UNIVERSITAT JAUME I
Escuela Superior de Tecnología y Ciencias Experimentales · ESTCE

Development of a 3D Tower Defense using Unreal Engine and Blender:

ASTROKEEPER

Juanma Pensado Ballester

Supervised by:
José Ribelles Miguel

29/07/2024

Bachelor's Degree in
Video Game Design and Development

A mis padres, abuelos y novio, gracias por
acompañarme mientras
convierto mi pasión en mi futuro.

Gracias a mis tíos, que en los momentos más complicados me ayudaron a tomar una de las decisiones más importantes de mi vida.

Gracias a mis primos y amigos que desde que tengo uso de razón fueron los que me acompañaron en mi pasión a los videojuegos

Y sobretodo gracias a esta increible comunidad que somos los DEVS por luchar por una industria altruista en la que todos nos ayudamos mutuamente, en concreto a Matt Aspland, Gorka games, Carlos Coronado y Ryan layle que sin sus tutoriales no habría podido salir esto adelante

ABSTRACT:

The document discusses the development process of a 3D Tower Defense game called ASTROKEEPER using Unreal Engine and Blender. The primary goal was to create an engaging and fully playable game with various levels and mechanics. Challenges were overcome, leading to the successful completion of a playable game available for download. The project showcases the developer's skills and determination to craft an original game in an unfamiliar engine.

KEYWORDS:

- ASTROKEEPER
- UNREAL ENGINE 5
- BLENDER
- 3D
- TOWER DEFENSE
- SHOOTER
- VIDEO GAME DEVELOPMENT

CONTENTS

INTRODUCTION

This chapter shows what the purpose of the work was in the beginning, why and how this project was going to be developed.

1.1 Work Motivation

Video Games are becoming more and more popular, becoming the first entertainment option paired with the films and series, but videogames have something that makes them unique, interactivity.

In the rise of video games, I wanted to make a game that showcases the love I have for video games intertwined with my will to improve my skills on game development.

At its core, my project is a 3D Tower Defense game where you're a stranded robot, fighting your way out of an unknown planet. This idea started when in "VJ-1227 Motors de Jocs" we (my group and I) made a similar game for a class project. But back then, I had to use free to use assets from the web due to time limitations.

Software wise, I really wanted to learn and improve my development skills using Unreal Engine 5, surfacing more with the Unity last controversy with the licensing prices, as Unity was the only game engine I really know how to use.
Now, I'm determined to craft something entirely original, by myself in an unknown Engine.

1.2 Objectives

The primary goal of this project is to make an engaging and fully playable game, with various levels and interesting mechanics, or at least making the game as I had on the prototype I made years ago in another engine.

While not having a specific research question, the project is made to fulfill and acquire new skills. By forcing myself to Unreal Engine 5 and leveling up my skills in Blender, I aim to acquire for myself the tools necessary for future work in the videogame industry.

1.3 Environment and initial state

At the beginning of this project I started to make it all by myself, to test my abilities and improve them and also as a project I could show in my portfolio without shame or insecurities.

The project didn't start as well as I had hoped because I was also enrolled in the internship of the degree, and other delicate personal problems, so the first few months of development seemed to show little progress.

PLANNING & EVALUATION

The meaning of this chapter is meant to deal with a more technical part of the work, showing the planning and evaluation of the project.

2.1 Planning

Detailed Time Planning:

- Conceptualization and Design:
  - Define the game concept, storyline, and gameplay mechanics.
  - Develop the game design document outlining all features and mechanics.

- Asset Creation:
  - Create concept art and sketches.
  - Model 3D assets such as characters, props, and environments in Blender.
  - Texture and UV map assets.
  - Rig and animate characters and objects.

- Programming:
  - Set up an Unreal Engine project.
  - Write code (blueprints) for gameplay mechanics, AI behavior, and user interfaces.
  - Debug and optimize code for performance.

- Testing and Polishing:
  - Playtest game mechanics and levels.
  - Implement visual effects and sound design.
  - Gather feedback and iterate on design based on testing results.
  - Fine-tune gameplay for balance and enjoyment.

- Documentation:
    - Create "marketing" materials.
    - Create the necessary documentation for the final submission

All these tasks shaped an action plan represented in Table 2.1

Task name	Hours	Status	January	February	March	April	May	June	JULY
Final degree project preparation	**20**	Done							
Technical proposal	10	Done							
GDD	10	Done							
Asset modeling & texturing	**70**	in pro...							
Concept Art	5	Done							
Main character	20	Done							
Enemies	10	Done							
Spaceship	15	in pro...							
Enviroment	20	in pro...							
Rigging & animation	**20**	Done							
Main character	15	Done							
Enemies	5	Done							
Unreal learning	**20**	Done							
Unreal implementation	**100**	To do							
Camera & player	20	Done							
Map	10	in pro...							
Enemies	20	Done							
Turrets	20	Done							
Upgrades	20	To do							
Hud	10	in pro...							
Testing	**20**	in pro...							
Documents	**50**	in pro...							
Memory	40	in pro...							
Presentation	10	To do							

Table 2.1: Outdated Gantt Chart

2.2 Resource Evaluation

As a solo developer, I will handle mostly all aspects of the game development process. This includes:

- Game Designer: Conceptualizing and designing the game (on this task my partner helped me).
- 3D Artist: Creating 3D models, textures, and animations in Blender.
- Programmer: Writing code and scripting in Unreal Engine.
- Quality Assurance: Testing and debugging the game for issues (In this task I asked Family and Friends to playtest).

Equipment/Resources:

- Computer Setup: Medium-performance PC, capable of running Unreal Engine and Blender smoothly and sufficient storage space for project files, assets, and backups.
- Software: Free Licenses for Unreal Engine and Blender, as well as any additional software tools for asset creation and development, like Krita, Gimp, or DaVinci Studio.
- Internet Connection: Access to online resources, courses, tutorials, and community support.

Cost Estimation:

- Software Licenses: All software I used is Free or OpenSource.
- Miscellaneous: All extra assets used are free to use, the hardware is around the 1000€ (*PcComponentes*, s. f.) would be amortized.
- Human Cost: The human cost of this game as it has been developed is not easy to calculate, since it was all done by one person, and no matter how much one can do, one person cannot do the same work as a team. Assuming that the game was developed in 300 hours (it was likely more), and that out of these, 100 hours were spent on modeling with an estimated monthly salary of around 1800€/month, about 11,25€/hour ("Glassdoor," s.f), 200 hours were spent on programming and testing with an average monthly salary of 1300€/month approx 8,125€/hour ("Glassdoor," s.f), and disregarding the hours for other tasks such as learning and document development which, for simplicity, would not be remunerated, the total human cost would be approximately 1125 + 1625 = 2750€ in total
- TOTAL COST: The cost of the game will round about the 4000€ summing all the costs

SYSTEM ANALYSIS AND DESIGN

This chapter presents the requirements analysis, design and architecture of the proposed work, as well as, where appropriate, its interface design.

3.1 Requirement Analysis

Every project originates from a problem that needs to be solved. In this case, the problem is to create the game, and for that, it is necessary to analyze how the game will be.

Astrokeeper is about a robot and it's ship, in a mission to clean the outer space, after a mistake calculating a route through an asteroid belt, they crash in a uninhabited planet similar to our earth, where a lost ancient civilization left their servant robots, which AI evolutioned to a point where they adapted their ald tasks to defend the planet. SPARKLE & ASTRA, have to escape this planet save & sound to continue their mission.

The game will have several scenes, including different menus and the playable level. In the initial menu, the player will find five buttons: "Play," which will take the player to the first level; "Options," which will open the options menu; "Controls and Rules," where the gameplay mechanics are explained; "Credits," an informational screen about the game; and "Exit," which will close the game.

In the options screen, there will be various sections where the player can adjust the game volume, language, screen mode (windowed or fullscreen), resolution, and the frame rate limit (FPS).

Once in the game, the player can move the character using WASD and jump with the space bar. The mouse will be used to move the camera, with the right button for aiming, and while holding it, the

left button for shooting. Additionally, the player can start the next wave with Z, interact with platforms and turrets with E, and speed up time with LCtrl.

