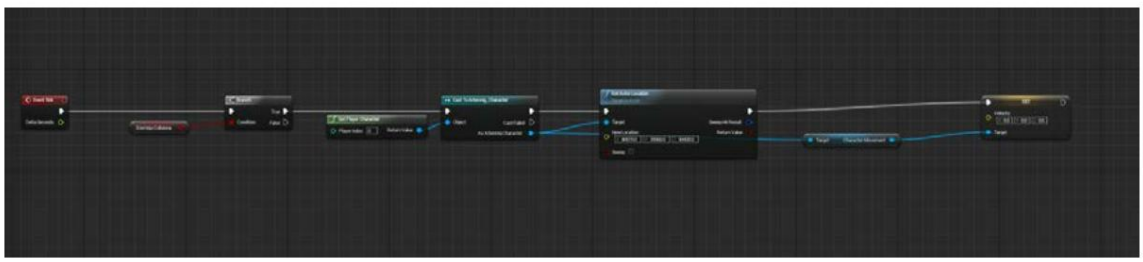


Figure 179. Capture of when the enemy collides, we change its position to the indicated one.

Source: self made.

On the other hand, in the blackboard, we create 3 variables, one of type object that corresponds to the target to be pursued, and two vectors, one for the current position, and one for the current position. target to move to.

When we create these variables, we save and assign the blackboard to the Behavior tree.

The behavior tree is a graph with a special function. In it, there is an initial node called root, and from this node other nodes go down, which will be executed from left to right. In In this graph it is possible to create selectors and sequences, which execute the child nodes from left to right.

right until one ends with success (selector) or failure (sequence), which causes it to go up in the tree. The conditions and checks that are performed on the behavior tree receive the name of decorators (conditional) and services (to update the blackboard and do checks), while the tasks that we want to be carried out when certain characteristics are met, they are called tasks.

Our Behavior tree is as follows:

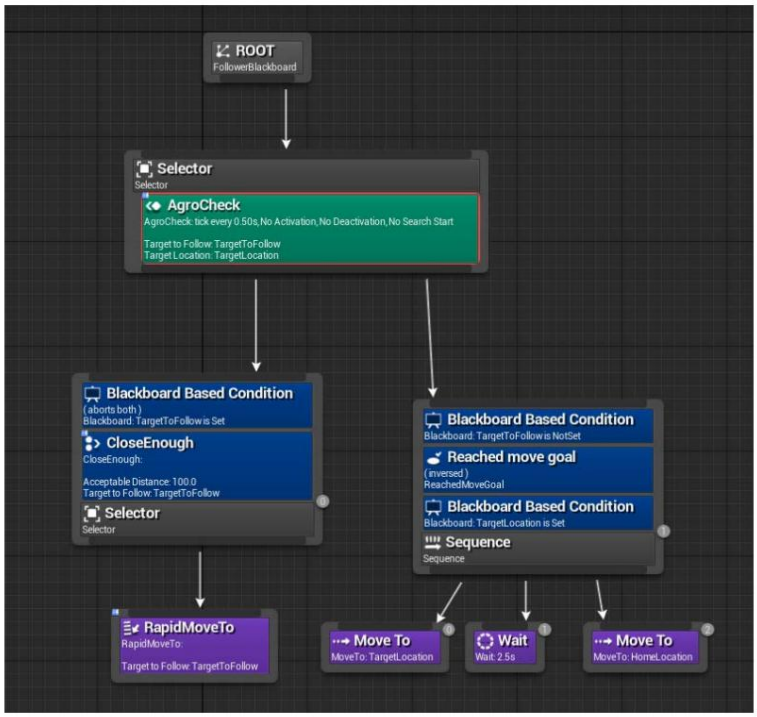


Figure 180. Capture of the general logic of the Behavior Tree of our AI.

Source: self made.

