UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

**FACULTATEA DE INFORMATICĂ**

LUCRARE DE LICENȚĂ

# Joc bidimensional Tower Defense în Unity

propusă de

**Bogdan-Dumitru Costea**

**Sesiunea:** iulie, 2024

Coordonator științific

**Lect. Dr. Alex Mihai Moruz**

UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

**FACULTATEA DE INFORMATICĂ**

# Joc bidimensional Tower Defense în Unity

**Bogdan-Dumitru Costea**

**Sesiunea:** iulie, 2024

Coordonator științific

**Lect. Dr. Alex Mihai Moruz**

Avizat,

Îndrumător lucrare de licență,

Lect. Dr. Alex Mihai Moruz.

Data: ..... Semnătura: .....

## Declarație privind originalitatea conținutului lucrării de licență

Subsemnatul **Costea Bogdan-Dumitru** domiciliat în **România, jud. Prahova, mun. Ploiești, strada Cameliei, nr. 10, bl. 38, et. 2, ap. 23**, născut la data de **10 octombrie 2001**, identificat prin CNP **5011014460031**, absolvent al Facultății de informatică, **Facultatea de informatică** specializarea **informatică**, promoția 2024, declar pe propria răspundere cunoscând consecințele falsului în declarații în sensul art. 326 din Noul Cod Penal și dispozițiile Legii Educației Naționale nr. 1/2011 art. 143 al. 4 și 5 referitoare la plagiat, că lucrarea de licență cu titlul **Joc bidimensional Tower Defense în Unity** elaborată sub îndrumarea domnului **Lect. Dr. Alex Mihai Moruz**, pe care urmează să o susțin în fața comisiei este originală, îmi aparține și îmi asum conținutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licență să fie verificată prin orice modalitate legală pentru confirmarea originalității, consimțind inclusiv la introducerea conținutului ei într-o bază de date în acest scop.

Am luat la cunoștință despre faptul că este interzisă comercializarea de lucrări științifice în vederea facilitării falsificării de către cumpărător a calității de autor al unei lucrări de licență, de diplomă sau de disertație și în acest sens, declar pe proprie răspundere că lucrarea de față nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Data: ..... Semnătura: .....

## Declarație de consimțământ

Prin prezenta declar că sunt de acord ca lucrarea de licență cu titlul **Joc bidimensional Tower Defense în Unity**, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test, etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de informatică.

De asemenea, sunt de acord ca Facultatea de informatică de la Universitatea "Alexandru-Ioan Cuza" din Iași, să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Absolvent **Bogdan-Dumitru Costea**

Data: .....  Semnătura: .....

# Contents

# Motivation

Since when I was a kid, a big part of my entertainment each day was through computer games. They represent, and still do, a method to sink into any type of world you desire: fantastical or realistic, relaxing or intense, funny or serious, the possibilities in this domain being endless, and the recent growth of the popularity of video games have solidified them in the entertainment industry, becoming a worldwide phenmomenon appreciated by more and more people.

For me, the purpose of this project is to solidify my capabilities in the IT domain and to understand the basics of constructing a video game in one of the most popular game engines, called Unity, in order to solidify my future in my carreer and start creating games, either as a part-time hobby besides my work in IT, either full-time if I will have success and luck in the domain of creating games.

# Introduction

In this Bachelor's Thesis I will present the game on which i worked on the duration of working on the thesis, called "STENIAD", a 2D Tower Defense with a special feature, being the control of the game using a terminal, creating a strong bond between code and game.

I will go through all the elements present in the project, from the graphical aspects specific to the Game Engine which I worked on, which is Unity, and how it's intertwined with the "spine" of the code, which is the code behind the game which ensures correct processing and running of the game in a stable and correct way.

This Bachelor's Thesis will be split into four chapters:

- Basic elements, where I will describe how I structured the Unity specific graphical elements and how I created the Intepretor, the main controller of the game;
- Level format, form loading it in the game space to the basic structural elements, both visible and invisible which, added together, create the layout of a functional level;
- Entities, more precisely of turrets and enemies, how and where they are created, and the interaction between them and the terminal;
- Game demo and additional aspects, how the game is being played and how it interacts with the player, and other elements added in order to improve the look of the game.

# Chapter 1

# Basic elements

In the game engine Unity, all the elements are created through a multitude of Game Objects, these having the posibility of being modified in order to create anything we wish, such as text, images, characters, etc., which can be manipulated through scripts.

In this chapter I will present how I created all these objects which will be or will contain other elements such as levels and entities in the future and how they are placed on the game screen, and how I created the Interpretor: how they are created, what graphical elements I used and the scripts which assures optimal functionality.

## 1.1 Placement

Object and the graphical interface made of game object are present in the figures 1.1, 1.2 and 1.3, which are organised as follows:

**Terminal:**

Represents the main object, or the parent object for every other game object, which holds the scripts of the Interpreter which we will talk about in section 1.2.

**Screen:**

Represents a Canvas-type object, meaning the interface where all the UI elements must be included in, from where all the other object are created as children of this object.

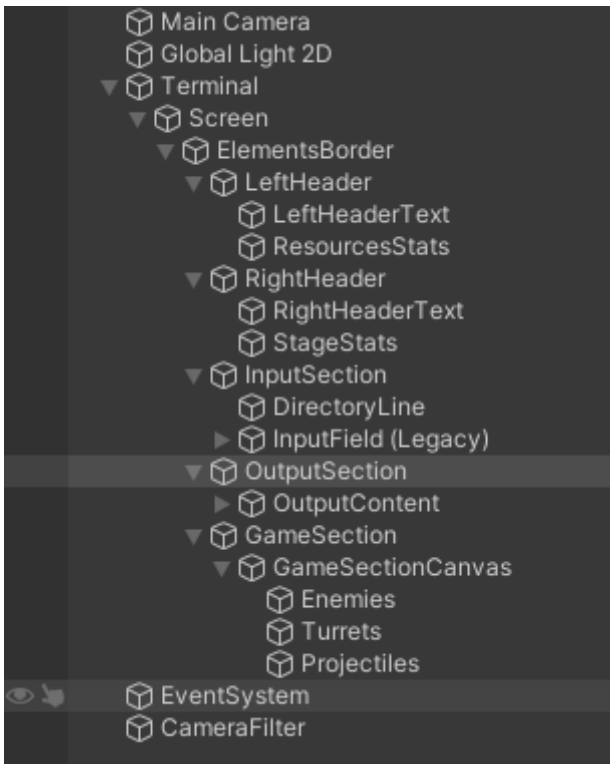


Figure 1.1: Project Hierarchy

**ElementsBorder:**

Represents an optional empty object which has the role of reducing the total space of game elements on the screen, in order to keep enough space between the elements and the screen border. At the figure above which shows the Scene View you can see the border, being the inner rectangle from figure 1.2.

**LeftHeader:**

Represents the object which contains two text elements from the upper right side, these being children of the object. The above text is just a static title of the terminal number, being visual only, and the text below being a child object which represents the remaining resources of the player, which can be called and modified by other scripts based on how the game is progressing.

**RightHeader:**

Just as in ”LeftHeader”, it represents an object which holds two text elements, this time in upper left, where both of these will be modified in order to contain details about the game, being the current health of the home base and how many enemies are

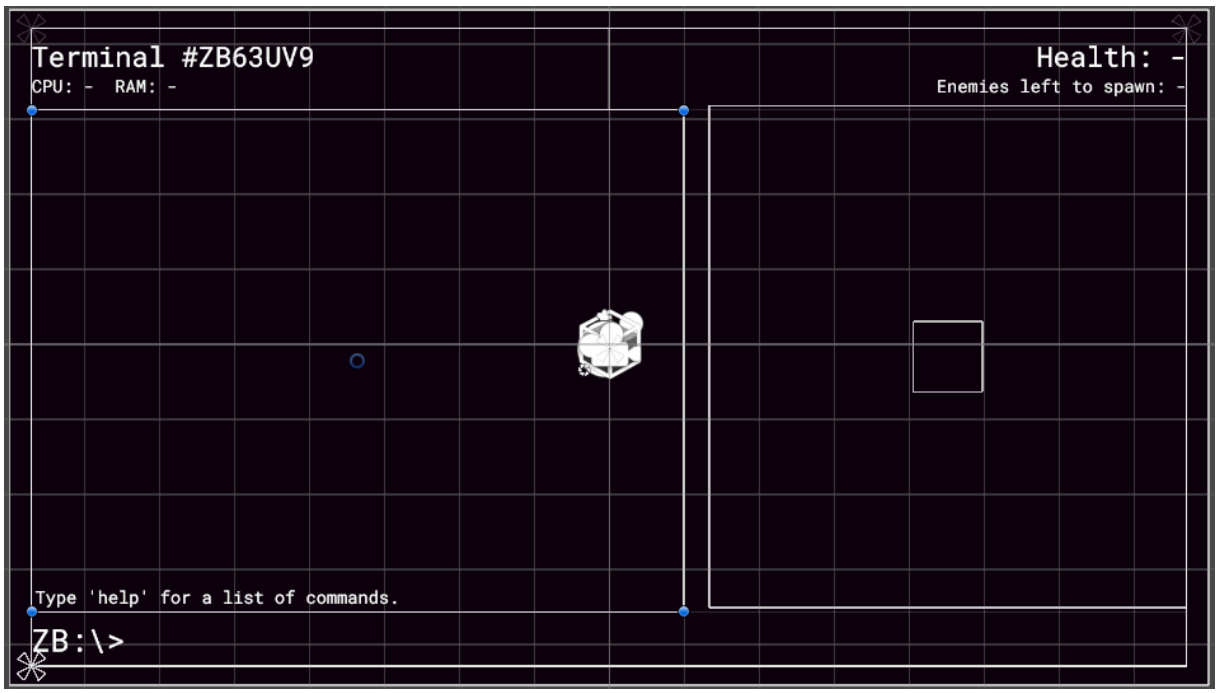


Figure 1.2: Game interface

left to spawn in the current wave, both being able to be called and modified by other scripts based on the game progression.

**InputSection:**

At the bottom of the gave we have the input section, which is an object which contains the module ”InputField”, which is similar to an input like every terminal. It can be called by the Interpretor, the text being extracted from this element on pressing ENTER.