The ESC and P buttons can open the pause menu, which has three buttons: one to continue the game, another to open the options menu, and a button to return to the main menu.

3.1.1 Functional Requirements

Functional requirements define a function of the system that is to be developed. From the previous analysis, they can be easily identified:

- R1: The player can start the game.
- R2: The player can change the game options.
- R3: The player can exit the game.
- R4: The player can move through the level.
- R5: The player can jump.
- R6: The player can aim.
- R7: The player can shoot (while aiming).
- R8: The player can start the next wave.
- R9: The player can spawn turrets.
- R10: The player can heal turrets.
- R11: The player can speed up the game time.
- R12: The player can pause the game.
- R13: The player can return to the main menu.
- R14: The player & turrets can hurt the enemies.
- R15: The turrets can be hurt by enemies

3.1.2 Non-Functional Requirements

Non-functional requirements impose restrictions on the design and implementation of the system, such as graphical aspects or quality standards. The following can be identified for this project:

- R14: The game can be played on PC.
- R15: The game mechanics will be easy to learn.
- R16: The aesthetic is stylized.
- R17: The UI will not be obstructive.
- R18: Enemy waves will scale in difficulty

3.2 System Design

In this section its presented the cases of use (Tables 3.2 to 3.12) and case of use diagram (Figure 3.1) taken from the functional requirements seen on the last section:

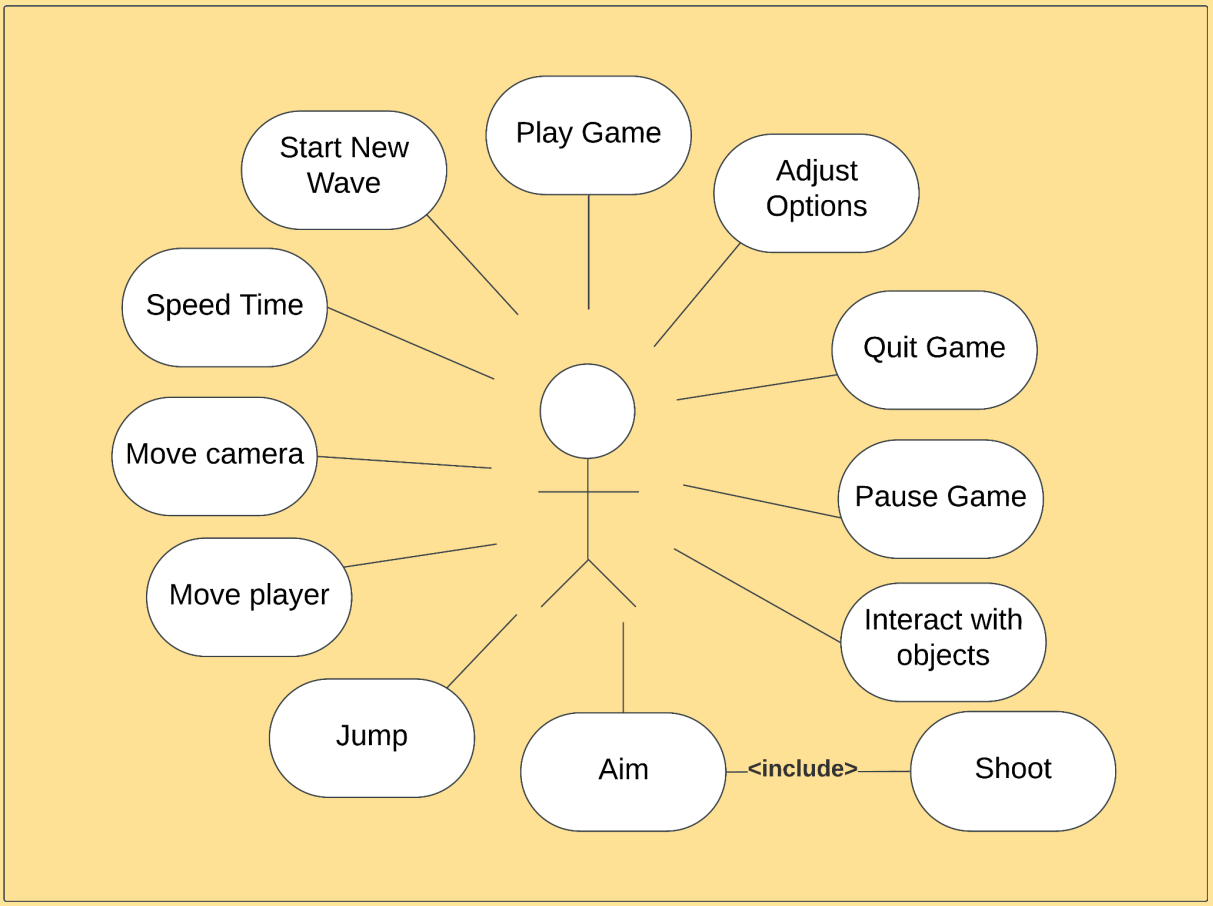

Figure 3.1. Case use diagram

Requirement:	R1
Actor:	Player
Description	The player starts the level by pressing the play button
Preconditions	1. The player must be in the main menu
Normal sequence	1. The player presses the play button 2. The game starts
Alternate Sequence	none

Table 3.2. Case of use <play game>

Requirement:	R2
Actor:	Player
Description	The player changes the options of the game
Preconditions	The player must be in the options menu
Normal sequence	The player presses any option The option changed is applied
Alternate Sequence	none

Table 3.3. Case of use <adjust options>

Requirement:	R3
Actor:	Player
Description	The player exits the game
Preconditions	The player must be in the main menu
Normal sequence	The player presses the exit button The game closes
Alternate Sequence	none

Table 3.4. Case of use <exit game>

Requirement:	R4
Actor:	Player
Description	The player move on the level
Preconditions	The player must be in game
Normal sequence	The player presses WASD The character moves
Alternate Sequence	none

Table 3.5. Case of use <move the player>

Requirement:	R5
Actor:	Player
Description	The player jumps
Preconditions	The player must be in game
Normal sequence	The player presses space The character jumps
Alternate Sequence	none

Table 3.6. Case of use <jump>

Requirement:	R6
Actor:	Player
Description	The player aims
Preconditions	The player must be in game
Normal sequence	The player presses RMouse The character aims
Alternate Sequence	none

Table 3.7. Case of use <Aim>

Requirement:	R7
Actor:	Player
Description	The player shoots
Preconditions	The player must be in game and aiming
Normal sequence	The player is aiming The player presses LMouse The character shoots
Alternate Sequence	The player presses LMouse The character do not shoot

Table 3.8. Case of use <shoot>

Requirement:	R8
Actor:	Player
Description	The player starts the next wave
Preconditions	The player must be in game and the previous wave should be finished
Normal sequence	The player presses Z The game start a new wave
Alternate Sequence	none

Table 3.9. Case of use <Start New Wave>

Requirement:	R9
Actor:	Player, object
Description	The player interacts with an object
Preconditions	The player must be in game and next to the object
Normal sequence	The player presses E The object does something
Alternate Sequence	none

Table 3.10. Case of use <interact with objects>

Requirement:	R10
Actor:	Player
Description	The player accelerates the time
Preconditions	The player must be in game
Normal sequence	The player presses LCtrl The time accelerates
Alternate Sequence	none

Table 3.11. Case of use <accelate time>

Requirement:	R11
Actor:	Player
Description	The player opens the pause menu
Preconditions	The player must be in game
Normal sequence	The player presses P or Esc The pause menu opens
Alternate Sequence	none

Table 3.12. Case of use <pause game>

3.3 System Architecture

The minimum requirements needed to play the engine are:

- Operating System Windows 10 version 1703, MacOS 13 Ventura or any reasonable new Linux distro from CentOS 7.x and up
- Graphic card any compatible with Direct X 11 or greater

These features are according to Unreal Documentation but the well function of this project is not guaranteed.

This project was tested and developed with the following features so any features equivalent or better than these will guarantee a smooth experience:

- Operating system: Windows 10 Home 22H2
- Processor: AMD Ryzen 5 3600 6-Core Processor 3.60GHz
- RAM Memory: 16GB
- Graphics: Nvidia Geforce GTX 750 Ti

Also the game is recommended to be played with mouse and keyboard but can also be played with a Xbox controller or similar.