It consists in that, from the root node, we have a selector that at each instant will check if the Pawn that the AI is going to possess is assigned, and, if not, it will assign it. In addition, it is also responsible for draw a series of lines to spot a (missable) target to chase. of this two blackboards start, each one to check if the player has been assigned or not. In case of been assigned, and that it is close enough (Decorator CloseEnough that check distances), then a chase task is launched. On the other hand, if the player does not is assigned, and has not yet reached its destination, but has a destination to move to (in the one that has lost sight of us), what it does is go to the position it had planned to go, wait 2.5 seconds, and then return to the initial position where it was.

For all this to work, from the editor, what we do is create a nav mesh bound element, which will be the navigation mesh that the character will run through.

5.2.9. Implementation of the general game flow of the game in the Level Blueprint

The general game flow in the level blueprint is divided into several parts.

First of all, when the level is created, the begin play event is called, which does the following:

1. Let the PlayerController own the Pawn and store it in a variable.
2. Load the game.
3. Create an array of elements for those cases where we have several objects of a same class (for the hoops and for the AI), and hide all the elements of the stage that should be (hoops from challenges 4 and 7, flyers from challenge 2, golems from challenge 6, challenge 8 flag, and the runes on the rune wall).

Once this is done, the tick function will check if we are in a challenge or not. For this we have created an enumerator with 9 values, 8 to indicate the challenges, and one more, which is deals with the none value, for when it is none. Since the character class we are going to use as a bridge for its easy access from any blueprint of the game, we create a variable both in the level blueprint and in the character of this type of enumerator.

At each instant, the logic we follow is as follows:

1. Check the value of the enumerator, and if it is different from none, it means that there is a challenge.
2. Check if the character enumerator variable is already updated to the challenge current, and if not, assign the value to that of our level blueprint.
3. Initialize the corresponding challenge.
    a. For challenges 1, 3, and 5, you don't need to initialize anything.
    b. For challenge 2, the brochures that we are going to deliver in the town are made visible.
    c. For challenges 4 and 7 we created the hoop timer widget, which store it in a variable, and add it to the viewport, then, like this widget we initialize the achieved hoops to 0, we assign the target hoops to achieve by passing the length of the array of rings that we have created in begin play, we indicate the time to overcome the challenge, and finally, the destination of teleport for when the challenge ends.
    d. For challenge 6 we make visible the flag that the player must get and the golems.
    and. For challenge 8, the flag that the player must get is made visible.

4. Check if the conditions to finish each of the challenges are met, and
when so, call a custom event that is responsible for spawning the rune
corresponding to the challenge that has been passed, and display the Challenge Blueprint Widget
overcome.

a. Challenges 1 and 5 directly call the end challenge event, because the
challenges start when you finish the conversation with the totem.

b. Challenge 2 calls to end challenge when all handouts have been
delivered. To determine this, we have a flag that is set to true when a brochure has been
delivered, and also a variable with the reference to the trigger.
where the player is, when a flyer is delivered, we add it to an array of
booleans in the level blueprint, and we destroy the trigger that the character is in so that
it cannot deliver brochures to it again, when the length
of the array reaches the number of leaflets, then it means that all the
brochures have been handed out.

c. Challenge 3 does so when all bushes have been placed where indicated
on the mission. At the end of challenge 3, because the trigger where they were placed already
contains them all, a variable created in the character is set to true, which
we read with the level blueprint.

d. Challenges 4 and 7 call the end challenge event when they are through
all the hoops in the indicated time. The logic of these challenges consists in
determine when a loop has been traversed, which will set a flag to true, so
so that at each instant, the array of rings is traversed, and added to a variable
integer +1 for each loop flag traversed, and, on completion of traversal, updates
the timer widget variable and the remaining rings. when the widget
notifies us that the challenge has ended and how, if it has been successful, call the
end challenge event, while if it ends without overcoming it, everything is reset,
setting totem challenge enumerators to none, setting to false
all the variables of hoops crossed, and hiding the hoops in the game.

and. For challenge 6 we call finish challenge when a variable created in
our character is set to true, after having placed where the character tells us
totem the flag to be collected. Also, the actors of the
challenge if it has been overcome. On the other hand, in case the challenge has not been
exceeded (which is indicated with another variable from the character), what we do
is to restart the challenge, positioning and hiding the flag again in its place,
and the same with golems. In addition to also changing the enumerators to
state of none, and show a widget that the challenge has not been passed.

