Application of genetic algorithms and swarm behaviors for the development of npc's in video games

Jon Hodei Martínez Soto

Final Degree Work
Bachelor's Degree in
Video Game Design and Development
Universitat Jaume I

June 4, 2020

Supervised by: Luis Amable García Fernández

To my dad, Juan Manuel,
who moved earth and sky in search of
video game careers so that I could fulfill my dreams
from when I was a kid. It seems that what I said when
I was 12 years old, that I wanted to be an architect, didn't come up.
I was predestined to develop video games, I know.

***

To my mum, Blanca,
who has supported me and has been there for me
all these years, even when I didn't have the strength to continue
and wanted to leave everything. Luckily that didn't happen in the end.
Let's see if in the future we can achieve what you once said,
to start a video game company and work side by side with me as a secretary,
so you can balance these 4 years that I have been 600km away from home.

***

To my older sister, Ainize,
with whom I have shared this hobby of video games
since I was a child. I miss being at grandma's house trying
to complete the pokedex together in Pokemon Diamond/Pearl.
Or spending whole afternoons playing Wii.
You know I'm better at tennis than you are.

***

To my little sister, Ziortza,
with whom I have also shared experiences with video games,
this time more like an instructor showing his disciple
the wonders of the art he teaches.

Acknowledgments

First of all, I would like to thank my Degree Final Project supervisor, Luis Amable García Fernández for his help provided before the beginning of the development, which was crucial in guiding the vague ideas I had in my head. As well as for his advice during the development of the project.

Abstract

This document presents the project report of the Video Games Design and Development Degree Final project by Jon Hodei Martínez Soto. It is a videogame titled *Zombies Are Dumb* that consists on exploring how to include genetic algorithms applied to a group of agents with the objective of being able to learn the player patterns and adapt to them. This will create a greater challenge considering that the AI will change its behavior depending on how the player plays.

Besides, the project also integrate swarm behaviors to create a sense of group and coordination between the agents. These two main ideas will cause an unpredictable playing environment for the player and, therefore, an unique and different experience each time.

Contents

Chapter 1

Introduction

Contents

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 have advanced a lot in recent years, as same as artificial intelligence. Both things combine very well together. Nowadays we can find AIs capable of defeating more than 99% of players who face it, as can be seen with AlphaStar, an AI specialized in playing *StarCraft II*. There are also AIs capable of improving the textures of a videogame to 4k, an example is the remake of *Metal Gear Solid: The Twin Snakes*. These things are both amazing and scary. I'm not talking about fear of machines rebelling against humans, no, far from it. I'm talking about the fear that, as a future videogame designer and developer, everything will be so advanced that it will be impossible for me to reach all that knowledge and live up to it or even to improve it.

That's why after having studied new things in Artificial Intelligence subject, the idea of knowing a little more about AI, crossed my mind.

It is clear that AI is very broad and covers many fields. But since we started working with Unity, we've seen some AI things: basic stuff for enemy NPCs, walking around in a navmesh, attacking the player, etc. But I wanted to do something different. I wanted the AI not to be preset or at least not to be noticed. So I decided to work on random, unpredictable AIs. The final decision was to develop a video game where the player has to face different types of zombies to rescue his brother. What's relevant is the AI of these zombies. Some of them are working with pathfinding based on a genetic algorithm and others acting as a swarm.

1.2 Objectives

Based on the motivation of the work, there are some goals to achieve:

- Understand how genetic algorithms work to apply them on different problems.
- Implement artificial intelligence techniques to create unpredictable NPCs.
- Understand how emergence behaviors work to adapt it to different NPCs objectives.
- Implement swarm artificial intelligence techniques to create coordinated NPCs.
- Learn how to combine these two main elements.
- Develop a demo featuring the results obtained in the mixture of both AI techniques shown.

1.3 Environment and Initial State

As I mentioned before, my initial interest revolved around artificial intelligence. I had read about artificial intelligences with the ability to learn, AIs that were able to progress based on their experience, that could find their way out of a maze after several attempts. So I did a brief search: *AI in videogames.* I had over 1.2 billion results. Many of them were things that I wasn't interested in, things that I already knew how to implement. I needed to narrow down the search so I tried with "*AI learning in videogames*". Okey so now I had 300 million results and 86 million videos.

I started watching videos about the things AI was able to do in video games. That's when I came across certain terms: reinforcement learning, neural networks, deep learning, neuroevolution, etc. I then looked for such techniques applied in video games. A video of a scene in unity in which a car was learning to drive around a circuit without crashing was what led me to this project, to make this DFP. I thought about making a video game that would take advantage of those capabilities that the car had, but I didn't want to do anything related to cars. While thinking about it, I came across a video about an AI that had learned to play hide-and-seek. That meant that was able to search and find the other NPC. "*So... if I do something similar, but the one you're looking for is the player?*" I thought it was a good idea, but I also didn't want to make a video game about hide-and-seek. I thought it could be a bit boring to play, so a new idea crossed my mind. I could make a video game where the player's enemy tries to find and kill him. This initial idea, along with the car video (in which about 30 cars were generated until one of them found the right road for that track) ended up becoming a video game where a group of enemy NPCs would look for the player to kill him. Since I wanted it to be a group of enemies, a kind of herd, it occurred to me that they could be zombies and therefore the herd would be a horde.

I already had the main idea of the video game: a zombie survival game in which the zombies would learn (I didn't know how yet) to locate the player in order to attack him. With this proposal in hand I went to see Luis Amable García, my tutor, to get his opinion and help me understand how I could develop it. He proposed me two options, the first one, Q-learning, a reinforced learning technique. He provided me with an article [1] that explained Q-learning and teaching how it could be programmed. The second option was emergent behaviors, in which we can find swarm behaviors or boids for example.

I started researching Q-learning and trying to program it to see if was similar to what I wanted to do. I realized that it didn't work, it wasn't possible to do the main idea that I had. Q-learning needs a static system, for example, given a maze where A is the entrance and B is the exit, finding the shortest path from A to B. But I needed something that would work in dynamic systems, my player was not going to be still, he was going to move, so Q-learning could not work efficiently in that situation. I met with Luis again and explained the problem I had. We ended up talking about genetic algorithms and thought it might work. But something similar happened. If in the first generation some subjects tried to find the player, and when they died we evaluated which of them had been closer to finding him (let's call them winners) so that the next generation would inherit part of those genes, the new generation would tend to go where those first winning subjects went. In short, they would go to where the player was in the previous generation. But let's remember that the player can move, so with the genetic algorithm we were not solving that dynamic system either. Still, I found the genetic algorithm interesting and I wanted to work on it. So I took another approach to the video game.