3.4 Interface Design

The ingame interface (figure 3.13) design is unobtrusive and simple, giving the player the necessary information as the wave number, the enemies remaining or the ship life.

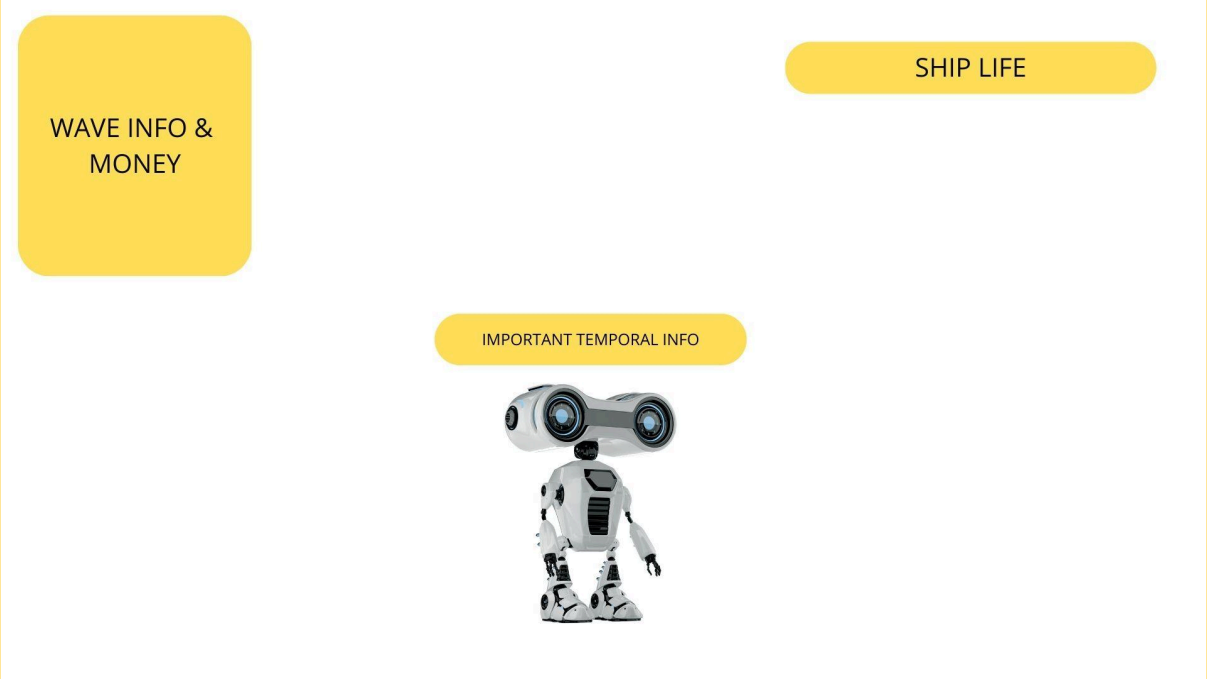

Figure 3.13. In Game Interface Design draft

The rest of the interfaces (Figures 3.14 - 3.16) are very simple and intuitive, letting the player play without complications.