**OutputSection:**

In the left center of the screen we have the objects which will contain answers given by the Interpretor, these being under the form of predefined objects, or prefabs, in which there are included text modules for modifying and displaying on the screen. Also it uses a ”ScrollRect” and a ”RectMask” which add the posibility of scrolling through the output lines and hiding any lines outside the object respectively.

**GameSection:**

Finally, in the right middle section we have the object which will include the game itself. It contains only three objects: enemies, turrets and projectiles, which have

organizational roles for the predefined objects which will be added on the course of playing the game.

## 1.2 Interpretor

The Interpretor represents the system which links graphical elements with logical ones and the player with the game, processing every command received and sent towards the necessary scripts. Knowing these things, it is understandable why it was the first thing to be implemented.

For the functionality of the Interpreter there are two scripts implemented, **CommandManager.cs** and **Interpreter.cs**, which communicate with each other constantly, processing received data by the user and sending back an appropriate response.

**CommandManager:**

CommandManager.cs has the role of collecting the user inputs, transmiting them towards the Interpreter, displaying responses and emulating the behaviour of a normal terminal.

The first part of the script is represented in figure 1.3. The **Start()** function is simple, making the link between the Interpretor and focuses the input field, and the **OnGUI()** function collects the input once the player presses ENTER and has wrote something in the input field, storing the received comand and showing it on the screen, resetting the input field text, sending the user input to the Interpretor for processing.

In the second part, which is displayed in figure 1.4, the Interpreter response is displayed using the **AddInterpreterLines()** and then scrolls the text from the terminal output, and the function **AddLine()** shows only a single line on the screen with text declared in the parameter of the function, and is used to display messages from events present in other scripts besides the Interpreter, for example once a wave is won or lost.

The function from part two of the script are using a prefab called ”OutputLine”, where it is duplicated, its content modified, and is added, optionally in order or line receival from the Interpretor in case of using **AddInterpreterLines()**.

```
[Header("References")]
[SerializeField] public GameObject responseLine;
[SerializeField] public InputField input;
[SerializeField] public ScrollRect scrollRect;
[SerializeField] public GameObject msgList;
[SerializeField] public Interpreter interpreter;

private void Start()
{
    input.ActivateInputField();
    input.Select();

    interpreter = GetComponent<Interpreter>();
}

// GUI Events
private void OnGUI()
{
    if (input.isFocused && input.text != "" && Input.GetKeyDown(KeyCode.Return))
    {
        string userInput = input.text;
        AddLine("> " + userInput);
        input.text = "";

        int lines = AddInterpreterLines(interpreter.Interpret(userInput));

        // Scroll to bottom
        scrollRect.verticalNormalizedPosition = 0;

        input.ActivateInputField();
        input.Select();
    }
}
```

Figure 1.3: CommandManager.cs part 1

```
public void AddLine(string userInput)
{
    Vector2 msgListSize = msgList.GetComponent<RectTransform>().sizeDelta;
    msgList.GetComponent<RectTransform>().sizeDelta = new Vector2(msgListSize.x, msgListSize.y + 19.0f);

    GameObject msg = Instantiate(responseLine, msgList.transform);
    msg.transform.SetSiblingIndex(msgList.transform.childCount - 1);

    msg.GetComponentsInChildren<Text>()[0].text = userInput;
    scrollRect.verticalNormalizedPosition = 0;
}

public int AddInterpreterLines(List<string> interpretation)
{
    for (int i = 0; i < interpretation.Count; i++)
    {
        GameObject response = Instantiate(responseLine, msgList.transform);

        response.transform.SetAsLastSibling();

        Vector2 listSize = msgList.GetComponent<RectTransform>().sizeDelta;
        msgList.GetComponent<RectTransform>().sizeDelta = new Vector2(listSize.x, listSize.y + 19.0f);

        response.GetComponentInChildren<Text>().text = interpretation[i];
    }

    return interpretation.Count;
}
```

Figure 1.4: CommandManager.cs part 2

**Interpretor:**

Interpretor.cs represents the script which creates links between every other script which has the role of game functionality, having more logic implemented in order to ensure that there is no edge case where an input received by the player isn't processed right and any variables collected from the input are not mispelled or inexistant, creating a stable and error-less functionality of the game.

In figure 1.5 and 1.6 we have an example of the main processing function of the input logic. The code, processed by the **Interpret()** function, is divided in more processing section based on the first word, and testing if there are any additional options stated, each one showing different outputs.

This example represents a small part of the expansiveness of the code, how it's expected from a script which takes account off all command combinations. So, in the following chapter I will describe how the Interpretor tests, processes and and creates a link with other systems of the game. Currently the given example creates a concrete idea of how the script functions.

In figure 1.7 two functions are presented which help format text. Unity text supports Rich Text, having the possibility of playing with formatting. In the script I added the **ColorString()** function which simplifies coloring text through a dictionary named colors containing the hex codes of colors, and the **LoadTitle()** function used for loading ASCII text in special situations, with a ASCII text example in figure 1.8.

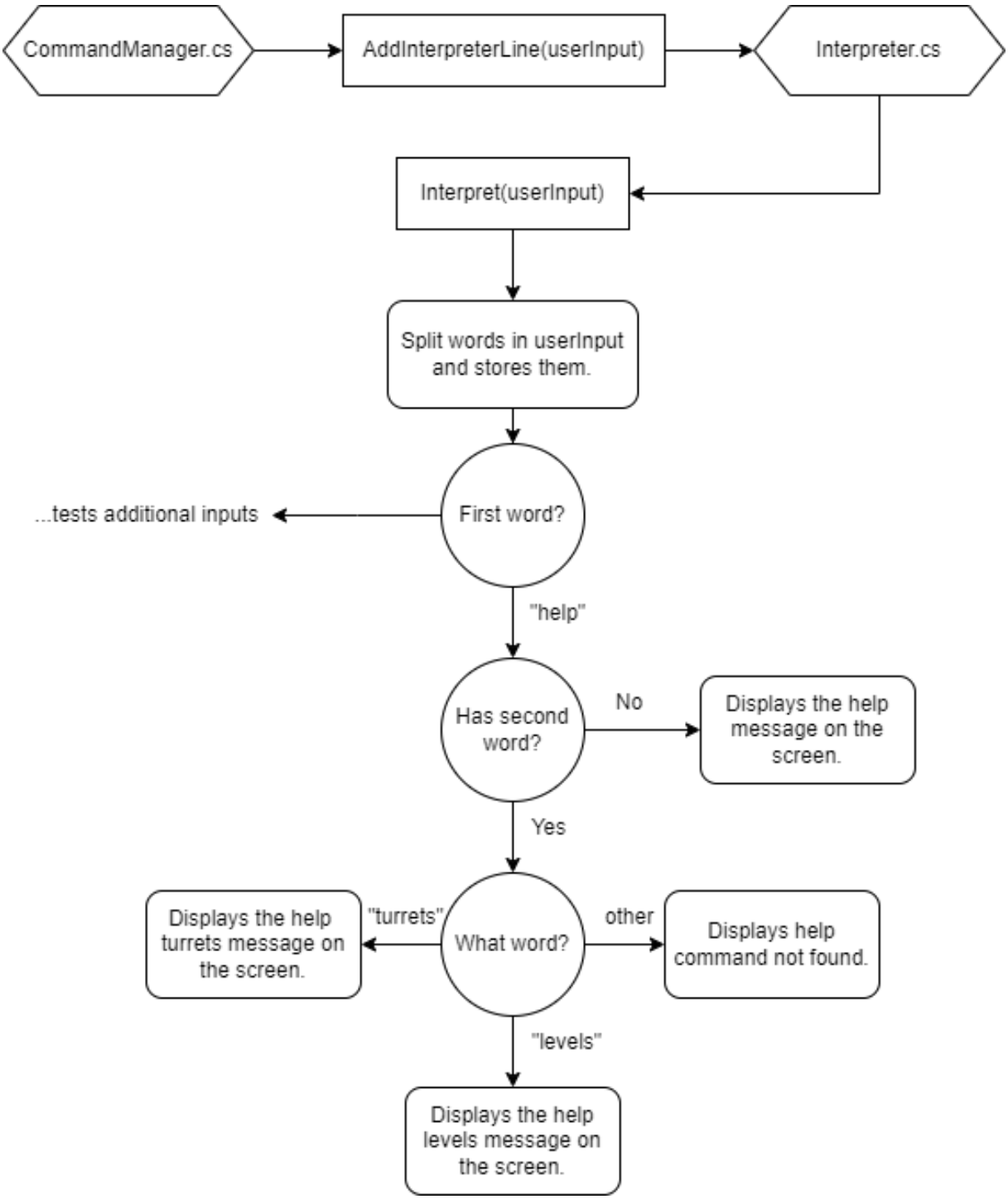


Figure 1.5: Example Interpret() ”help” diagram


Figure 1.6: ”help” command in game

```
private string ColorString(string s, string color)
{
    string leftTag = "<color=" + color + ">";
    string rightTag = "</color>";

    return leftTag + s + rightTag;
}

private void LoadTitle(string path, string color, int spacing)
{
    StreamReader file = new StreamReader(Path.Combine(Application.streamingAssetsPath, path));

    for (int i = 0; i < spacing; i++)
    {
        response.Add("");
    }

    while(!file.EndOfStream)
    {
        response.Add(ColorString(file.ReadLine(), colors[color]));
    }

    file.Close();
}
```

Figure 1.7: Interpretor utilitary functions


Figure 1.8: Example ASCII

# Chapter 2

# Level format

Levels represent the frame where the action happens, having the greatest influence over the interaction between the player and the game, which plays a great part in design decisions of a developer.

In this chapter i will describe all the tehnical elements which build a level, from the visible to the invisible elements.

## 2.1 Plots

Being the elements which is the most interacted with by the player, plots represent the objects where defensive turrets will be placed, having a strategic role depending on positioning and angles between turrets and enemies in order to maximize efficiency and covering the path that enemies will take.

This version of the game features three types of plots: normal plots, which allow placing a turret on them; supercharged plots, which in addition to normal plots amplifies the power of turrets placed om them, and the blocked plots, which do not allow construction of turrets on them, often placed in points that would make the game too easy.