Since the zombies couldn't learn to find the player because he would be moving, I decided to include another NPC that would be static, so the zombies would be able to search him. To give this some context in the game, I thought that the NPC would be the player's brother, who would have been lost in the woods. This gave the player a target, a thing that wasn't thought yet.

The player would have to go through the map, a forest, to save his little brother by killing any zombies he came across on the way. The problem with this is that if the zombies had the only task of looking for the brother, the player could walk around without any threat. So I thought I'd include a different group of zombies who would just have the goal of killing the player. But I didn't want these zombies to know where the player was and go after him. I wanted to make it realistic. Luis proposed me, like in our first meeting, the emergent behaviors. I could get this other group to act together, like a swarm, to really feel a horde of zombies wandering around the area until they see the player and attack him.

And this is how we ended up starting the development of the project. With clear ideas of the gameplay development and how the zombies would work internally. During the development of the project I would be able to support myself in the great community that there is in the Unity Asset Store, and thanks to the assets, models, textures and others that they offer, I would be able to focus on other aspects of the game more oriented on the programming of the game.

Chapter 2

Planning and resources evaluation

Contents

This chapter deals with a technical part of the work. It also shows the planning for the development of the project.

2.1 Planning

The planning is a very important task in the project to keep the workflow. Here it is detailed how the development of the project is divided. Some things have been changed from the initial days until now. The planning is the following:

- **Task 1 (15 hours)**: reading and learning theory of genetic algorithms.
- **Task 2 (60 hours)**: Think the main problem that the genetic algorithm will solve, and design (develop) that algorithm. Investigate what kind of problems I can solve with a genetic algorithm and after that, think what genotype will have each zombie and what will be the parameters to decide the best zombie of each generation. (I have on mind to do a pathfinding algorithm)
- **Task 3 (20 hours)**: Be able to “train” those agents and save the genes for the final version of the video game.

- **Task 4 (10 hours)**: Reading and understanding theory and examples of emergence behaviors.
- **Task 5 (30 hours)**: Think and design which swarm behavior it's the best for our type of NPCs and developing it.
- **Task 6 (45 hours)**: Coordinate genetic algorithm based agents and swarm behavior based agents to make them look like they belong to the same group.
- **Task 7 (30 hours)**: Searching or making 3D models for the game characters and implementing them in unity with animations.
- **Task 8 (60 hours)**: Project report and other documents.
- **Task 9 (30 hours)**: Finishing a playable demo with menus, UI and at least 1 complete level.

Total hours: 300.

Initially these tasks were to be completed in the order in which they are written, but as was said before, some things have been changed from the initial days until now. So here's a summary of these tasks in the order that have been done (see Figure 2.1)

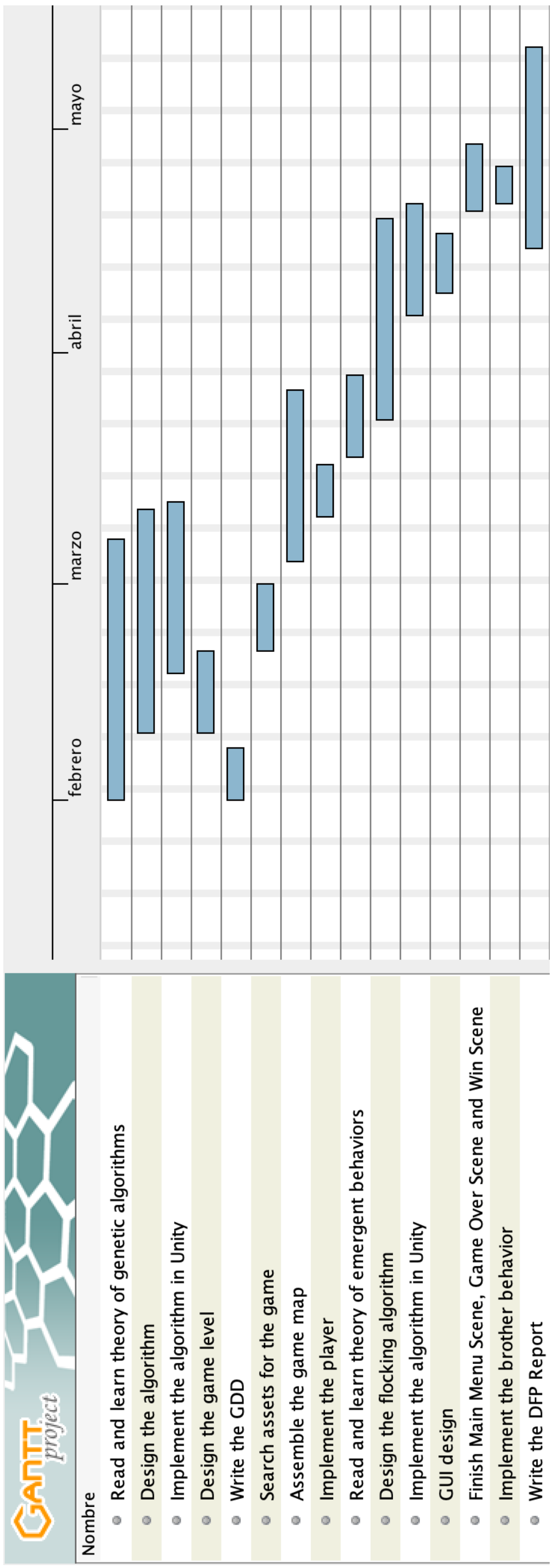


Figure 2.1: DFP task planner (made with Gantt Project [2])

2.2 Resource Evaluation

In this point is listed the software and hardware needed to do this project. Not all are essential but they help the correct development of it.