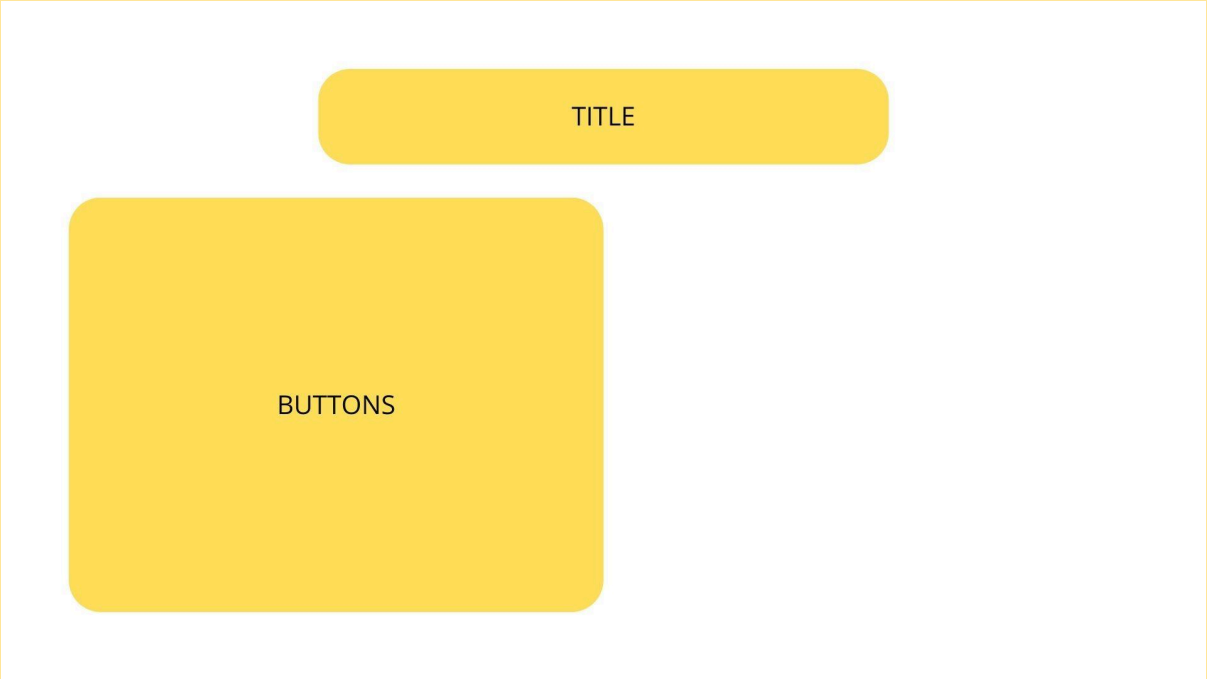

Figure 3.14. Main menu and options Interface Design draft

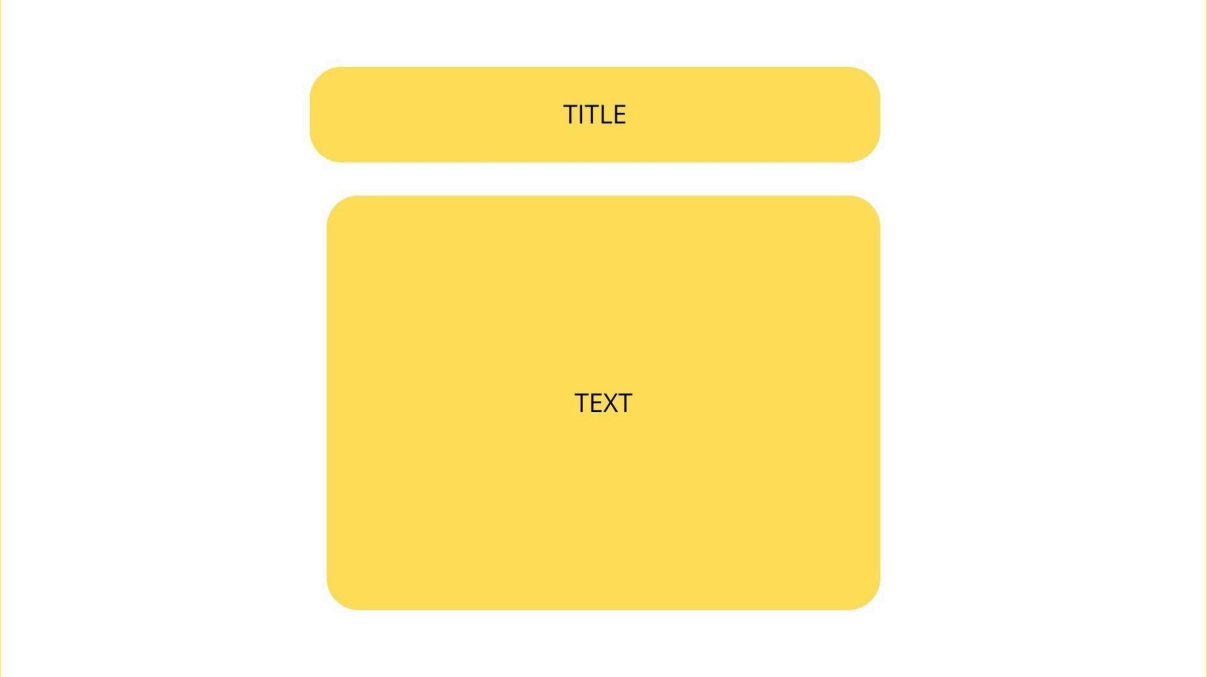

Figure 3.15. Credits and Controls Design draft

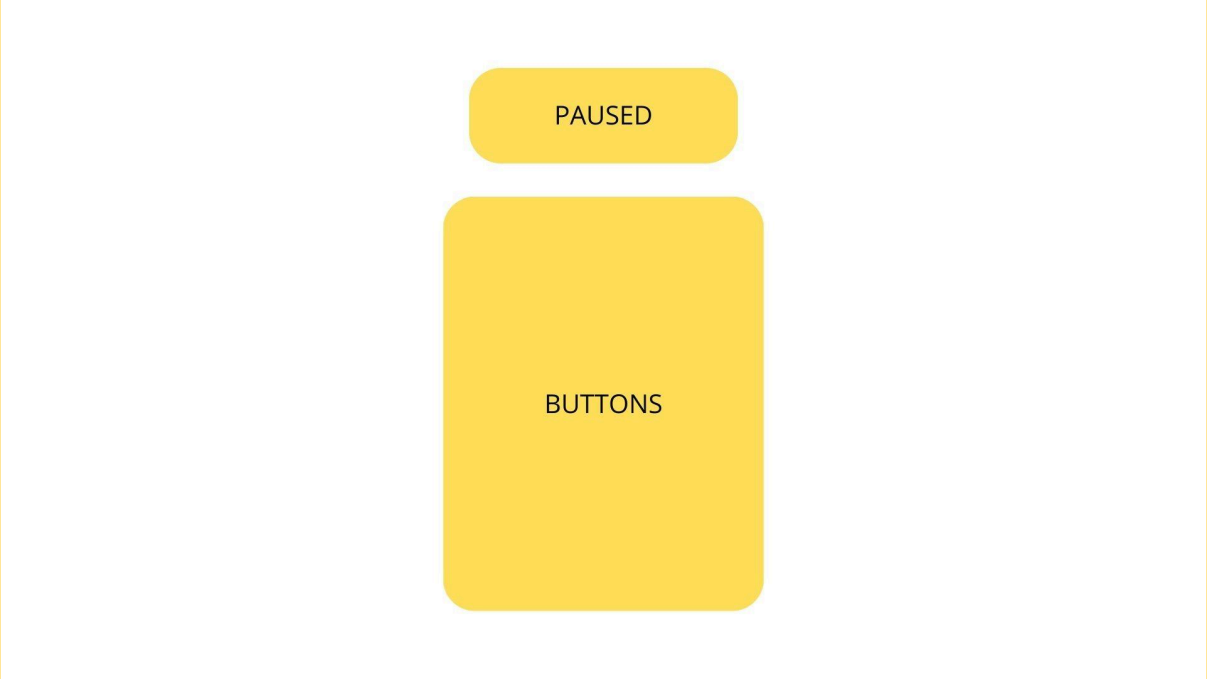

Figure 3.16. Pause menu Interface Design draft

WORK DEVELOPMENT AND RESULTS

Starting from a solid understanding of the tasks to be performed, the requirements of the video game, and the desired gameplay style, this section will explore the project's progress, the intermediate objectives achieved, and the modifications implemented during the process. Additionally, the results obtained will be evaluated, highlighting how some original ideas were adjusted and why.

4.1 Work Development

This section will be explained in the different areas of work: Modeling and Programming. This is the simplest way to explain the work done, as these tasks have been intertwined during development, and applying them in chronological order would be complicated. Also all these blueprints and functions mentioned can be seen on the Source Code point of this document.

MODELING:
The first task undertaken was modeling, specifically the modeling of the main character, SPARKLE, a cute space cleaning robot who, after some calculation errors and an asteroid accident, ends up crash-landing on an unknown planet where an ancient civilization left its servant robots in charge of the planet's defense.

There were three key aspects to its design: it had to be round and cuddly, low-poly, and easy to rig. It needed to be round and cuddly because who can resist the charm of such a character? It needed to be low-poly to keep the project lightweight, and easy to rig due to my practically nonexistent knowledge of this skill. For inspiration and references, I turned to Pinterest and created a

board with images that had the characteristics I was looking for. The result is shown in Figure 4.1, fulfilling the mentioned points to a greater or lesser extent, to achieve the desired roundness, more vertices were needed, although there are not too many, approximately 3000.

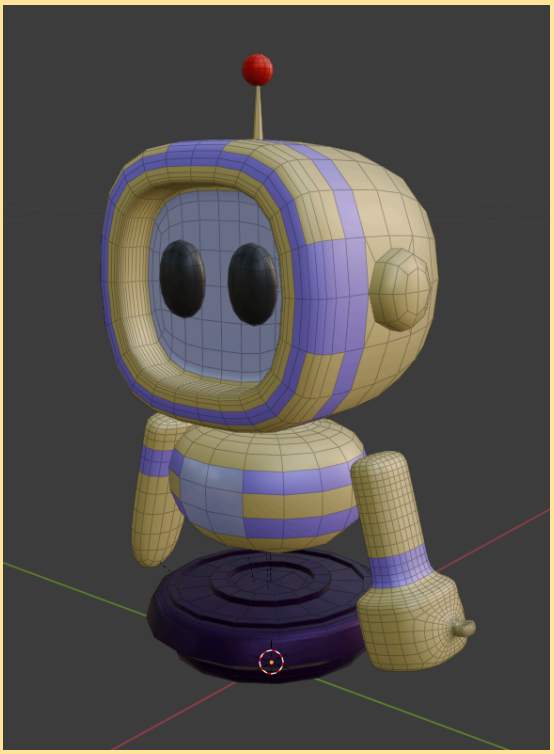

Figure 4.1. Sparke Model Wireframe & materials

After the main character, it was necessary to model the enemies. Following the same three pillars mentioned earlier and using the same method of searching for images on Pinterest, I developed five different models (Figure 4.2), each representing an enemy with a special ability.

The red one would be the common enemy, the yellow one a faster enemy, hence the thrusters on its sides, and the blue one, which was not implemented in the game, was intended to be the healer enemy, able to heal the other enemies. The purple one, the destroyer, deals a lot of damage but has little health, and the green one, the tank, would be a nod to a real tank with high health but slow movement.

Figure 4.2 The enemies models

After the enemies, it was time to model what is considered the foundation of any tower defense game: the turrets and the "tower" to defend. Here, I based the design on a simple concept that would align with the game's aesthetics and the main character. In this case, the turrets (Figure 4.3) consist of two parts: the platforms where they spawn and the turret bodies themselves, which are the shooting mechanisms.

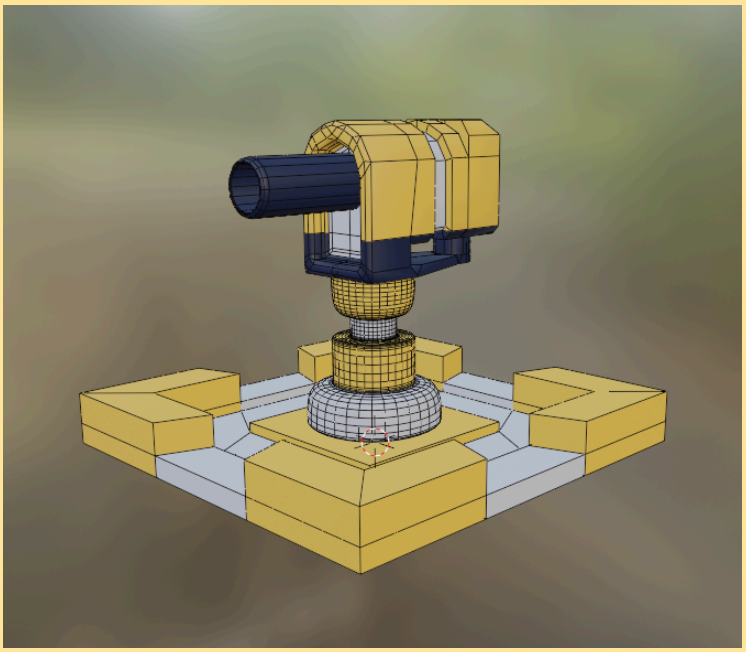

Figure 4.3. Turret model

The "Tower" that the player has to defend is our character's crashed spaceship, which is being "repaired" since it is controlled by an artificial intelligence. This spaceship underwent a couple of variations because the first version (Figure 4.4) didn't quite convince me. It had too many vertices and didn't fully match the game's aesthetics. Later on, I modeled another one that was more suitable, as shown in Figure 4.5.