5. Check if the player dies, and if so, open the endgame level and remove
el HUD.
6. Check if a rune has been spawned, and if so check if the player has spawned it
picked up already or not by checking if it is a valid object. Depending on the challenge that was the
we were doing, we destroyed the trigger that allowed us to talk to the totem of said
challenge, and after that, we set the active challenge enumerators to none.
7. Finally, we check if the player has opened the final door or not, that is,
checking if the boolean that we have in the trigger to place the runes has been set to true, and if it is
true, we call the door to rotate and make the glued runes visible
to the final wall.

6. Conclusions

After having finished the project, we can highlight a series of conclusions about it and
also, carry out an evaluation on the tools used after several months of use.

It should be noted that it has been possible to comply to a greater or lesser extent with the postulated objectives
in section 3 of the report, that is, the complete design of a video game and a
Once created, it has been implemented using the Unreal Engine 4 engine, covering the aspects
creation of the game world and its complete logic, it has also added game menus,
cinematics of prologues and epilogues, has been made compatible with the peripheral oculus rift DK1, and is
has sounded the project. With all this, we can determine that the result obtained has been
Right.

Apart from this, after several months of development using the blueprint scripting system, it was
can affirm that it is an extremely powerful system, and that it allows us to make video games
100% complete without having to touch any code. Although yes, the blueprint system
it is graphical scripting, that is, what we have in the code is in Blueprints represented with
nodes to have everything in a more visual way. And with this I want to emphasize that a person
with no programming knowledge or just basic knowledge, you will probably
have very complicated to develop projects, so that the profiles of designers or
artists who want to make use of it, must have knowledge of programming oriented to
objects and the execution logic that programs follow.

On the other hand, the engine has a learning curve that is a bit tricky in
some aspects progress. Although it is true that with the UDK engine it has improved
Undoubtedly, the learning curve offered by engines like Unity is relatively steeper.
simple, especially to start using it.

In terms of the possibilities offered by the engine, it is clear why it is one of the
most relevant and famous engines in the world, with all the tools that it includes, the interface
much easier to understand than in UDK, and the power and visual results that
lets show are quite remarkable.

It should be mentioned that, although he has previously carried out projects with other engines for
video games, including Unity and UDK, I had never done a project with UE4 before,
and all the mechanics and visual sections, and everything in general, has been learned from scratch in this
motor.

As a negative conclusion, I must indicate that during development, the engine has suffered constant forced closures, sometimes reaching the level of about 10 a day on the days that I dedicated myself to project completely. In addition to this, normally when the project is closed unexpectedly, it is not possible to recover the progress made, so it is very convenient save for every few changes made to the project.

In addition to this, it has been noted that although the capabilities of a multimedia engineer allow you to successfully complete a video game, the artistic skills you possess are relatively limited, unless you have a practice and love for them, so when it comes to developing video games with a more professional cut, it would be convenient to carry out some kind of complementary artistic training if you want to be able to develop projects for a single person.

6.1. Proposals for improvement and future work.

As proposals for improvement and future work, among other aspects, the main one would consist of perform a modeling and animation of the character in the first person, whose mesh and animation are those offered by default by the engine for FPS projects.

On the other hand, offering the player the possibility to control the volume of music and game sounds separately, replacing the free unreal assets used in the project with own assets created with a modeling program like 3ds max, improve the controls of game and adding Xbox controller support would also be proposals for improvement.

In addition, for future work, we will try to improve the skills where they have faltered, which are in the artistic section, or perform them together with people who have artistic skills professionals.