In addition to the ones listed above, there are two blocked plots with special textures, being the enemy and home base, which define where the enemies will come from and where the enemies will go respectively.

Plots are placed in rows and collumns, their numbering starting from 0, with every plot having a different texture which defines their type.

In figure 2.1 we have the first level of the game, before creating turrets or enemies.


Figure 2.1: Example level 1 Plots


Figure 2.2: Plots prefab list

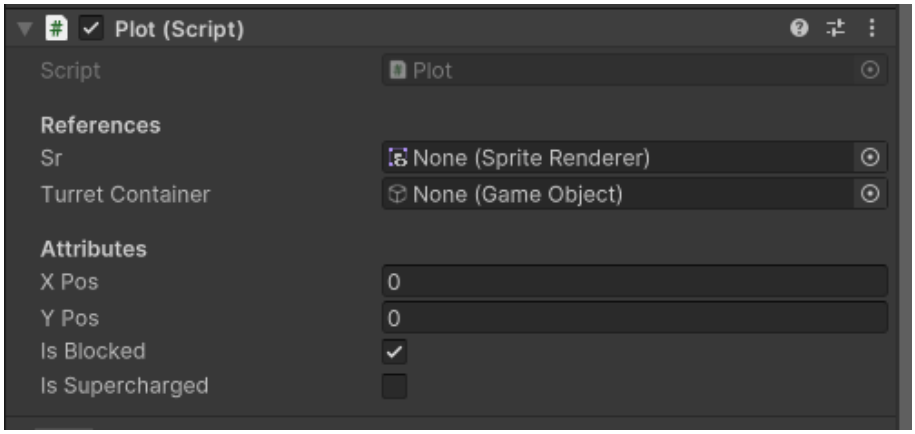


Figure 2.3: Settings for each plot

We can observe all plot types in the figure, for example one appearance of each tower can be seen: (0, 2) for a normal plot, (2, 5) for a supercharged plot, and finally at (0,0) we have a blocked plot.

The special ones are the two colored ones on row 3, where the top one represents the enemy base, and the bottom one the home base.

In figure 2.2 we have the 5 plot types as prefabs, those being duplicated, modified and loaded in the level, and in figure 2.3 we have the settings for the special script for each plot which will be directly modified from the game engine, these being the two integer variables which define the coordinates of the plot, and two boolean varabiels which define if a plot is blocked or supercharged.

**Plot.cs:**

The script associacted with every plot object is **Plot.cs** and has the role of defining the position on the plot to be easily identified by other scripts and to manage the creation of turrets.

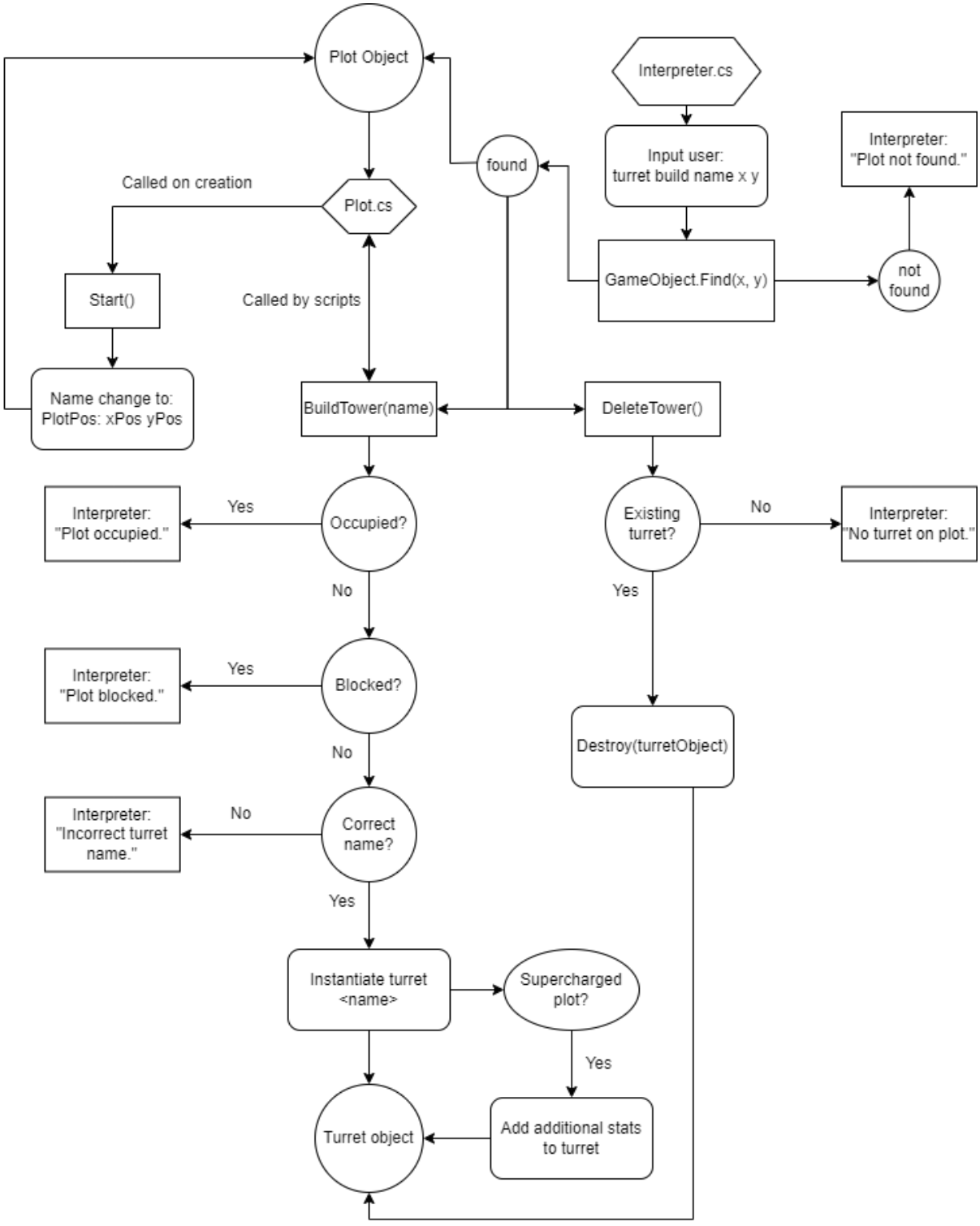


Figure 2.4: Plot.cs and Integer.cs diagram

The figure 2.4 represents all the functions used in the script, those being:

- Start(): This function is called when the level is loaded, changing the names of each plot to ”PlotPos: xPos, yPos” in order to be idenfified in the future by other scripts by using ”GameObject.Find(Name)” which searches for an object by name;
- BuildTower(): The function which is used in creating and assigning a turret on the specified plot. It tests if there is already a tower on the plot and if the name of the tower exists. Also tests if the plot is supercharged, adding bonus stats to the turrret;
- DeleteTower(): Function which allows deletion of a turret from a plot which contains one, which is also being tested.

The functions **BuildTower()** and **DeleteTower()** return an integer which has the role of being sent to the Interpretor to display a message if an input from the player is invalid.

## 2.2 Path

The path which enemies will follow is represented through invisible objects, from which we can extract their position and send it to the scripts required to move the enemies in the game.

In figure 2.5 we have a prefab loaded at the same time as the level, which contains the path objects as children.

An example of the way these objects are laid out is present in figure 2.6, representing the level loaded by the game, but in Scene View, where the diamond shapes from the level are the coordinates set for the path, enemies following them, starting up towards down and following the path. These diamonds are invisible in game view and are only present for debugging purposes.

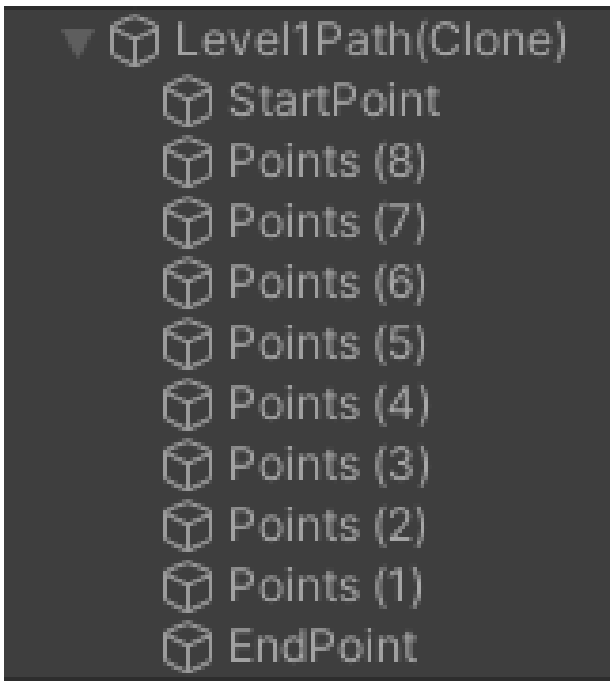


Figure 2.5: Path Hierarchy level 1

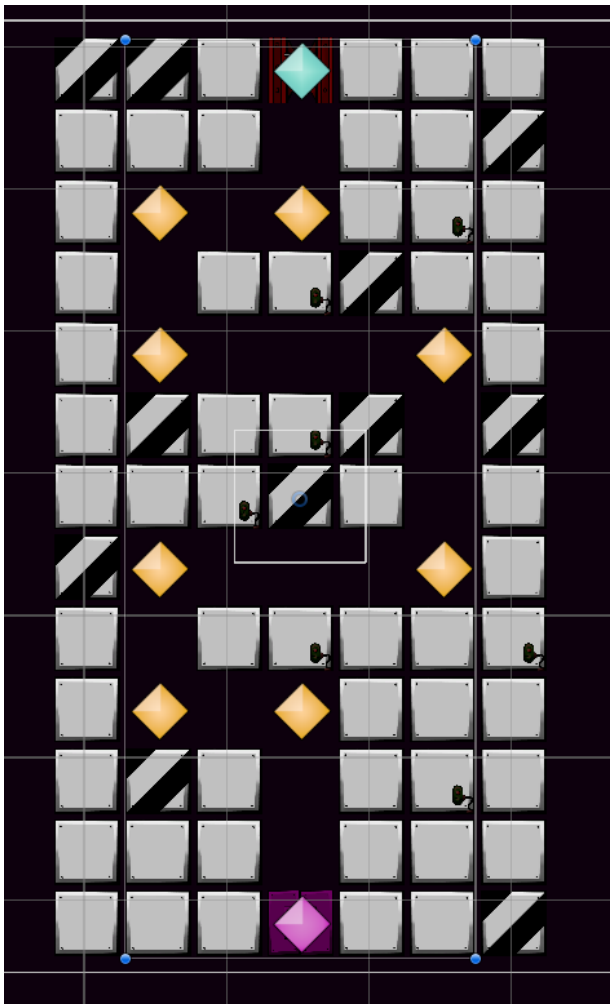

Figure 2.6: Scene View level 1

## 2.3 Level loading

Combining the aspects from the previous sections of this chapter, loading levels is done relatively simply with the help of prefabs.