Figure 4.4. The "bad" ship model

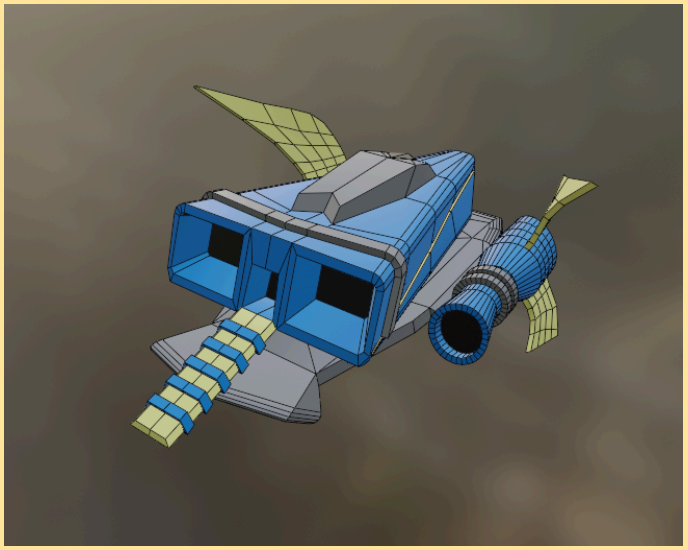

Figure 4.5 The new & final ship model

Other modeling work I did later on includes a simple projectile model without complications, while in a more advanced phase of the project, using Unreal's own Landscape tool, I created the level's environment (Figure 4.6) where I had a some problems as some of the textures used, as I was painting them on the landscape would simply disappear, after some research, it was caused by some issue with the texture buffer cache or something similar, a quick fix .I also modeled various foliage elements in Blender (Figures 4.7 and 4.8), the decorative elements, which I placed throughout the level using Unreal's own tool.

Figure 4.6 Level 1 Landscape

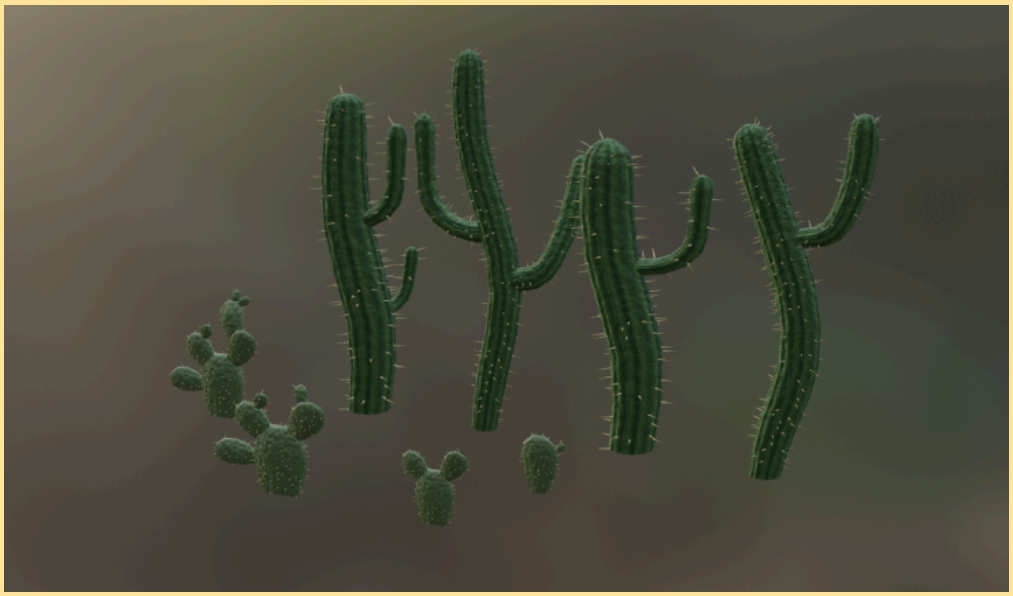

Figure 4.7 Cacti models

Figure 4.8 Rocks models

As a way to summarize and complete the section, I have carried out a study on all the models made and their statistics for the project and I have captured it in the following table (Table 4.9):

MODEL	VERTS	MATERIALS	In use
Sparke	3729	5	☑
Enemies	~ 2000 Each	4 each	☑
Turret	4010	3	☑
Ship	2027	4	☑
Bullet	500	2	☑
Small Rocks	40-70	1	☑
Large Rocks	1100-7700	1	☑
Small Cacti[1]	350 -1400	2	☑
Large Cacti	2000-3900	2	☑
Terrain	8836	5	☑
Spawer	360	1	☑
Clouds	100-300	1	☐

Table 4.9 Models Stats

[1] This number is this high due to all the spikes

PROGRAMING:

On the other hand, we must not forget the foundation of any game: programming. Since Unreal was chosen as the platform for this project, it allows for programming through Blueprints (Figure 4.10), a visual scripting tool based on object-oriented programming. This tool enables developers to create and manage game logic using a system of nodes and connections that represent events, functions, and variables. It simplifies the design and implementation of game behaviors and mechanics without the need to write traditional code.

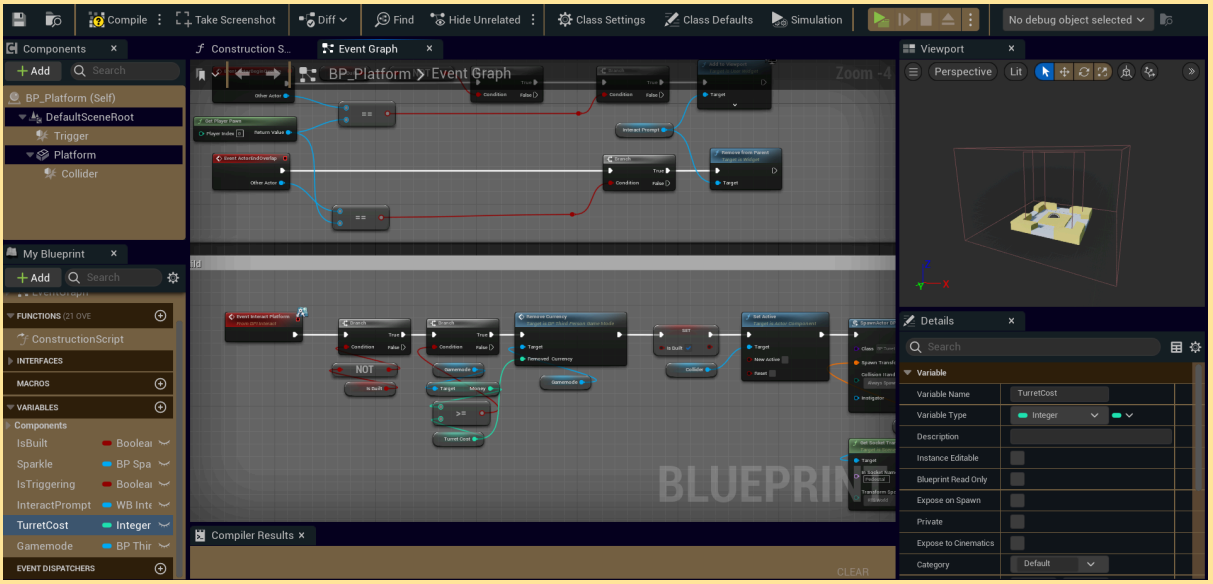

Figure 4.10 Blueprint example

This section will be explained according to the different blueprints, starting with the main character's blueprint:

CHARACTER:

The blueprint for SPARKLE inherits from the Character class, which is predefined in Unreal. At the start of this blueprint, the engine's input system is defined and assigned to this character, along with the camera limits. This camera has two positions: one when the character is not aiming and another when the character is aiming. These positions are similar but have some differences. When not aiming, the character will orient towards the movement direction, while when aiming, the character will be fixed towards where the camera is pointing, with a slight change in the field of view (FOV) and the character's speed.