- **MacBook Pro (i7 CPU, 16GB RAM and GPU Radeon Pro 555)**: i will use this PC in a regular way
- **Unity 3D 2019.2.6f1**: the engine of the project [3].
- **Visual Studio 2019 for Mac**: attached with Unity 3D is a very useful tool for coding and allows switching between Unity and Visual. Besides, Visual Studio [4] has a powerful debug function which will help me to find and solve bugs. I will use C# language because it is the one I use for all the projects.
- **GitHub (GitHub Desktop)**: a git repository [5] where I will upload my work. It is a good tool to avoid losing progress in the project if a problem occurs on the computer.
- **Adobe Photoshop CC 2019**: it will be used to design the user interface (GUI) and some of the elements of the menu scenes of the game. [6]
- **Adobe Premiere Pro CC 2017**: to edit the videos that are included inside the project and the videos that this report contains. [7]
- **Trello**: a website used to control the work. Divided on pending, doing and done it can be used to divide the whole work into tasks. It helps a lot with the organization. [8]
- **Unity Asset Store**: a website for buying and selling assets. This page offers a very large amount of free assets, so it is very useful to download 3D models and focus on the game programming. [9]
- **Mixamo**: a website that allows you to apply animations to 3D models you upload and then download them for integration into Unity. Very useful for adding animations to the player and NPCs. [10]

Chapter 3

System Analysis and Design

Contents

This chapter deals with the requirements analysis, design and architecture of the proposed work. It also shows the design of the *GUI* (see figure 3.1), commonly known as the game user interface.

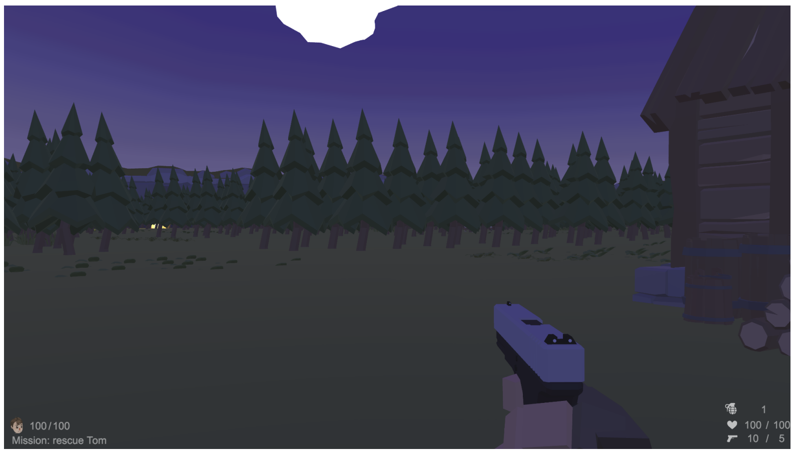


Figure 3.1: Image that shows the Game User Interface

3.1 Requirement Analysis

Having a job to do can also be seen it as a problem to solve. To address this problem a preliminary analysis of the necessary requirements must first be carried out. As I once heard someone say: *just as we put the horses in front of the cart, let's put a problem first and then a solution.*

Well, here the "problem" is to make a video game, a first-person-shooter mixed with a survival zombie. Maybe this is too generic and there aren't many specific requirements that can be drawn from it, so let's describe the problem (the video game in this case) better.

Zombies are dumb is a *FPS* (First Person Shooter) video game in which the main character will try to save his lost brother. He will do this by defeating the enemies he meets around the map, trying to find the way to his brother through the labyrinthine forest.

As soon as the game is opened a main menu (see figure 3.2) will appear with 5 options: *Play*, *Tutorial*, *Options*, *Intro* and *Quit*. Pressing the latter will close the game. With the *Options* tab, another menu (see figure 3.3) will open where the user can choose if he wants the music to be active or not and can adjust the difficulty between *easy*, *normal* and *hard*. Pressing *Tutorial* will change to a scene (see figure 3.4) where the basic controls can be tested. The *Intro* button is used to watch the introductory video. This video serves to put the player in context about what the story is about or what his mission will be during the game. And obviously, the *Play* tab is to start playing the game.

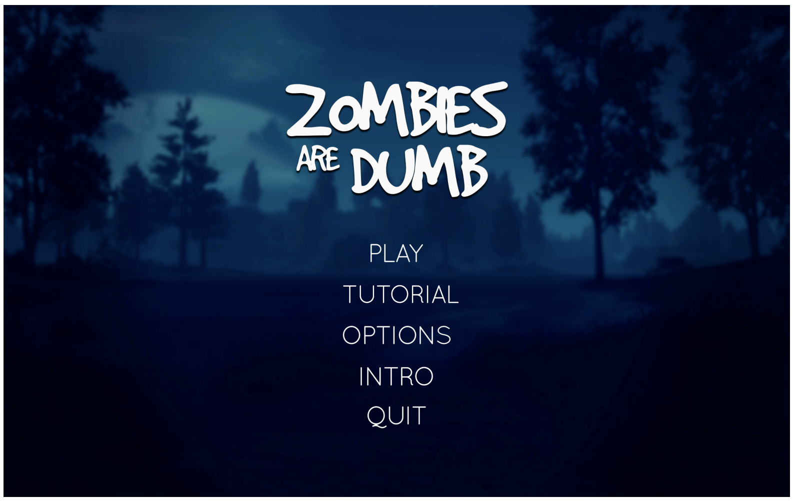


Figure 3.2: The Main Menu screen.