The Interpretor contains two lists of game objects with receive prefabs that can be loaded through the terminal.

Also, we are making use of a new script which is added in the ”GameSection” object, called **LevelManager.cs**, having the role of monitoring the current level and to load and modify the values of variables necessary for every level according to the progress of the game.

The most important functions of this script are **Update()** which monitores the stage of the game, looking for moments where it needs to act, and the functions which register the resources available to the player to build turrets.

In **Update()** three aspects of the game are verified, if the exit from a level is called, if the level is beaten or if the game is lost. If the exit from a level is called, then a lot of the required variables are reset in order to stop the game.

Moreover, **LevelManager.cs** has the role of modifying the game interface, specifically the objects ”LeftHeader” and ”RightHeader” described in chapter 1, in order to inform the player about the resources and the stage of the game.

More details about resources, winning and losing a level presented in chapter 3.

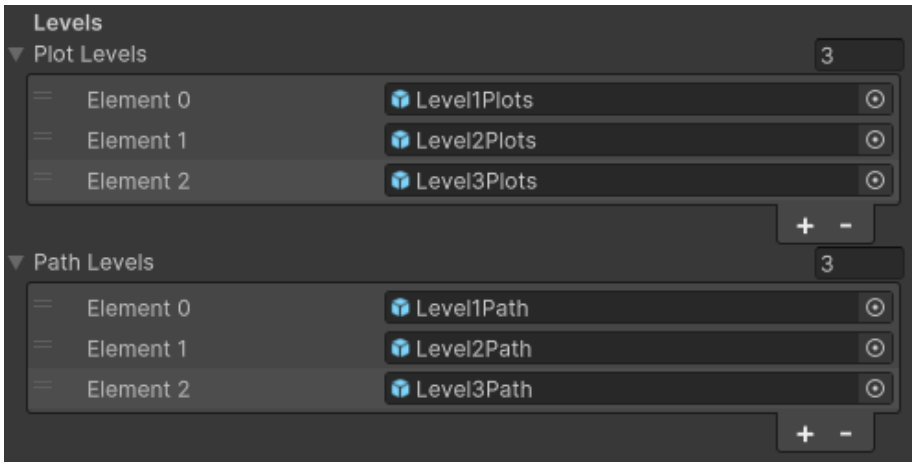


Figure 2.7: Level lists din Interpretor.cs

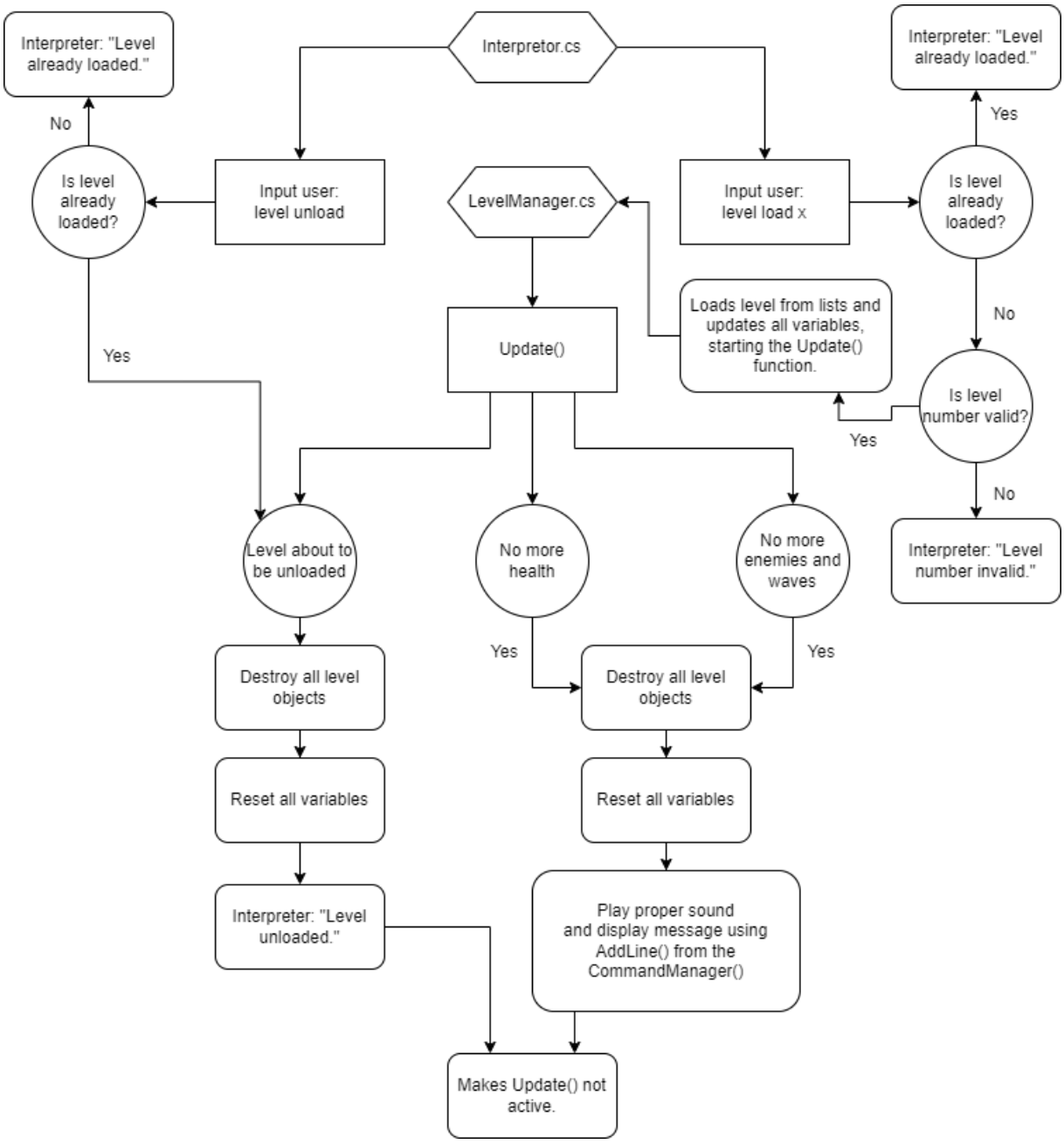


Figure 2.8: LevelManager.cs and Interpreter.cs diagram

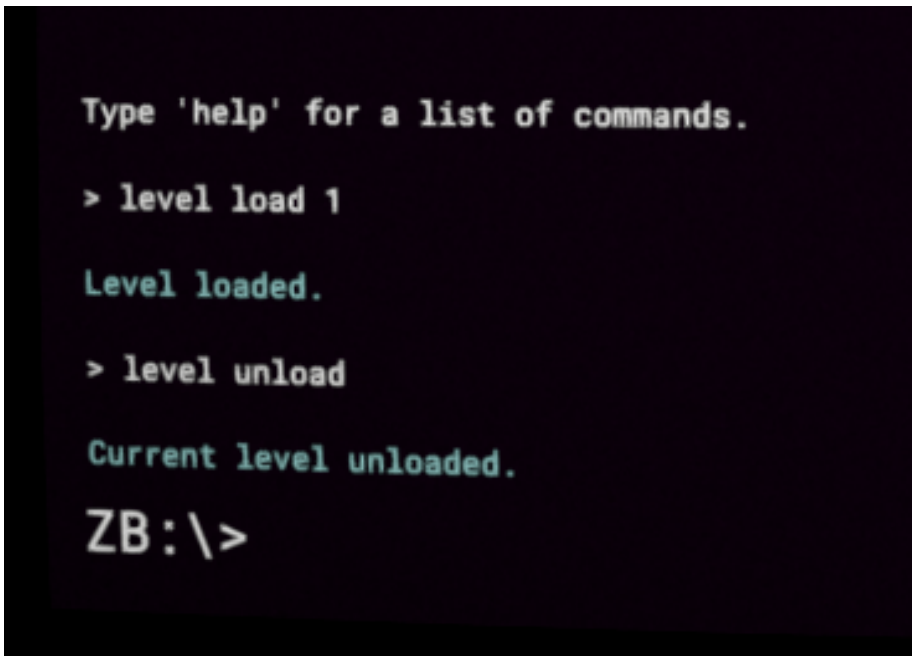


Figure 2.9: Terminal testing level commands


Figure 2.10: Hierarcy after loading level

In the Interpretor we have the command ”level load x” which uses the prefabs from figure 2.7, in order to load them based on the introduced number and ”level unload” which deletes all the prefab which are tied to the level. Like all the previous parts from the Interpretor, there are tests implemented in order to avoid eventual problematic situations.

There are three levels available to the player, each one being implemented with a visual and strategic theme in mind, offering variety.