Regarding Sparkle's movement, Unreal provides simple inherited functions to which only the input values need to be connected, including movement and jumping (which can only be performed when not aiming).

As for shooting, the character model has a socket, which is a point where something can be attached. This is where the projectiles are instantiated when shooting.
To interact with different objects in the game, they must implement an interface (a collection of functions without implementation that different classes can implement, allowing for common communication and functionality between unrelated objects in the game). Each time the interact button (E) is pressed, it will check each nearby object that implements this interface and, depending on the object, it will call the corresponding function in the blueprint of the object to be interacted with.

Since this blueprint essentially controls the game's inputs, it also contains the functions that manage the activation of the Widget (a type of blueprint for UI, to summarize) for the pause menu (Figure 4.11), the function to speed up time, and the function to start a new wave, which I will explain in more detail later.

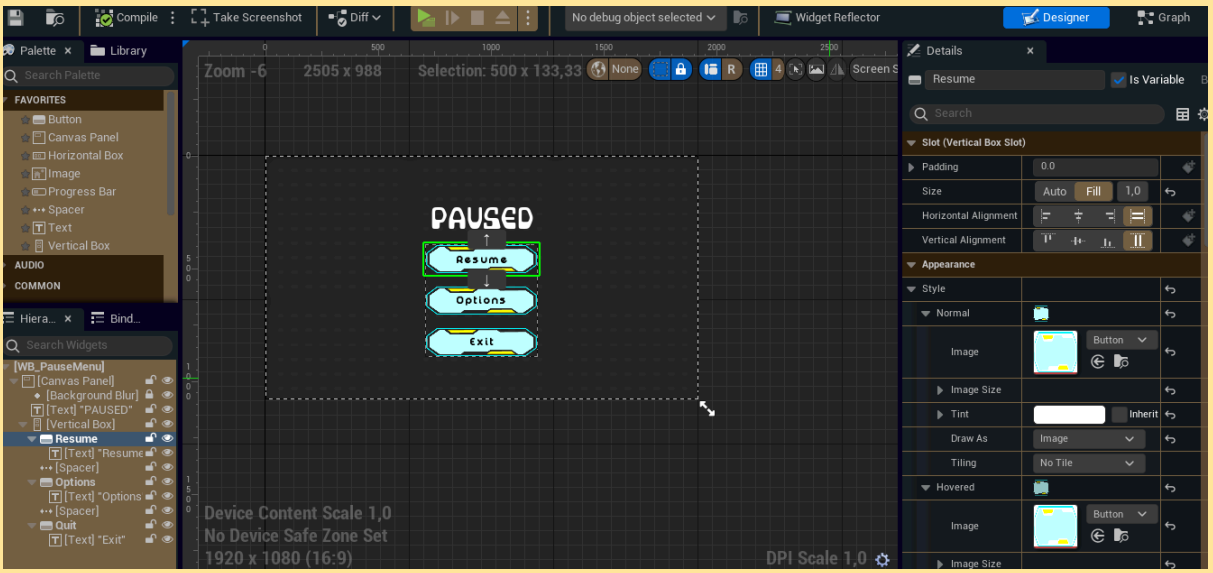

Figure 4.11 Pause menu Widget Blueprint

ENEMIES:
In this project, most of the programming efforts are dedicated to the enemies, their behavior, and how they appear. Taking advantage of object-oriented programming, all enemies originate

from the same blueprint, with each one then modifying variable values (shown in Table 4.12) and the 3D model accordingly.

Enemy	Health	Speed	Damage	Shoot speed
Standard	250	5	5	0.5s
Quick	100	13	3	0.1s
Destroyer	250	3.5	20	1.5s
Tank	500	2	10	1.5s

Table 4.12 Enemies Stats

Each time an enemy is spawned, several references to other actors in the scene are assigned to it, such as the game mode, the path it follows, the spaceship, and other variables are initialized. These include tags, its health (displayed as text in the game), and the initialization of its movement. Additionally, a call is set up for each game tick to the function responsible for detecting turrets and shooting at them.

The function that handles the enemy's movement (Figure 4.13) is simple: a path is established using a spline that the enemy follows from start to finish at a set speed in the direction of the path.

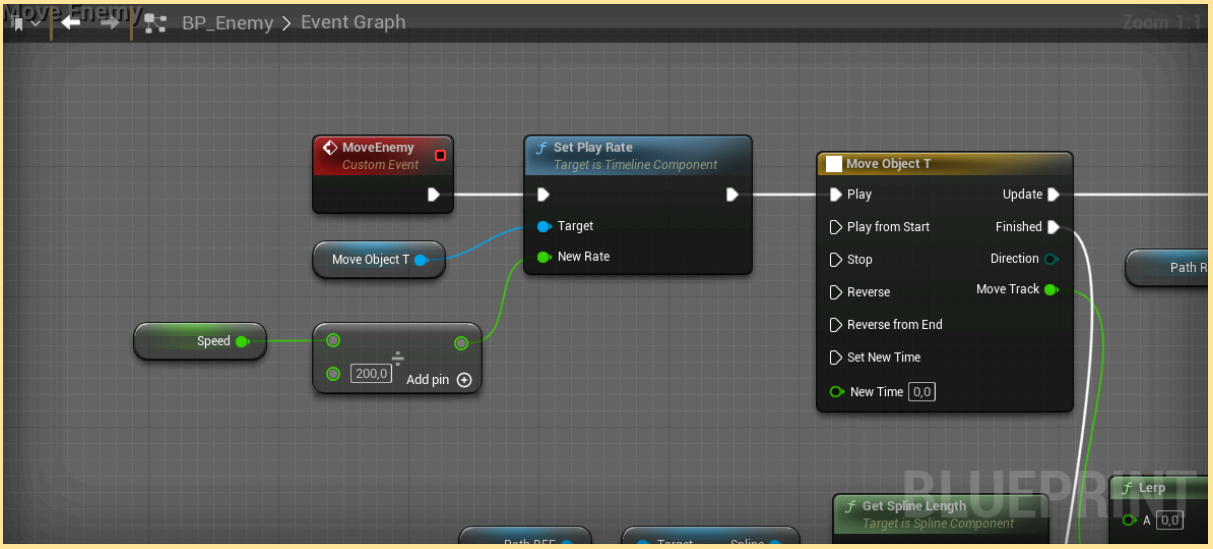

Figure 4.13 Part of the function in charge of the enemy movement

The complication arises when the enemy detects one or more turrets near the path.
It detects them through a system of triggers that check if what has entered its vision range are turrets, adding these to a list that contains all the turrets the enemy can shoot at. When these turrets leave the range, they are removed from the list.

This is where the shooting function comes into play. While there are no turrets in sight, meaning there are no turrets in the list of possible targets, the shooting function does nothing and allows the movement function to take full control of the enemy. The moment it detects one or more turrets, it calls a function that iterates through all the turrets in the list and returns the nearest turret. After this, it calls a function that handles the enemy's rotation (as shown in Figure 4.14). If this turret is "dead," it does nothing, but if it is alive, it sets the enemy's rotation to face the turret. Once rotated, and after a small delay, the enemy will start shooting, just like the main character, from a socket. Additionally, when the projectile is instantiated, it is added to a list so that when the enemy dies, any remaining projectiles in the game are removed.

Figure 4.14 Enemy and turret aiming at each other

In the enemy blueprint, the next function handles the damage dealt to it, utilizing Unreal Engine's built-in damage system, which offers convenient features. This system allows specifying the type of damage, the amount, the object causing it, and the actor responsible for it. Each time this function is called, it calculates the new health, updates the health bar displayed in the game, and then checks if the enemy is dead. If it is, it sets a boolean flag to indicate this and performs several tasks:

Firstly, it iterates through the previously mentioned list of projectiles and removes them from the game. Then, using a random number generator, it determines if the enemy will drop money for the player to build more turrets (Figure 4.15). There's a 60% chance to drop 50 coins, a 30% chance for 100 coins, and a 10% chance for 200 coins. Afterward, it generates a particle system for the explosion effect and subsequently destroys the actor.