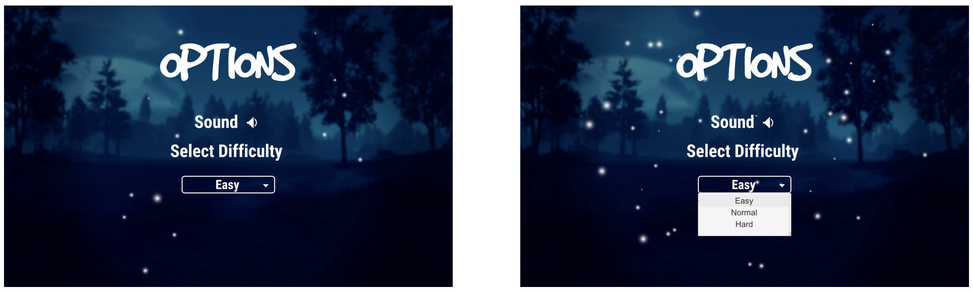
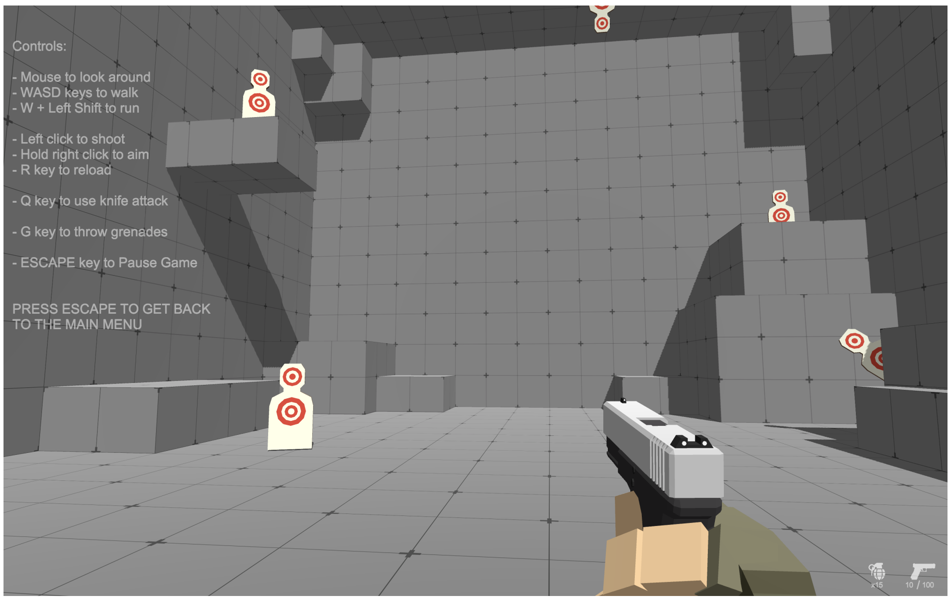


Figure 3.3: Options screen.



Controls:

- Mouse to look around
- WASD keys to walk
- W + Left Shift to run

- Left click to shoot
- Hold right click to aim
- R key to reload

- Q key to use knife attack

- G key to throw grenades

- ESCAPE key to Pause Game

PRESS ESCAPE TO GET BACK TO THE MAIN MENU

Figure 3.4: The tutorial scene with the controls instructions.

Once inside the game it is possible to perform different actions with the character. With the WASD keys the player will move around the map. While moving, if the SHIFT key is kept pressed, the player will run. With the Q key the player will do a knife attack. With the G key the player can throw grenades (if he has any). If the player's gun has bullets, he can shoot with the left click. He can also aim with the right click to improve his accuracy when shooting. You can reload the gun with the R key. Along the map the player can find ammunition, first aid kits and some stuff which can be taken by interacting with them with the E key. All these controls can be tested, as said before, in the Tutorial, where a list is provided with the corresponding keys so that the user can adapt to the basic mechanics.

The main character will try to defeat the different enemies he will meet on the way. There will be different types of enemies, three different types of zombies. I wanted to differentiate them by putting a different color on each type. The blue ones, acting as a swarm, will walk around the map until they find the brother and will then go to kill him. These ones will ignore completely the player. The orange ones will act just like the blue ones, but they will attack the player and ignore the brother. Finally there are the grey ones whose main objective is to find the brother (using a genetic pathfinding algorithm). They will have a period in which they try to find him and after a while they abandon that task and will stay patrolling. While they're patrolling, if they see the player or the brother, they'll go and attack him. So we can say, the last ones are the most dangerous, since they attack both. In the current version of the game, they are all the same color. This way the player doesn't have the advantage of knowing which of the zombies he's looking at will attack him, ignore him, etc.

Finally inside the game we will also have the option to press pause. Pressing the *ESCAPE* key will pause the game. A screen (see figure 3.5) with several options will be displayed: resume, return to the main menu, adjust volume and adjust mouse sensitivity.

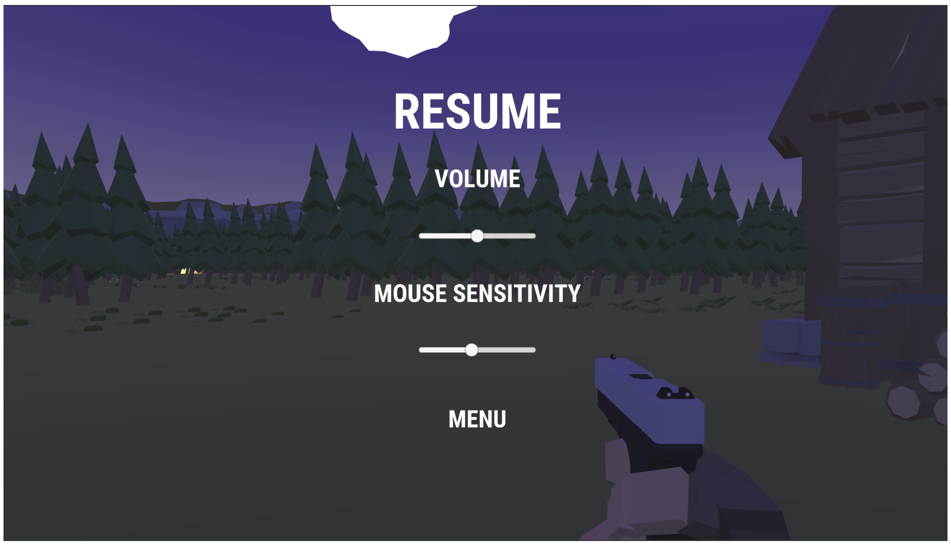


Figure 3.5: Pause menu.

3.1.1 Functional Requirements

A functional requirement defines a function of the system that is going to be developed. Let's see these requirements:
R1. The player can choose the difficulty.
R2. The player can change the volume of the music.
R3. The player can quit the game by pressing Quit button.
R4. The player can start the game by pressing Play button.
R5. The player will be able to move the character sideways (with AD keys) and frontally (with WS keys) troughout the map.
R6. The player can attack enemies with a knife when the Q key is pressed.