Figure 2.11: Level 1

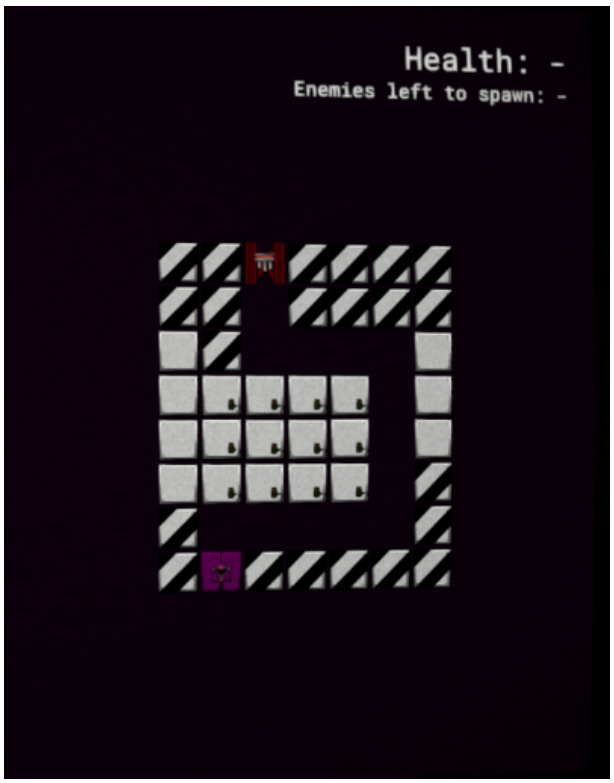


Figure 2.12: Level 2

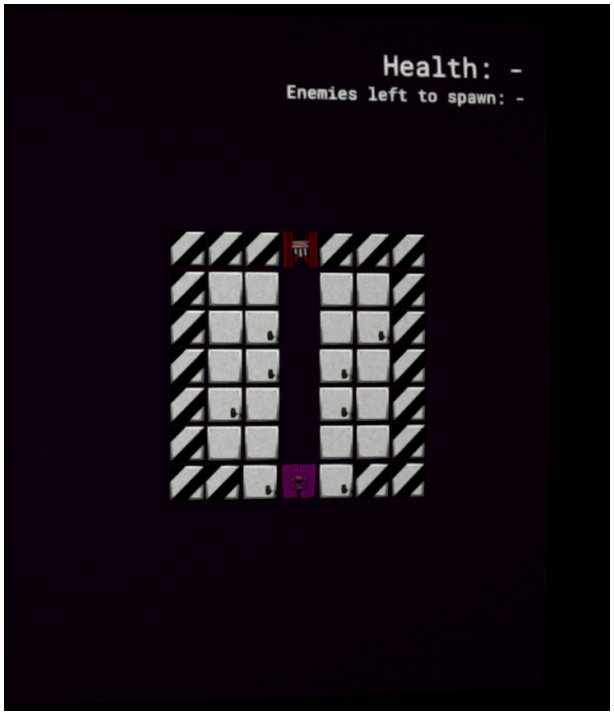


Figure 2.13: Level 3

# Chapter 3

# Entities

We cannot have a complete game without the enitities which we interact with, either with allied or enemy entities. These represent the main factor in growing the engagement between the player and the game.

In this project the player interacts with two major groups of entities, enemeies and turrets, both of these coming in multiple types for diversification and to influence the player to create strategies towards success.

## 3.1 Enemies

Enemies represent the main obstacle towards winning a level, multiplying the more time passes and the more the player is more advanced in terms of resources.

The enemies are creating with the **EnemySpawner.cs** script, at the same time managing the difficulty parameters. Every level has a parameter which adds difficulty to the other variables called ”difficultyScalingFactor”, raising the difficulty using the other parameters, which are:

- baseEnemies: defines how many enemies will appear each wave, does not have an upper limit;
- enemiesPerSecond: defines how the spawn rate of enemies, the harder the game becomes the lower the number falls, has a lower limit of 0.1 seconds;
- timeBetweenWaves: defines how fast the next wave of enemies starts, the harder the game becomes the lower this number falls, until a lower limit of 0.5 seconds.

These values don't seem too hard to manage, so in order to further raise the dificulty, there is a chance that the script spawn different types of enemies. These enemies are:

- Normal: The main enemy; good at everything, perfect at nothing, represents the enemy with the lowest threat which serves as a good distraction for all the other types of enemies which have greater potential;
- Tank: The enemy that can absorb the most amount of damage before being taken down and is very dangerous to the allied base, causing double damage, but moving at a slower speed, making it vulnerable to sustained fire. In greater numbers they can prove quite a challenge even when the player has advanced quite a lot into the level;
- Swift: The most dangerous enemy; without any strategy, this enemy, either in large packs or just a few at a time breaking through the line of defense, can cause massive damage to the base because of their greater speed. Even though their maximum durability is low, in combination with the above enemies they can prove quite a challenge if the player is not smart with their resources.

Once destroyed, every enemy grants resources to the player, which are split in two types: CPU and GPU, these being used in purchasing defensive turrets. Every enemy grants a different number of resources and in a different ratio: the normal enemy grants an equal amount of both resources, the tank offers more CPU resources but less GPU, and viceversa for the swift.


Figure 3.1: Enemy Prefabs

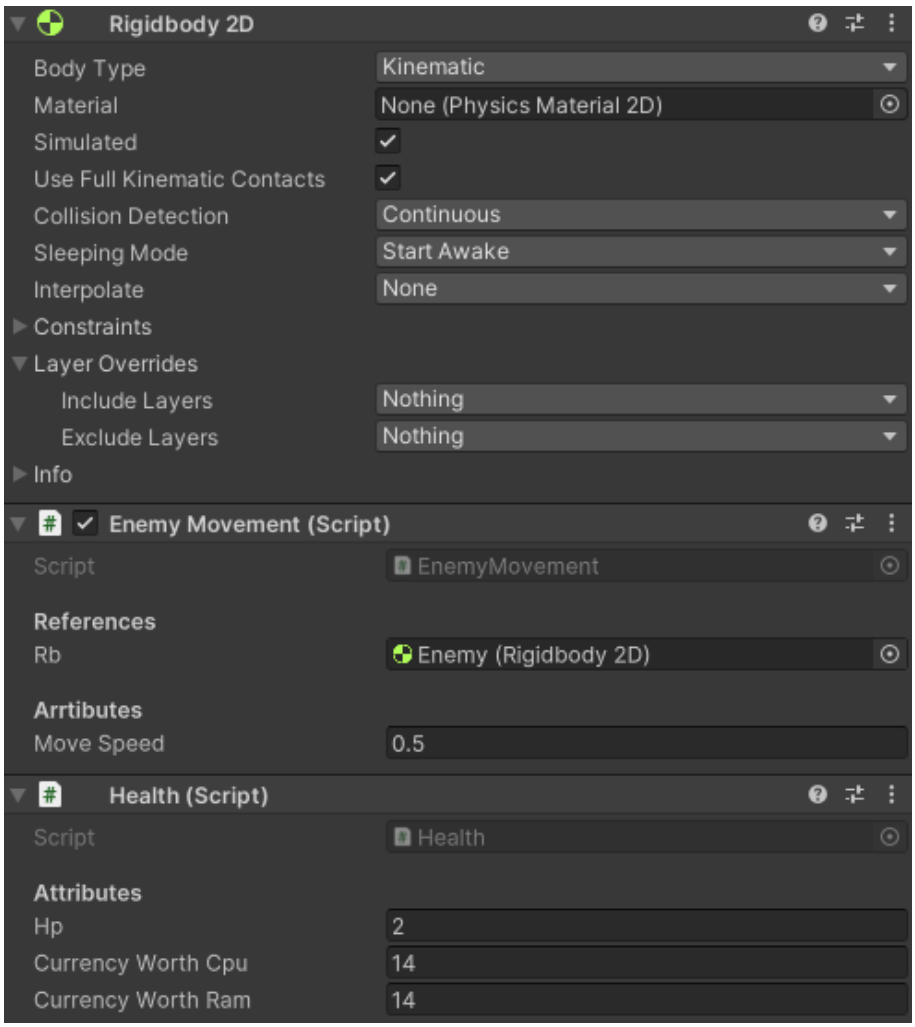


Figure 3.2: Inspector for Normal Enemy

Every enemy has two scripts associated to it which manage all of their properties: **EnemyMovement.cs**, which defines the path each enemy must take towards the allied base and how fast it moves towards it, and **Health.cs** which defines how resistant it and how many resources it grants once destroyed.

Also, enemies contain a game engine element called ”Rigidbody 2D” which is used to detect if a projectile has hit the enemy, causing damage. More details about this are discussed in section 2 of this chapter.

```
public class Health : MonoBehaviour
{
    [Header("Attributes")]
    [SerializeField] private int hp = 2;
    [SerializeField] private int currencyWorthCpu;
    [SerializeField] private int currencyWorthRam;

    private bool isDestroyed = false;


    public void TakeDamage(int dmg)
    {
        hp -= dmg;

        if (hp <= 0 && !isDestroyed)
        {
            EnemySpawner.onEnemyDestroy.Invoke();
            LevelManager.main.IncreaseCurrency(currencyWorthCpu, currencyWorthRam);
            isDestroyed = true;
            Destroy(gameObject);
        }
    }
}
```

Figure 3.3: Health.cs

The **Health.cs** script is simplistic, only keeping track of the enemy's health, and if it reaches 0, automatically destroys the object and grants resources to **LevelManager**.

The **EnemyMovement.cs** script goes through the list of the path object which are automatically assigned to the **LevelManager** and, in order, chooses the next coordinates that the enemy must travel towards. If the script detects that the enemy has reached the final point, the enemy will be automatically destroyed and the vitality of the allied base will decrease. Also, it includes functions which alter the speed of the enemy, utilized for a specific turret.

The **EnemySpawner.cs** script is divided into two parts, the one which creates enemies at the enemy base while the wave is running using the **Update()** function, and the part which modifies the difficulty variables when the wave is not active through the **EndWave()** function and starts another wave with **StartWave()**.

The interpreter has a short implementation, where we have only the "wave start" option which tests if a wave is already in progress, giving the proper answer back. There is no option to stop the waves from coming in order to intensify the pressure of the game.