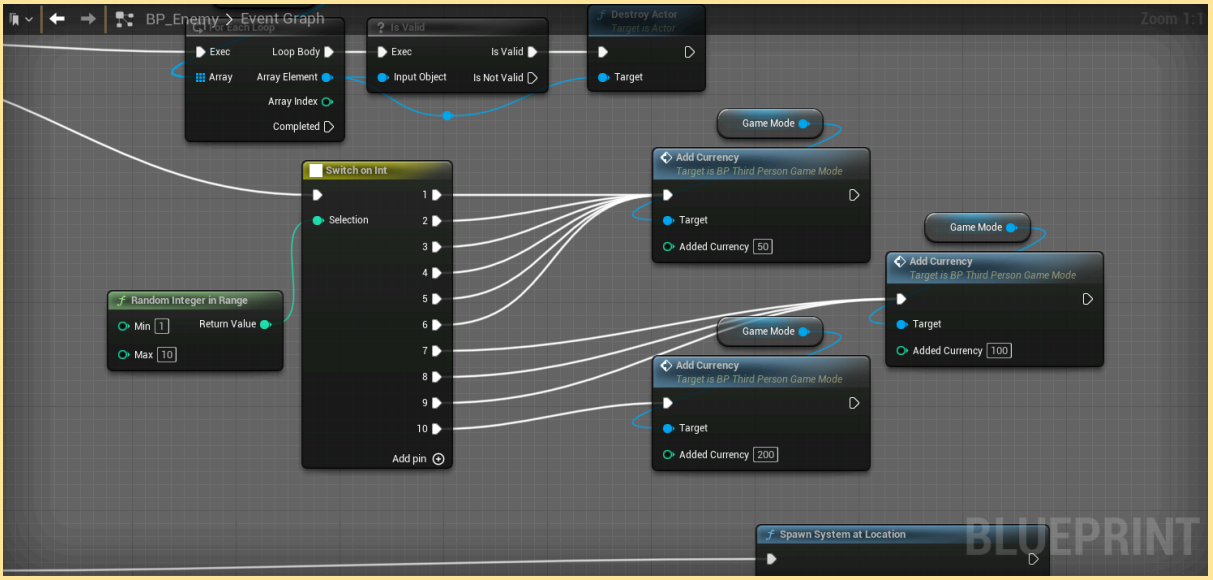

Figure 4.15 The random money system

This previously described function is called from within the same blueprint, and the enemy has another integrated trigger ('Hit'). Each time something passes through it, it checks the nature of this actor, and if it's a projectile not fired by itself, it applies damage because upon instantiation, the projectile is assigned damage and its instigator. Immediately after, this projectile is destroyed.

In this same blueprint, we also find a simple function that is called every game tick, which updates the rotation of the health bar of these enemies to ensure it is always visible to the player.

TURRETS:
The turrets operate in a similar manner to the enemies. The Start function initializes the actor's tags, displays the health text in the game, starts a tick timer that calls the function responsible for targeting enemies, and initializes the maximum health.

The enemy detection system works identically to its counterparts, with the strategy of shooting the closest enemy, using a trigger (Shown in Figure 4.16) and a list. Similarly, the function responsible for targeting enemies operates likewise with two slight variations. Firstly, since turrets cannot be destroyed but only weakened (if health reaches zero), there is a preliminary check to ensure this function does not execute if the turret is already weakened.

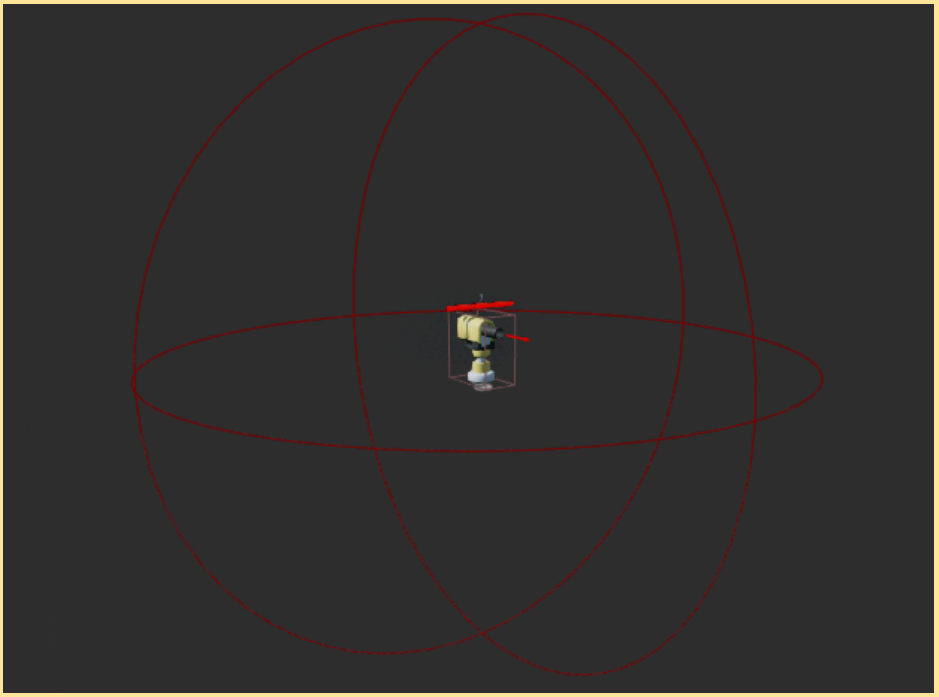

Figure 4.16 A turret and all its components

Similarly, the function that manages collisions checks that projectiles are not fired by itself and applies damage. In this case, before destroying the projectile actor, it also removes it from the list mentioned earlier for each enemy. The function that applies

damage to the turret follows a similar approach: it verifies that the projectile was fired by an enemy to avoid friendly fire. It calculates the new life of the turret, and based on its condition, it sets the boolean IsDead?

The only function that differs somewhat from the enemy is the healing function, which is called while the player is holding "E" near the turret. This function heals the turret by the specified amount with a delay determined by the player's blueprint.

PLATFORMS:
The other part of the turrets are the platforms, plates distributed throughout the map (pre-distributed at the start of the level) that act as turret placement points. Their structure and programming are straightforward, consisting of a couple of functions and a trigger. When the player comes into contact with this trigger, a message appears on the screen indicating that they can place a turret and showing its cost (which disappears when they exit the trigger). While inside the trigger, if the player presses the interact button ("E") and has enough money, a turret will be constructed on the socket assigned in the platform's model.

PROJECTILE:

Figure: 4.17 "Friend" Bullet

There's not much to say about the projectile blueprint (Figure 4.17) It simply changes color depending on who fires it (yellow for friendly fire, pink for enemy fire). Additionally, to prevent actor overload in the world, there's a function that deletes the instantiated projectile upon collision with anything other than an enemy or turret (considering the list of enemy projectiles).

SHIP:
Another straightforward blueprint to explain is the ship. At the start of the level, it initializes the health widget (and controls it), and then this blueprint also includes a function for applying damage inflicted upon the ship. When the ship's health reaches zero, it calls the 'end level' function, responsible for ending the level. How does the ship take damage? Enemies, upon completing their path along the previously mentioned spline, inflict proportional damage to the ship based on the remaining enemy health and the damage they cause, according to the following formula (Figure 4.18):

$$\left(10 \times \frac{Actual\ Enemy\ Health}{Max\ Enemy\ Health}\right) \times Enemy\ Damage$$

Figure 4.18 Ship damage formula

WAVE SYSTEM AND SPAWNER:
Once all the actors in the equation have been explained, it's time to look into the core mechanics of the game: waves and enemy spawning.

These are primarily implemented in the game mode blueprint, which runs at the level scope rather than being tied to a specific actor, although levels themselves also have their own blueprints. Functions in this blueprint include for example adding or subtracting coins from the player, which are referenced by the respective blueprints that use them.