R7. The player can throw grenades when the G key is pressed.
R8. The player will be able to attack the enemies form a distance with a gun.
R9. The player can collect objects by interacting with them by the E key.
R10. The player can reload his gun if the gun magazine it's not full and the player has ammunition.
R11. The player can can open the pause menu during the game if he press the ESCAPE key.
R12. The player can exit the game and return to the main menu by pressing "MENU" button. R11 is required.
R13. The zombies (from the genetic algorithm) will learn to find their way to the brother.
R14. The system will be able to save the genes of the zombies so that every time the user plays, the zombies keep learning and saving that experience.
R15. The zombies (from the flocking algorithm) will be able to avoid obstacles.
R16. Zombies (from the flocking algorithm) will be able to advance together in groups.
R17. Zombies (all of them), if they see the player or the brother, will be able to go and attack them.

3.1.2 Non-functional Requirements

Non-functional requirements are requirements that impose restrictions on design or implementation such as restrictions on design or quality standards. These are properties or qualities that the product must have.
R18. The aesthetics will be cartoon.
R19. The UI must be simple, not obstructing the vision and attractive to the player.
R20. The controls must be comfortable for the player.
R21. The character mechanics must be fluid and its animations attractive to the player.
R22. The zombies will give a group feeling.

3.2 System Design

This section presents the logical and operational design of the system to be carried out. To do the logical design is shown with a use case diagram (see Figure 3.6). And the descriptions of these cases are here:

Requirements:	R4
Actor:	Player
Description:	At the beginning, the player will have the option to select "Play" to start the game which will be on the title screen.
Preconditions:	1. The player is on title screen.
Steps normal sequence:	1. The player selects "Play". 2. The game loads the map. 3. Control is returned to the player.
Alternative sequence:	None.

Table 3.1: Case of use «CU01. Play»

Requirements:	R2
Actor:	Player
Description:	The player can activate or desactivate the sound.
Preconditions:	1. The player must be on the title screen. 2. The player must have entered the options screen.
Steps normal sequence:	1. The player selects to turn the sound on or off. 2. The volume of the game is adjusted to the player's selection.
Alternative sequence:	None.

Table 3.2: Case of use «CU02. Change music»

Requirements:	R1
Actor:	Player
Description:	The player can select the game difficulty between *easy*, *normal* and *hard*.
Preconditions:	1. The player must be on the title screen. 2. The player must have entered the options screen.
Steps normal sequence:	1. The player selects the difficulty. 2. The difficulty of the game is adjusted to the player's selection.
Alternative sequence:	None.

Table 3.3: Case of use «CU03. Select difficulty»

Requirements:	R3
Actor:	Player
Description:	Allows the player to quit the game.
Preconditions:	1. The player must be on the title screen.
Steps normal sequence:	1. The player selects "Quit". 2. The game is closed.
Alternative sequence:	None.

Table 3.4: Case of use «CU04. Quit Game»

Requirements:	R11
Actor:	Player
Description:	Allows the player to exit to the main menu.
Preconditions:	1. The player must be on playing the game.
Steps normal sequence:	1. The player opens the pause menu. 2. The player chooses the option "MENU". 3. The player is taken to the main menu.
Alternative sequence:	None.

Table 3.5: Case of use «CU05. Return to main menu»

Requirements:	R5
Actor:	Player
Description:	Each time the player presses any key of movement he can move around the map.
Preconditions:	1. Having started the game (See CU01 in 3.1)
Steps normal sequence:	1. The player presses the move keys. 2. The game moves the player according to the move he has chosen (left/right/forward/backward).
Alternative sequence:	[1.1] The player will attempt to move in the direction of an obstacle and will not be able to perform the action.

Table 3.6: Case of use «CU06. Move»

Requirements:	R6
Actor:	Player
Description:	The player can perform melee attacks by pressing the knife-attack key.
Preconditions:	1. Having started the game (See CU01 in 3.1)
Steps normal sequence:	1. The player presses the knife-attack key. 2. The player makes the melee attack.
Alternative sequence:	[2.1] If there is an enemy in the attack radius, it damages him.

Table 3.7: Case of use «CU07. Melee attack»

Requirements:	R8
Actor:	Player
Description:	The player can attack from a distance by pressing the shoot button.
Preconditions:	1. Having started the game (See CU01 in 3.1) 2. Have bullets on the gun.
Steps normal sequence:	1. The player presses the shoot button. 2. A bullet is fired from the gun. 3. The bullet travels a distance and disappears.
Alternative sequence:	**1.1** In the case where the player's gun has no bullets, he will not be able to shoot. **3.1** If the bullet hits an enemy before it disappears, it damage the enemy.

Table 3.8: Case of use «CU08. Shoot»

Requirements:	R7
Actor:	Player
Description:	The player can throw a grenade by pressing G key.
Preconditions:	1. Having started the game (See CU01 in 3.1) 2. Have grenades.
Steps normal sequence:	1. The player presses the G key. 2. A grenade is thrown. 3. The grenade travels a distance and explodes.
Alternative sequence:	**1.1** In the case where the player has no grenades, he will not be able to throw them. **3.1** If the grenade's explosion hits an enemy, it damage the enemy.

Table 3.9: Case of use «CU09. Throw a grenade»

Requirements:	R8
Actor:	Player
Description:	The player can reload his gun.
Preconditions:	1. Having started the game (See CU01 in 3.1) 2. Have ammunition. 3. Not having the gun magazine full.
Steps normal sequence:	1. The player presses the reload key. 2. The gun is reloaded.
Alternative sequence:	[1.1] In the case where the player has no ammunition or the gun's magazine is full, the gun will not be reloaded.

Table 3.10: Case of use «CU10. Reload»

Requirements:	R9
Actor:	Player
Description:	The player can take ammunition or first aid kits.
Preconditions:	1. Having started the game (See CU01 in 3.1)
Steps normal sequence:	1. The player is in front of a box with ammunition or first aid kit and press the interacting key. 2. The item is taken.
Alternative sequence:	**2.1** In the case where the box has ammunition, the player will take 20 bullets. **2.2** In the case where the box has a first aid kit, the player will recover automatically 20 health points.