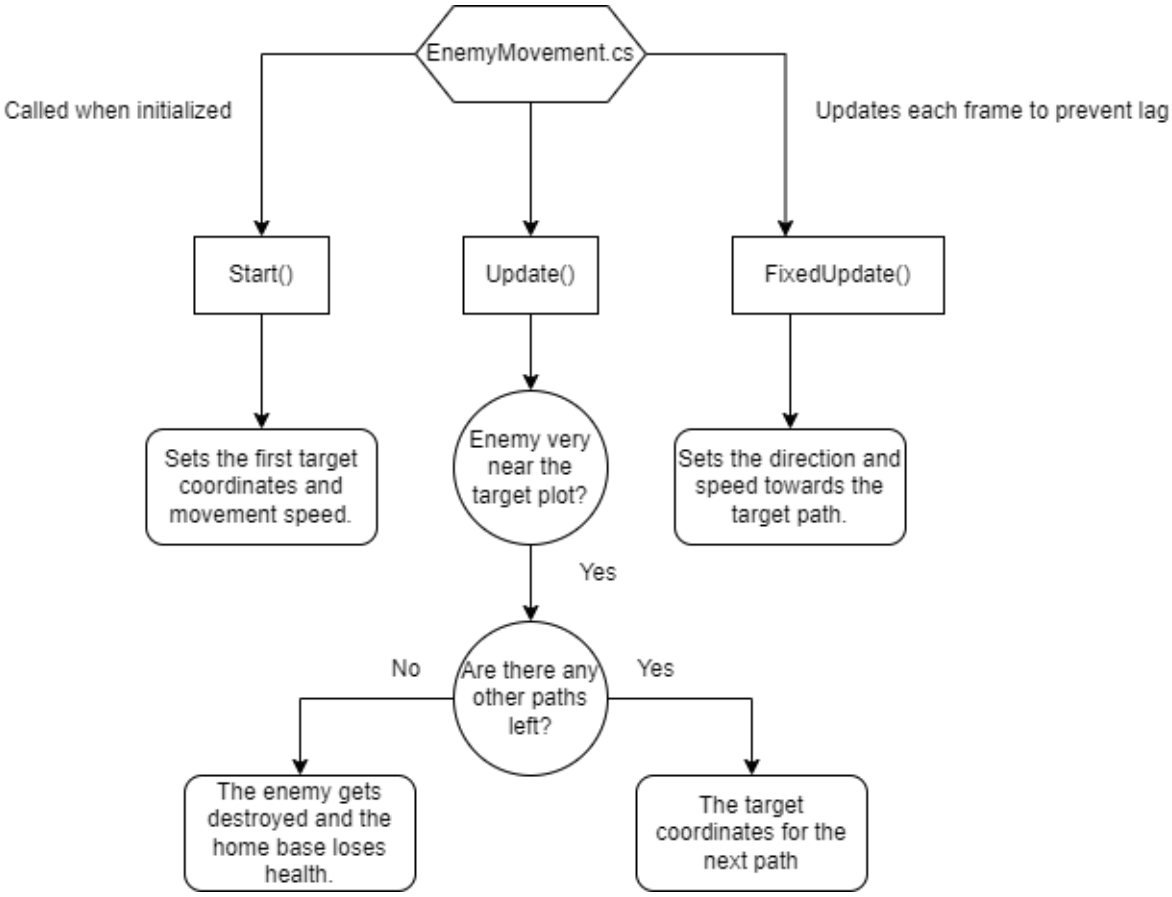


Figure 3.4: EnemyMovement.cs diagram

```
private void Awake()
{
    onEnemyDestroy.AddListener(EnemyDestroyed);
}

void Update()
{
    if (!isSpawning) return;

    timeSinceLastSpawn += Time.deltaTime;

    if (timeSinceLastSpawn >= (1f / enemiesPerSecond) && enemiesLeftToSpawn > 0)
    {
        SpawnEnemy();
        enemiesLeftToSpawn--;
        enemiesAlive++;
        bottomLeftText.text = "Enemies left: " + enemiesLeftToSpawn;
        timeSinceLastSpawn = 0f;
    }

    if (enemiesAlive == 0 && enemiesLeftToSpawn == 0)
    {
        EndWave();
    }
}


// Game management:

private void SpawnEnemy()
{
    int index = Random.Range(0, enemyPrefabs.Length);
    GameObject prefabToSpawn = enemyPrefabs[index];
    Instantiate(prefabToSpawn, LevelManager.main.path[0].position, Quaternion.identity, enemyContainer.transform);
}
```

Figure 3.5: EnemySpawner.cs part 1

```
private IEnumerator StartWave()
{
    yield return new WaitForSeconds(timeBetweenWaves);
    isSpawning = true;
    enemiesLeftToSpawn = EnemiesPerWave();
    bottomLeftText.text = "Enemies left: " + enemiesLeftToSpawn;
}

private void EndWave()
{
    isSpawning = false;
    timeSinceLastSpawn = 0f;
    currentWave++;

    if (timeBetweenWaves > 0.5f)
    {
        timeBetweenWaves -= 0.5f;
    }

    if (enemiesPerSecond > 0.1f)
    {
        enemiesPerSecond -= 0.1f;
    }

    if (currentWave <= nrOfWaves)
    {
        StartCoroutine(StartWave());
    }
}
```

Figure 3.6: EnemySpawner.cs part 2

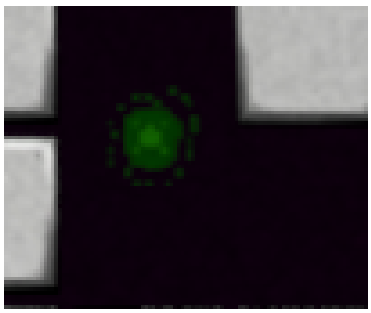

Figure 3.7: Enemy Swift

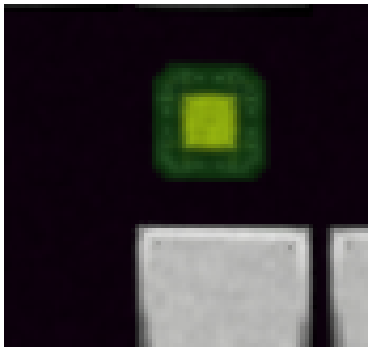

Figure 3.8: Enemy Normal

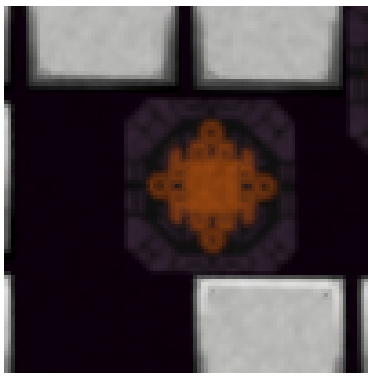

Figure 3.9: Enemy Tank

## 3.2 Turrets and Projectiles

In order to create a defensive barrier against the enemies described above we will need turrets, objects which are placed on empty and unblocked plots in order to launch projectiles towards enemies which can damage and destroy enemies.

There are multiple types of turrets, each having their specific projectile, having different statistics which are found in both, these being:

- Sickle: A short range but with a very good rotation speed turret, with decently high attack speed with fast moving projectile that cause minimal damage, assures a constant source of damage. Because of its low attack range, it is ideal that it's placed so that it covers as much area of the enemy path as possible;

- Slugger: Slow but powerful, with short range and a medium projectile speed, this turret is used as a supplement for the Sickle turret, helping against enemies with alot of health, but must be places strategically in order to avoid its weaknesses, which are the chance of missing due to lower projectile speed, ideally positioned to fire down a corridor;

- Tenderizer: Very slow fire rate, but superior in every other statistic, demolishing every enemy it fires towards. This turret is a weapon which shouldn't be ignored. Having a near global range, it's best placed far from the main path of enemies in order to make way for other turrets;

- Disruptor: The only utilitary turret, it offeres utility instead of destructive capabilities, being the only tower that doesn't fire projectiles, but slows every enemy close enough to the pulse it emits. It's ideal that this turret is placed as often as

possible and cover as much enemy path as possible to raise the efficiency of this and every destructive tower that can make use of this one.

The statistics of turrets are unique and determined by the two scripts associated with them, which are **TurretScript.cs** and **RoundScript.cs** for turrets and projectiles respectively, where the definining variables are:

- TargetingRange: radius in which a tower can begin rotating and fire towards an enemy
- RotationSpeed: how fast the tower can align with an enemy;
- FireSpeed: how often the tower shoots;
- RoundSpeed: how fast the projectile moves;
- RoundDamage: how much damage does the projectile deals.

The Disruptor tower is a special case, having different types of statistics and no projectile script, using only **DisruptorScript.cs**:

- TargetingRange: radius in which enemies get slowed;
- APS: how many slowing pulses are emitted per second;
- FreezeTime: how long the enemies are slowed.

If a turret is placed on a Supercharged Plot, it will benefit from additional TargetingRange, RotationSpeed, FireSpeed, and in the case of the Disruptor turret all of its stats will grow.

Every tower that shoots projectiles has a point that rotates with the tower, which indicates the location where projectiles get created, copying coordinates and rotation.

Projectiles contain the **RoundScript.cs** script and a Rigidbody2D property, which, once entering the rigidbody of an enemy, destroys the projectile and causes damage to it.

Scriptul are rolul de a asigna direcția și viteza pe care o ia proiectilul atunci când este creat, și distrugerea lui în contact cu un inamic.

The script has the role of assign direction and speed which the projectile will have once created, and the destruction of it when in contact with an enemy.


Figure 3.10: Prefabs for turrets and projectiles

Also, the script for the turret are **DisruptorScript.cs** for the utility tower, and **TurretScript.cs** for the rest, where the first script tests if enemies are in the targeting range of the turret, slowing them when the turret has finished cooling down, and the second one rotates the tower to the first enemy in targeting range and generates a projectile towards the enemy based on the fire rate.