The game mode starts by scanning the level for all actors with the spawner blueprint and adds them to a list (useful for using multiple spawners per level). Next, it initializes the HUD and sets the number of waves depending on the level. Then, it waits for player input. When it receives input, it triggers the EnterTransition function, which in turn calls the function that starts everything once the specified transition time has elapsed.

NewWave is responsible for incrementing the current wave number, calling StartWave, and updating the widget with the current wave number and the number of remaining enemies. StartWave prepares the list of enemies that should appear during the wave

and passes it to the spawner. Depending on the level, there's a data table with this information. Each row in this table represents a wave and lists references to enemy types and their quantities. This step also allows for the possibility of generating waves algorithmically.

These data are referenced in a dictionary-like structure, shown in Figure 4.19, where the key is the enemy type and the value is the number of enemies of that type to spawn. With this structure, it's straightforward to calculate the total number of enemies in the wave by iterating through each key and summing all the values. This total is then used to iterate through the same structure randomly, adding these enemies to a queue in the spawner blueprint.

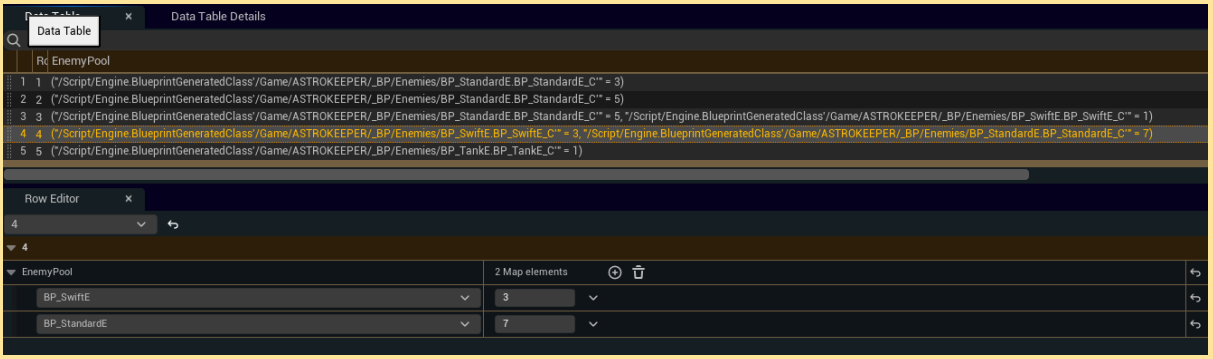

Figure 4.19 Data Table with the enemies in each wave

To clarify, these enemies are added to a queue using the AddEnemyToQueue function in the spawner blueprint (Actor shown in Figure 4.20). If there are no enemies nearby, the spawner attempts to spawn an enemy by calling SpawnEnemy. If SpawnEnemy isn't called by AddEnemyToQueue, a timer set during the spawner blueprint initialization triggers SpawnEnemy every 2 seconds.

The role of SpawnEnemy is to take the first enemy in the queue and spawn it. Additionally, it sets up an event binded to the death of this enemy.

Figure 4.20 Spawner Actor on the level

The event mentioned, OnEnemyDeafeated, is triggered when an enemy dies, whether by the player, a turret, or reaching the end of its path. Its purpose is to decrement the remaining enemy count and check the EnemyMaxCount variable.

The EnemyMaxCount variable ensures that there's a maximum number of enemies always present in the game, particularly useful for large waves and not having lots of enemies to deal with. If there are still enemies left to spawn, they continue appearing.

Finally, if no enemies remain to be spawned and depending on whether it's the last wave of the level or not, it either ends the level or prepares for the player to start the next wave.

OTHER:

During the project, many other less significant tasks were also carried out, such as searching for sound materials and free-to-use sources on websites like Dafont or Pixabay, several redesigns of the visual appearance of the UI (Figures 4.21-22), and work within the engine to create particle systems and visual effects.

Figure 4.21 Old Button design

Figure 4.22 New Button design

This last image is part of almost all of the widgets referring to menus, like the Main menu (in which the background is also made by me in blender) or the option menu which lets the player change some of the game options like the screen resolution or the game volume.

4.2 Results

After completing the project and reflecting on the established objectives, I believe that while I didn't achieve all goals 100%, particularly in not developing more than one level as planned, I do feel that the game is highly playable, and I am satisfied with the functionality of the developed mechanics. Some of this results can be seen on figures 4.23-25 shown below:

Figure 4.23 Main menu

Figure 4.24 A wave starting

Figure 4.25 Sparkle shooting an enemy

As one of the objectives stated, I aimed for the game to be fully playable and now it's available for download at[2]:

- Google Drive: ASTROKEEPER_DEMOS (Windows only)
- Itch.io: https://pensadox11.itch.io/astrokeeper (Windows & Linux)

Additionally, there is a complete project download available on Google Drive with the different versions of the project (too large for GitHub): Projects Folder

A playtesting campaign is also being carried out, leaving a list of bugs and features to improve for the future. The link to the form is https://forms.gle/bVjhw1kv1gPSAMGQ6 or it can be found also in the Itch.io page linked before.

[2]Mac builds are exclusive for Mac users, since I am not one it is impossible for me to generate one

CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

During this past year, I have finally discovered what I want to dedicate my professional career to, or at least give it a try. This project has been the final step that made it clear to me.

In this industry, teamwork is usually the norm, but having managed to develop a video game almost from scratch (whether more professional or less) has helped me overcome the impostor syndrome that we all struggle with. It has taught me that this feeling can be mitigated through hard work, practice, and effort. No one enters the professional world knowing everything or knowing exactly what to do at all times

5.2 Future work

This project was rescued from a course where I learned a lot, and I held some affection for it because it was where I realized I could program, despite entering the degree without even knowing what a for loop was.

The success of the game is partly due to the enthusiasm I had for the project since its conception in that course, so now I am determined not to let it fall into oblivion.

There are many things left to do, things that I hadn't initially considered, such as an initial cinematic or intermediate cutscenes, adding an ending, and giving meaning to the story. Regarding mechanics, I aim to fully achieve the initially set objectives: creating 4 or 5 levels with different environments to enhance my modeling skills, adding other types of turrets that cause different types of damage, implementing a character and turret upgrade system, and even including the healer enemy that was left out along the way. Most of these "to do" improvements have been mentioned in the responses of the playtesting survey among others that have already been solved.

Someday, this project is going to be uploaded on steam, that's the goal.

BIBLIOGRAPHY

A.1 Bibliography

- Hardware and Software Specifications. (s/f). Epicgames.com. Retrieved 29/07/2024 https://dev.epicgames.com/documentation/en-us/unreal-engine/hardware-and-software-specifications-for-unreal-engine#requirementsforue5renderingfeatures

- AstroKeeper Board. Pinterest. (s/f). Pinterest.com. Retrieved 29/07/2024 https://www.pinterest.es/juanmapensado/astrokeeper/

- DaFont - Descargar fuentes. (s/f). Dafont.com. Retrieved 29/07/2024 https://www.dafont.com/es/

- Música y efectos de sonido. (s/f). Pixabay.com. Retrieved 29/07/2024 https://pixabay.com/es/sound-effects/

- PcComponentes. (s. f.). PcComponentes.com. Retrieved 29/07/2024 https://www.pccomponentes.com/pccom-ready-amd-ryzen-5-5600x-16gb-1tb-ssd-rtx-4060

- Glassdoor. (n.d.-a). Sueldo: Diseñador 3D. Retrieved 29/07/2024 https://www.glassdoor.es/Sueldos/diseñador-3d-sueldo-SRCH

_KO0,12.htm#:~:text=El%20sueldo%20medio%20para%20el,929%20€%20y%201020%20€.

- Glassdoor. (n.d.-b). Sueldo: Desarrollador de videojuegos. Retrieved 29/07/2024 https://www.glassdoor.es/Sueldos/desarrollador-de-videojuegos-sueldo-SRCH_KO0,28.htm

A.2 List of Figures

SOURCE CODE

This project is located in ready to download a drive folder (see 4.2 Results) and needs the Unreal 5.3.2 Editor version to open it.

On the other hand HERE I linked a google drive folder with images and folders to all the blueprints mentioned (some blueprints are more than one image) due to the limitation to zoom on the photo in a pdf.