Table 3.11: Case of use «CU11. Take item»

Requirements:	None
Actor:	Player
Description:	At the beginning, the player will have the option to select "Tutorial" and play a tutorial to learn the basic controls.
Preconditions:	1. The player is on title screen.
Steps normal sequence:	1. The player selects "Tutorial". 2. The game loads the tutorial scene. 3. Control is returned to the player.
Alternative sequence:	

Table 3.12: Case of use «CU12. Play Tutorial»

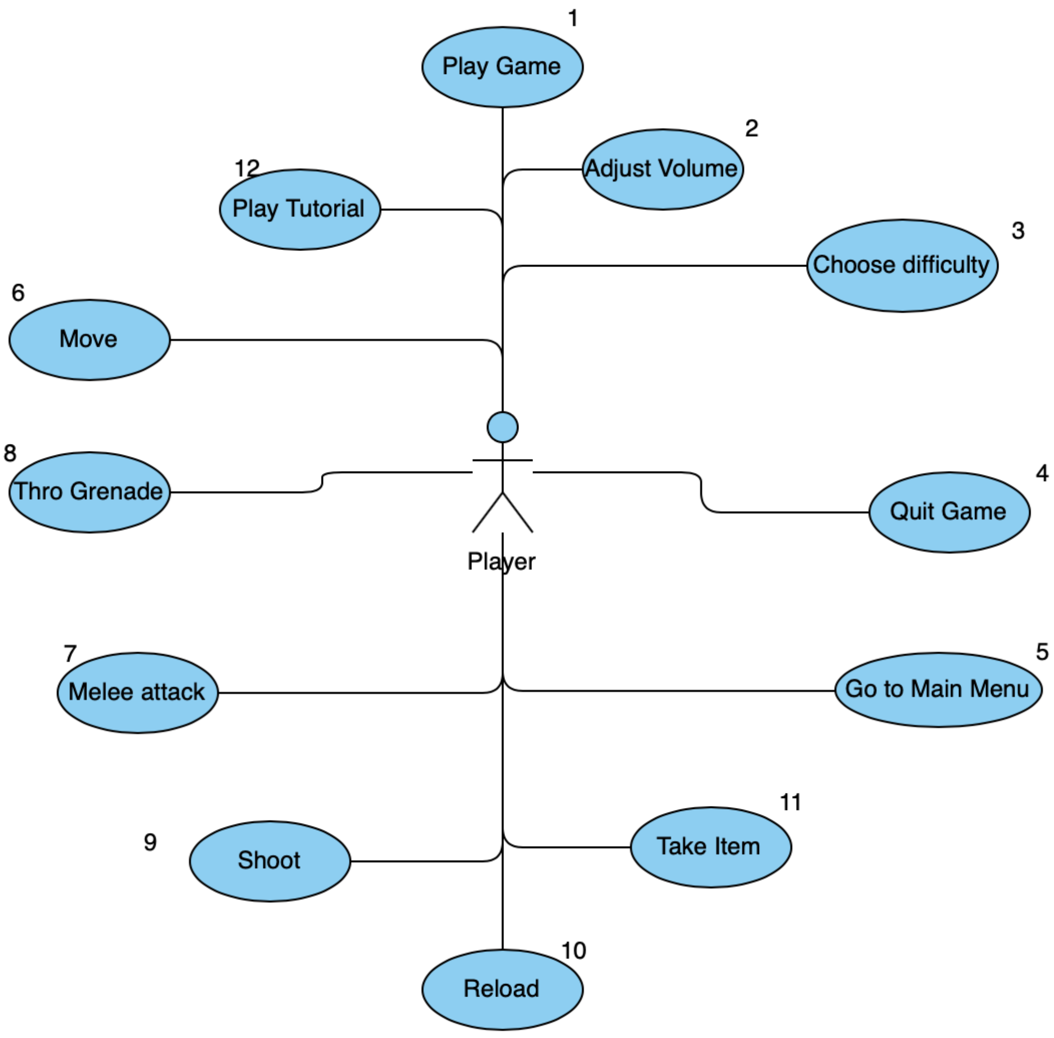


Figure 3.6: Case use diagram (made with https://online.visual-paradigm.com)

3.3 System Architecture

This section describes the arquitecture of the projected system. The video game is made with Unity3D engine, specifically with 2019.2.6f1 version. For running games made with this engine the minimum system requirements are:

- Operating system
  - Windows 7 SP1+
  - macOS 10.12+
  - Ubuntu 16.04+
- CPU: SSE2 instruction set support.
- GPU: Graphics card with DX10(shader model 4.0) capabilities.

This information is taken from Unity's documentation [11], but does not offer a guarantee that it will work on all systems that satisfy those requirements. The game has been developed and tested on a computer with these hardware and software features:

- Operating system: macOs 10.15.3
- Processor: Quead-Core Intel Core i7
- Memory: 16GB
- Graphics: Radeon Pro 555 2GB

It is recommended, for comfort, to play with a mouse instead of a touch panel (if playing on a laptop).

3.4 Interface Design

The game user interface, GUI, is very simple and unobtrusive. It gives the player the necessary information about basic aspects such as the ammunition he has or the life he has left.

He can also see the remaining life of the brother, so he can control whether the zombies have already started attacking him or not. It should also be noted that during the game, text is added to the user interface that tells him what to do. For example, when approaching an ammunition box or an obstacle. To understand this better, we can see some images that show it. (See Figures 3.7, 3.8, 3.9)