7. Bibliography and references

7.1. Introduction.

[1] History of video games:

https://es.wikipedia.org/wiki/Historia_de_los_videojuegos#Balance.2C_presente_y_futuro_de_los_videogames

[2] Video Game Billing:

http://elpais.com/diario/2008/04/09/radiotv/1207692005_850215.html

[3] Destiny most expensive video game in history:

http://www.abc.es/tecnologia/videojuegos/20140909/abci-destiny-gameplay-videojuego-RPGfPS-activision-201409091206.html

[4] Flappy Bird Success:

https://es.wikipedia.org/wiki/Flappy_Bird

[5] Capabilities Multimedia Engineer from the University of Alicante:

http://cvnet.cpd.ua.es/webcvnet/planestudio/planestudiond.aspx?plan=C205

7.2. Theoretical Framework or State of the Art.

[6] Video game definition by Bernard Suits:

The Grasshopper: Games, Life and Utopia, Bernard Suits, 1978.

[7] Video game definition by Wikipedia:

https://es.wikipedia.org/wiki/Videojuego

[8] History of engines

Game Engine Architecture, Jason Gregory, 2015 (second edition)

[9] Unreal Engine is created in 1998:

https://es.wikipedia.org/wiki/Unreal_Engine

[10] The Source engine is introduced in 2004:

https://es.wikipedia.org/wiki/Source

[11] List of game engines:

https://en.wikipedia.org/wiki/List_of_game_engines

[12] Logo Unreal Engine:

https://upload.wikimedia.org/wikipedia/commons/e/ee/Unreal_Engine_logo_and_wordmark.png

[13] Logo Unity:

https://upload.wikimedia.org/wikipedia/commons/5/55/Unity3D_Logo.jpg

[14] Logo CryEngine:

https://upload.wikimedia.org/wikipedia/commons/8/8d/CryEngine_Next_Gen(4th_Generation).png

[15] UE4 vs Unity vs CryEngine:

http://blog.digitaltutors.com/unity-udk-cryengine-game-engine-choose/

[16] Advice on engine selection in slides of the subject "Engines for video games" of Video games 2.

https://moodle2014-15.ua.es/moodle/pluginfile.php/18327/mod_resource/content/4/vii-08-engines.pdf

[17] Basic engine architecture

https://docs.unrealengine.com/latest/INT/Programming/UnrealArchitecture/index.html

[18] Image of motor class diagram

https://docs.unrealengine.com/latest/images/Gameplay/Framework/QuickReference/GameFramework.jpg

[19] Information about Blueprints

https://docs.unrealengine.com/latest/INT/Engine/Blueprints/index.html

https://docs.unrealengine.com/latest/INT/Engine/Blueprints/Overview/index.html

7.3. Methodology [20]

What is kanban and how is it used:

http://www.javiergarzas.com/2011/11/kanban.html

7.4. Video Game Design Document.

[21] GDD Template of Fundamentals of Video Games, by Fidel Aznar.

[22] PEGI logo:

https://upload.wikimedia.org/wikipedia/commons/thumb/4/44/PEGI_12.svg/426px-PEGI_12.svg.png

[23] PEGI12 information:

http://www.pegi.info/es/index/id/96/

[24] Gameplay definition by Wikipedia:

https://es.wikipedia.org/wiki/Jugabilidad

[25] The minimum software and hardware specifications are those described in the FAQ on the web of UE4:

https://www.unrealengine.com/faq

7.5. Development and implementation.

[26] Static mesh import pipeline in UE4:

https://docs.unrealengine.com/latest/INT/Engine/Content/FBX/StaticMeshes/index.html

7.5.1. Frequently consulted bibliographies

The Unreal Engine documentation website:

https://docs.unrealengine.com/latest/INT/index.html

The official Unreal Engine 4 youtube channel:

https://www.youtube.com/channel/UCBobmJyzsJ6Ll7UbfhI4iwQ

The Unreal Engine 4 forum:

https://forums.unrealengine.com/

UE4 AnswerHub:

https://answers.unrealengine.com/

UE4 Audio Specifications:

https://docs.unrealengine.com/latest/INT/Engine/Audio/WAV/index.html