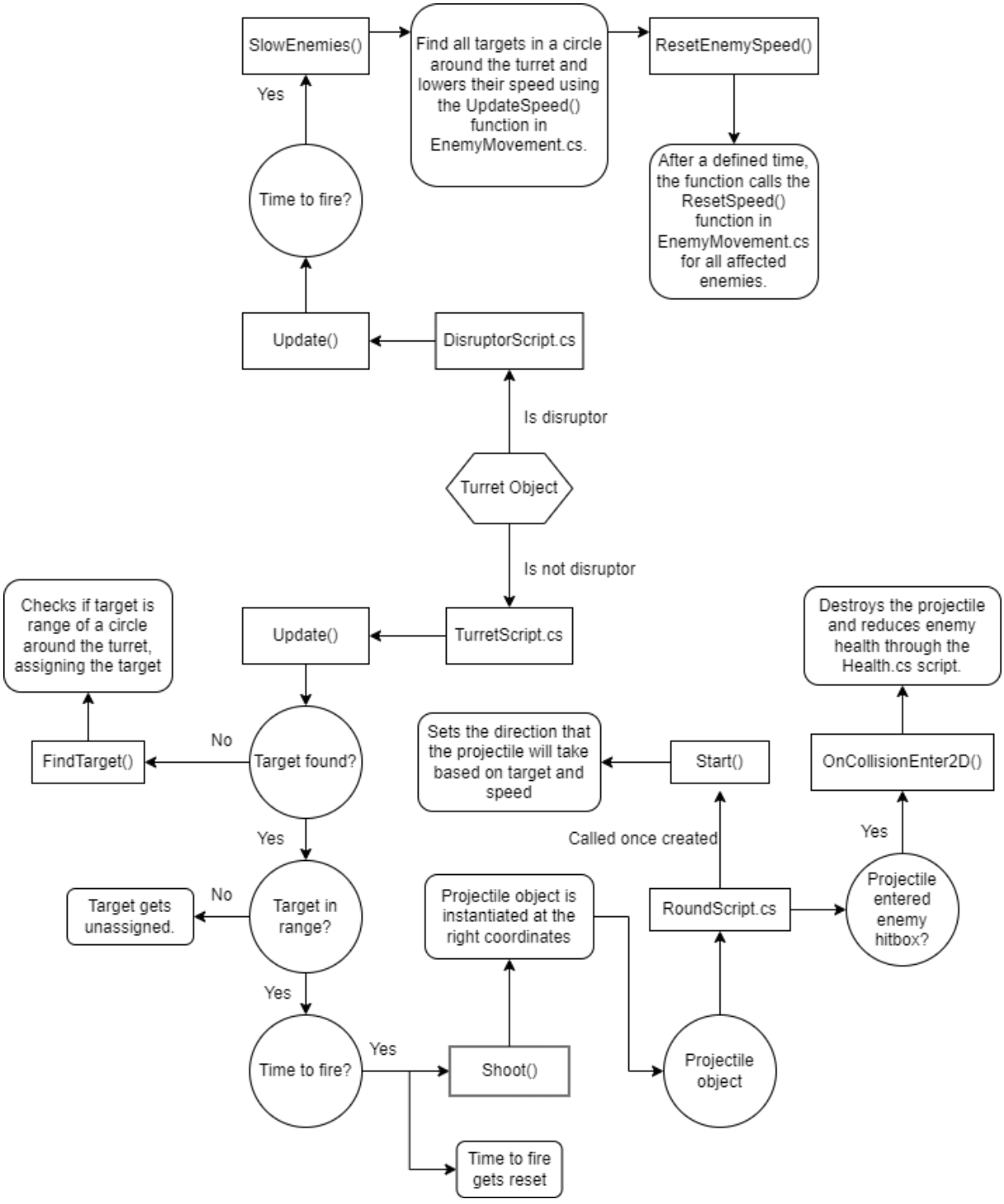


Figure 3.11: Turrets and Projectiles diagram

Figure 3.12: Turret Disruptor

Figure 3.13: Turret Tenderizer

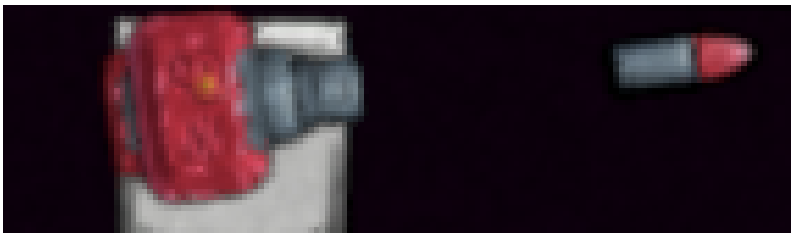

Figure 3.14: Turret Slugger

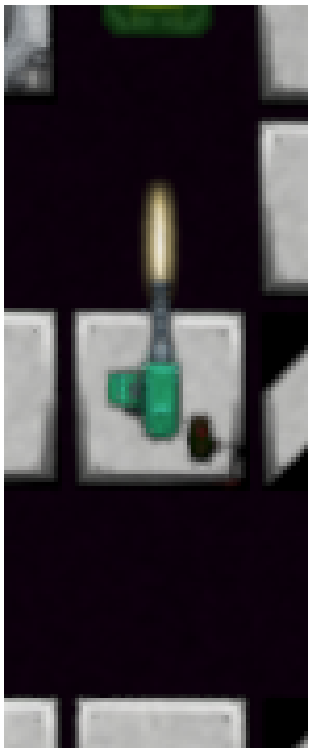

Figure 3.15: Turret Sickle

# Chapter 4

# Game demo and additional aspects

In the last chapter of this project I will walk through how a game session is played, and additional visual and audio elements to spice up the game experience.

## 4.1 Game walkthrough

Once in the game, as shown in figure 4.5, the player is encouraged to type **help** to receive information about the possible commands of the terminal. Also, there are some advanced help commands: **help turrets** and **help levels**, which explain everything about all the types of towers and all available levels.

Additionally, a command called **intro** is implemented with the role of explaining the rules of the game to the player, in order to push him towards creating strategies to beat the game.

After the player is well informed, he loads a level, initialising the scripts and prefabs talked about in the previous sections.

The player receives some initial resources in order to place defensive turrets before the waves of enemies come out.

When ready, he calls the start command, which spawns enemies constantly until there are no more waves and enemies left for this level, or when the home base suffered fatal damage.

The number of waves is hidden in order to induce suspense, making the player ask themselves ”How much left until it’s over?”.

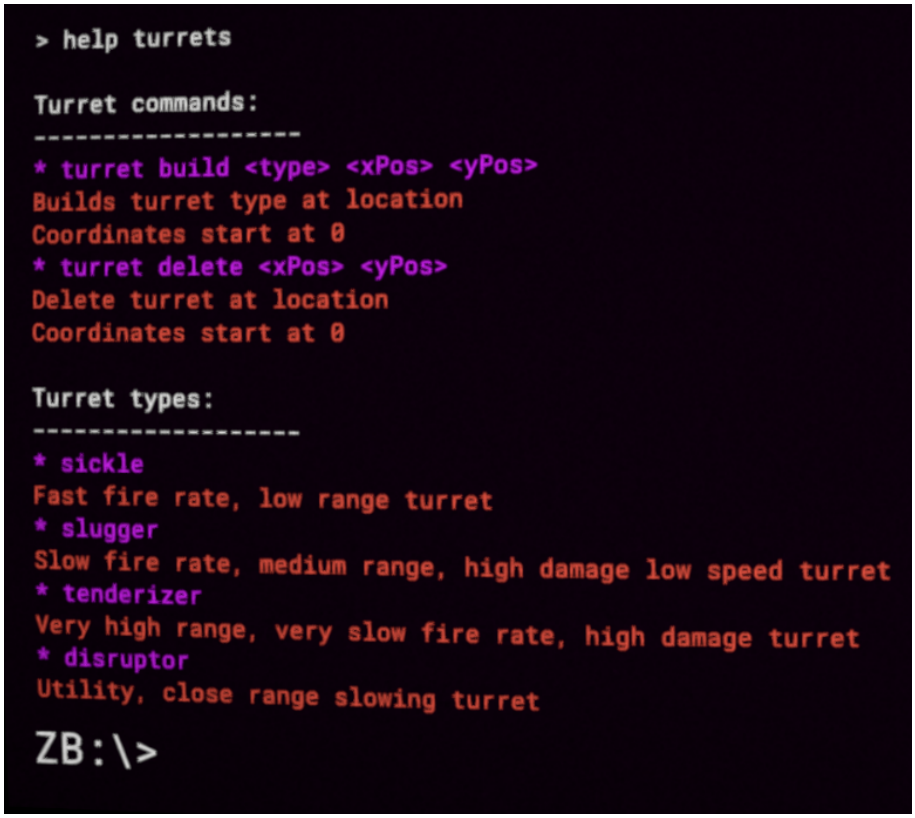


Figure 4.1: Example advanced help

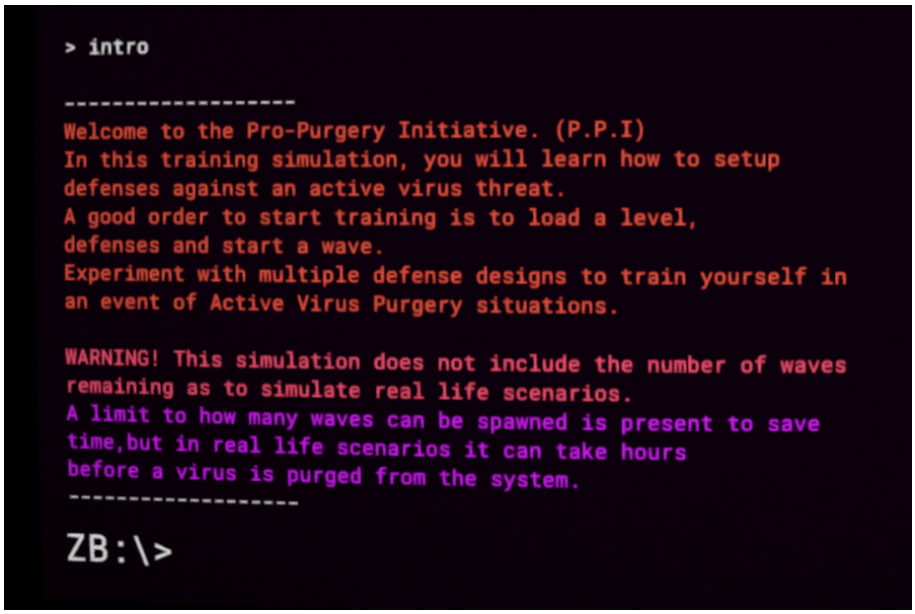


Figure 4.2: Intro command


Figure 4.3: Gameplay example

## 4.2 Aditional aspects

In order to refine the aspect of the game, certain special elements have been added, both visual and audio in order to add atmosphere to the game:

**Visual:**

Besides the text coloring previously discussed about in the Interpretor section, I also created an object called **CameraFilter** which modifies the ”lens” of the camera, imitatind an old CRT monitor, making the game look like an old terminal.