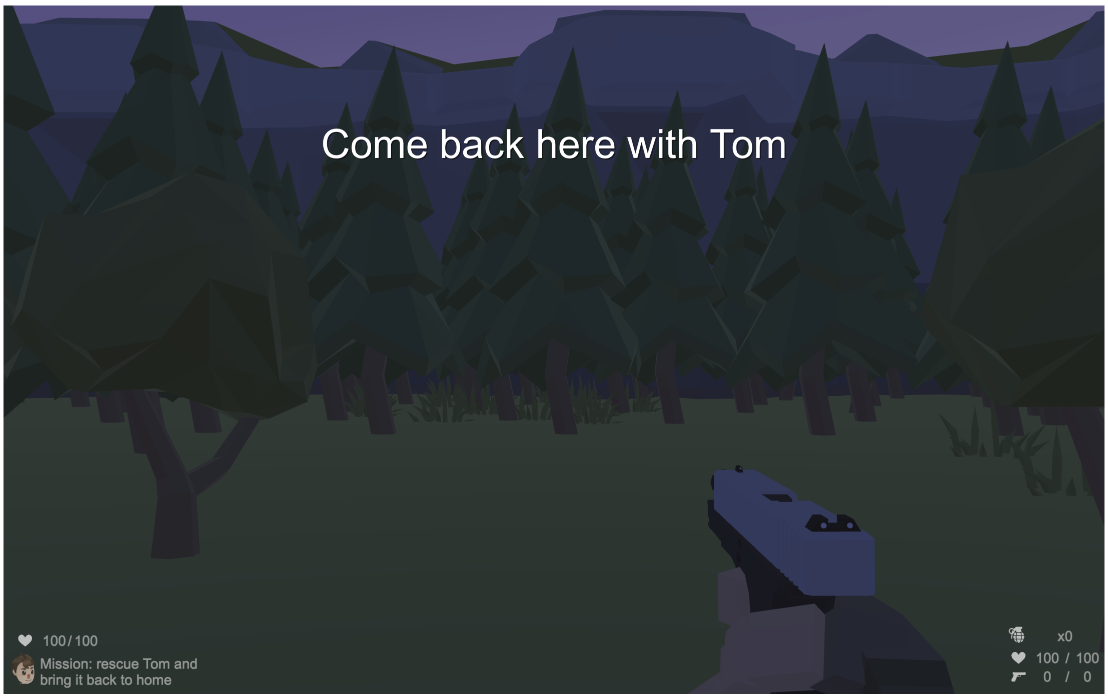


Figure 3.7: At the start of the game (image taken in game)

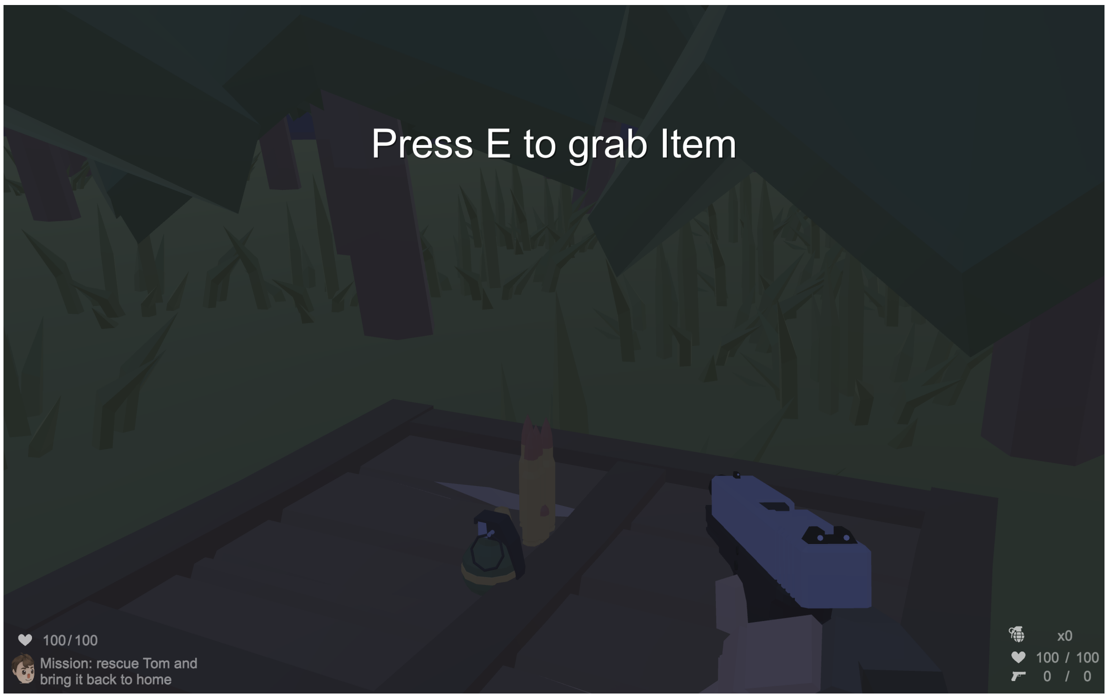


Figure 3.8: When approaching a box (image taken in game)

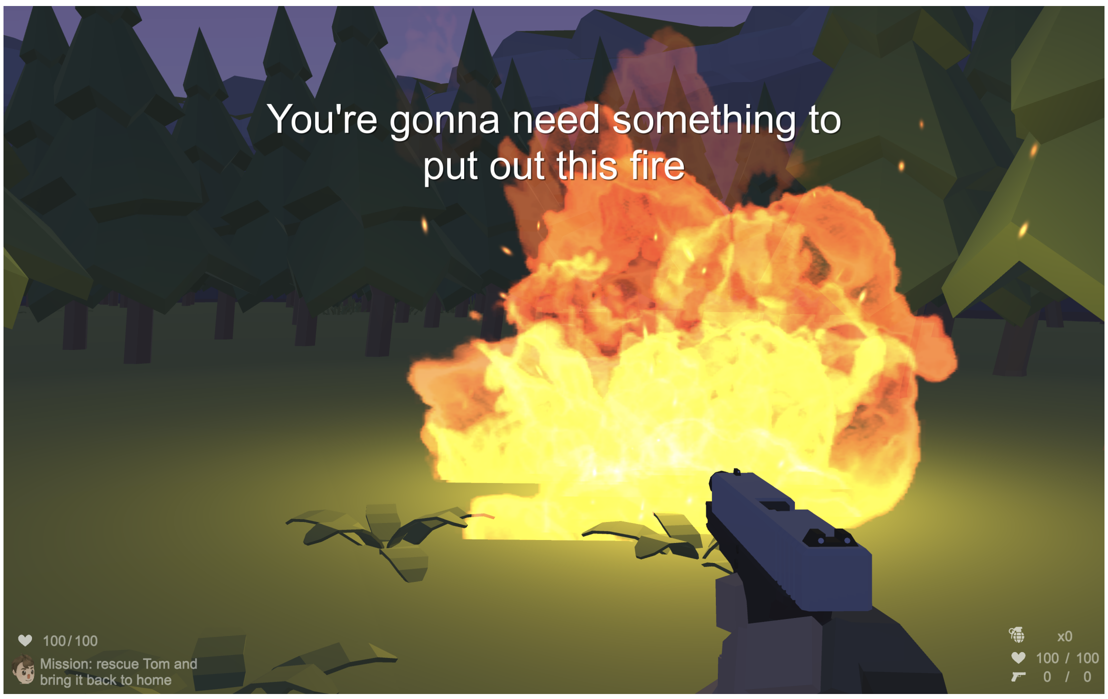

You're gonna need something to put out this fire
100/100
Mission: rescue Tom and bring it back to home
x0
100 / 100
0 / 0

Figure 3.9: When approaching an obstacle (image taken in game)

Chapter 4

Work Development and Results

Contents

After having a solid idea of the work to be done, of the requirements of the video game and of the gameplay style it will have, in this chapter the development of the project will be seen. How it has progressed, the mini goals that have been achieved and the changes that may have been made along the way.