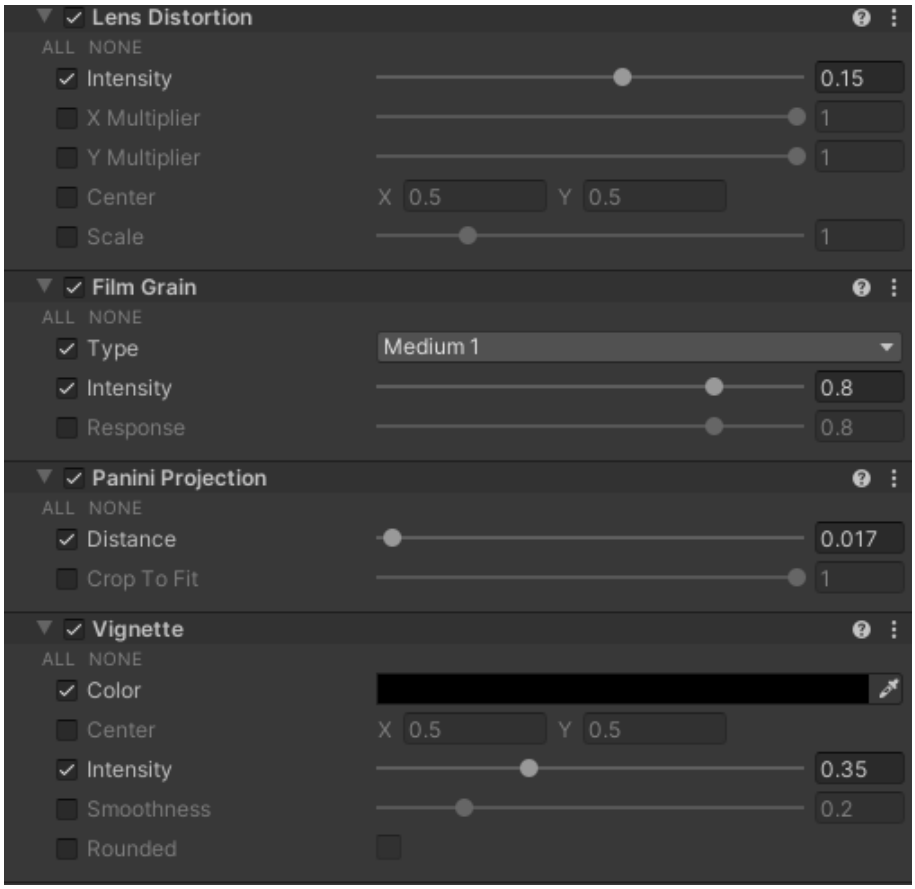


Figure 4.4: Camera filters

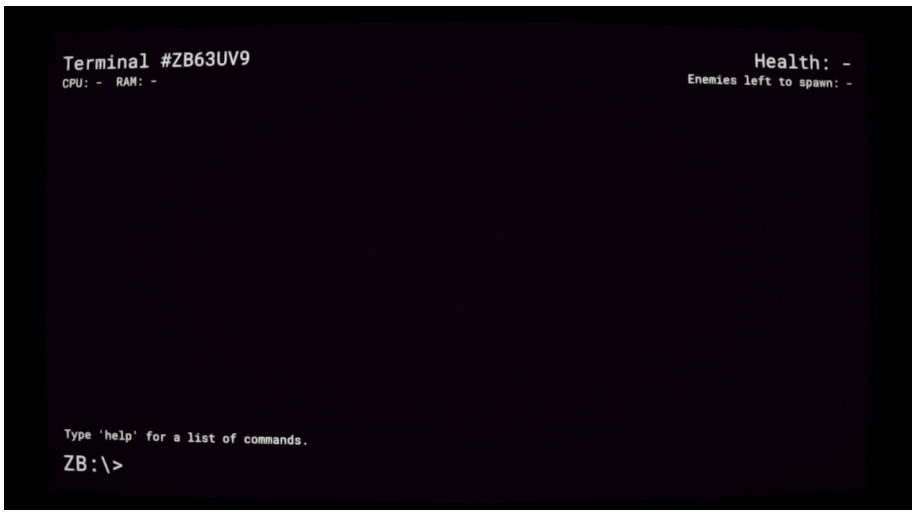


Figure 4.5: Game terminal with camera filters

**Audio:**

Sound effects have been implemented to give an unique atmosphere and to highlight certain moments of the game. The implemented sounds are:

- Background music, which represents sound effects of a real-life server room, a constant white noise relaxing ambience;
- Win sound, which represents a friendly shutdown noise, marking the survival of a level;
- Lose sound, which represents an unexpected and sudden shutdown sound, grabbing the player's attention and marking the failure of surviving a level;

Besides the background music, sounds are called by the **LevelManager**, used in the necessary situations.

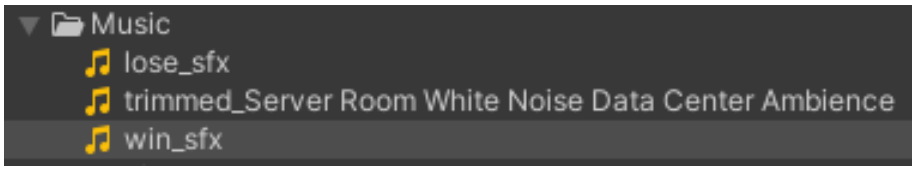


Figure 4.6: File list for sounds

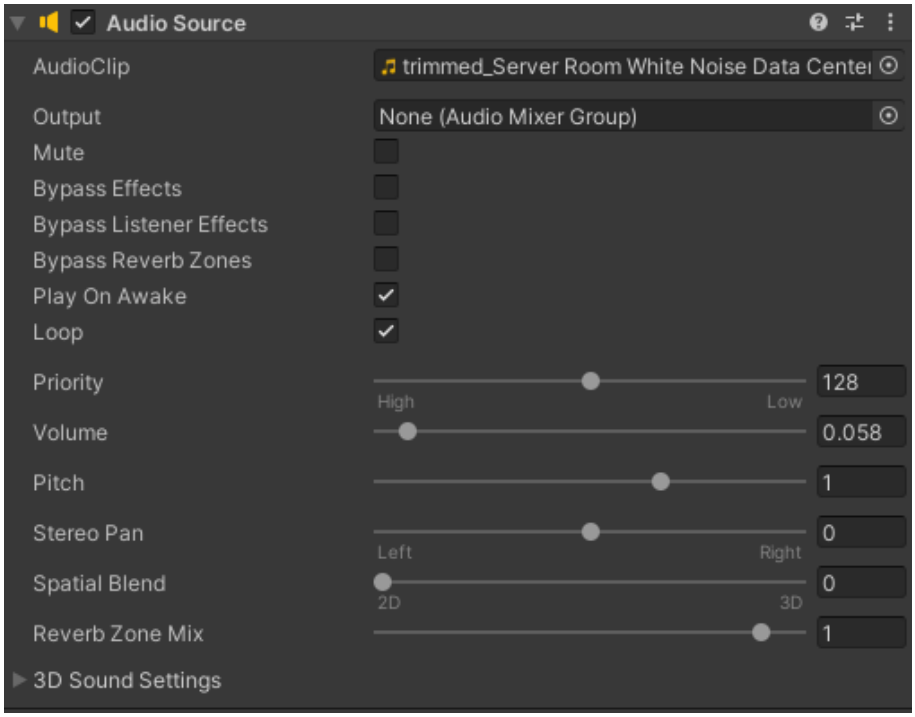


Figure 4.7: Background music settings

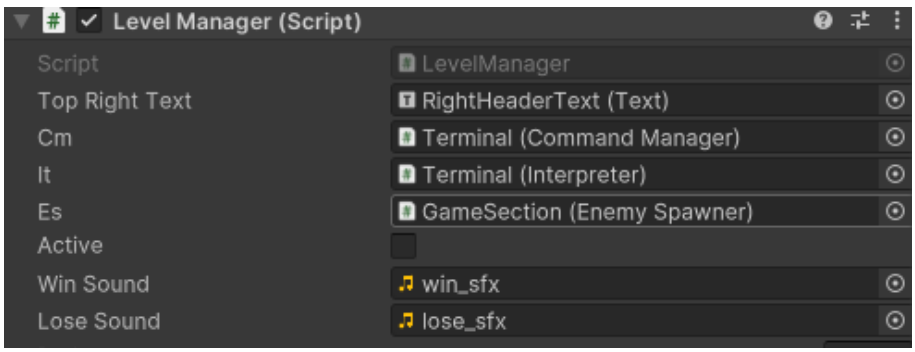


Figure 4.8: Win and lose sounds in LevelManager

# Conclusions

In conclusion, through my detailing about the creation of this game, important design decisions can be extracted, which can be used to scale this type of project to great heights, using modular implemented programs so that many other characteristics can be implemented, such as additional stats for towers, difficulties which can scale differently depending on preference, more plots with different properties etc., and more types of objects for enemies and turrets with their projectiles.

This project represents, in my opinion, a good foundation for entering the game design domain, which helped me form my train of thought about how to identify different problems which I may stumble upon while creating the game, and how to solve them in an efficient and creative way, and gave me a perspective about what i need to improve on in the future to become more experienced in this domain.

# Bibliography


- Unity Manual, *https://docs.unity3d.com/Manual/index.html*
- Texture Pack 1, *https://opengameart.org/content/gameboy-tower-defense-sprites*
- Texture Pack 2, *https://opengameart.org/content/tower-defence-basic-towers*
- Background Music, *https://youtu.be/-BofZ-seOc0?si=KAEGoH-fRiYZtzqu*
- Win sound, *https://freesound.org/people/Cooltron/sounds/332919/*
- Lose sound, *https://freesound.org/people/13FPanska_Sychra_Petr/sounds/379385/*