4.1 Work Development

In this section it will be discussed what has been done and the decisions that have been taken at certain times, but only the most relevant aspects of the project will be deepened. Much of the information for the realization of this project comes from, in addition to specific books and articles that will later be cited, tutorials on YouTube. This includes both theory of certain concepts and implementation of certain things in Unity. All this is collected in a youtube playlist [12].

What was most challenging was the genetic algorithm and the pathfinding associated with it. Luis left me a book, Artificial Intelligence for Developers - Concepts and Implementation in C# [13] which contained some examples of genetic algorithms. This was very useful for me since those examples taught how to implement them in C#, a language that Unity uses and, therefore, I was going to use. Of the examples in the book I was guided by the Travelling Salesman Problem, TSP. The TSP is a classic in the world of algorithms: finding the shortest possible route that visits each city only once on a map, returning to the city of origin.

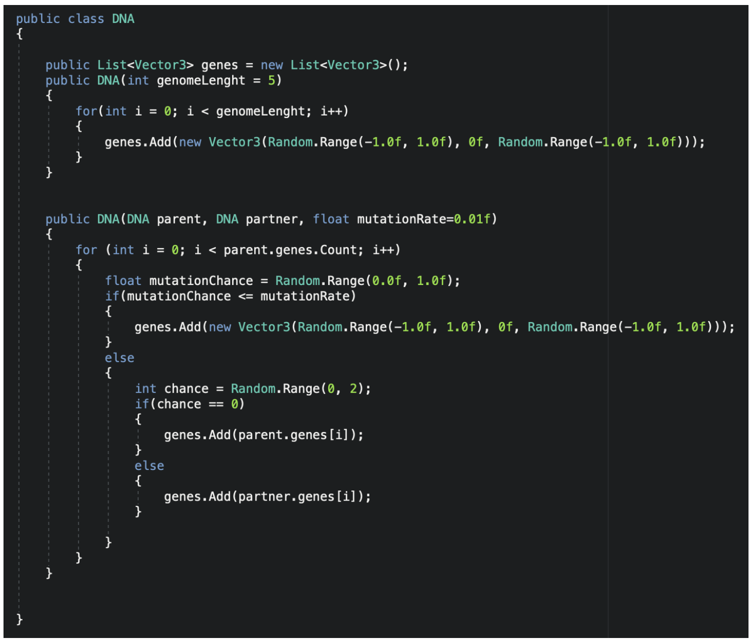
The thing is that by reading that implementation I learned the basic tools to build a genetic algorithm. First of all: DNA. You can't talk about genetic algorithms and not mention DNA, genes, genome, etc. In the case of the TSP, the genome was a list, a permutation of the cities to be covered. For example, being n the number of cities and n = 8, the genome of one of the individuals could be (3 2 4 1 7 5 8 6). In that problem the individuals are possible solutions to the problem, so in this case we had to see our individual, the zombie, as a solution to our problem, looking for the brother (let's call him Tom from now on).

Although I had the theory and knew how I should program the DNA class in C#, I was not sure what we could build the genome on. I looked for an example that might inspire me and found one about a maze [14]. That helped me a little to understand how to shape my code.

What set of components could form our individual? A list of movement vectors could work. A Vector3 list where in each Vector3 the X component and the Z component would oscillate between -1 and 1. It would be like instructions that would indicate the zombie the little steps that he would have to give to find Tom. But let's keep in mind that if the distance he has to travel is large, a list of 5 movement vectors, for example, might not be enough. So the length of the genome is something we'll have to vary to see which works best. So we have in mind what could be the constructor of the DNA class.

We could already think about another constructor corresponding to the gene mutation. That is, given a parent, a partner and a mutation ratio, mix the genes of both. So to create a new individual from those two, for every gene that forms the genome, there will be a probability (defined by the mutation ratio) that that gene is: random, the parent gene or the partner gene.

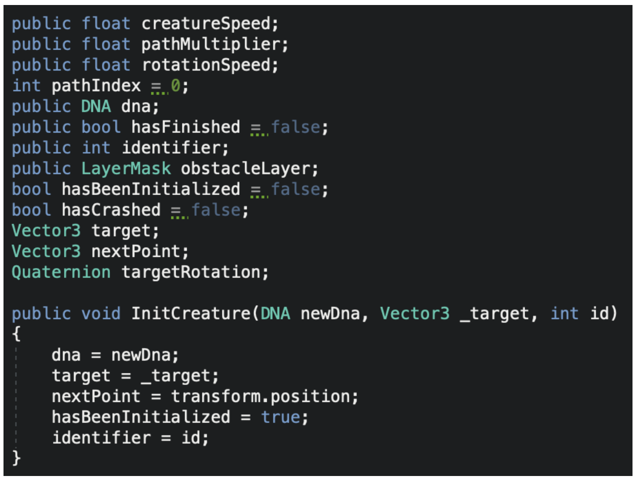
And below we have the result of these two things.



```
public class DNA
{

    public List<Vector3> genes = new List<Vector3>();
    public DNA(int genomeLenght = 5)
    {
        for(int i = 0; i < genomeLenght; i++)
        {
            genes.Add(new Vector3(Random.Range(-1.0f, 1.0f), 0f, Random.Range(-1.0f, 1.0f)));
        }
    }


    public DNA(DNA parent, DNA partner, float mutationRate=0.01f)
    {
        for (int i = 0; i < parent.genes.Count; i++)
        {
            float mutationChance = Random.Range(0.0f, 1.0f);
            if(mutationChance <= mutationRate)
            {
                genes.Add(new Vector3(Random.Range(-1.0f, 1.0f), 0f, Random.Range(-1.0f, 1.0f)));
            }
            else
            {
                int chance = Random.Range(0, 2);
                if(chance == 0)
                {
                    genes.Add(parent.genes[i]);
                }
                else
                {
                    genes.Add(partner.genes[i]);
                }

            }
        }
    }


}
```

Figure 4.1: DNA Class and its constructors.

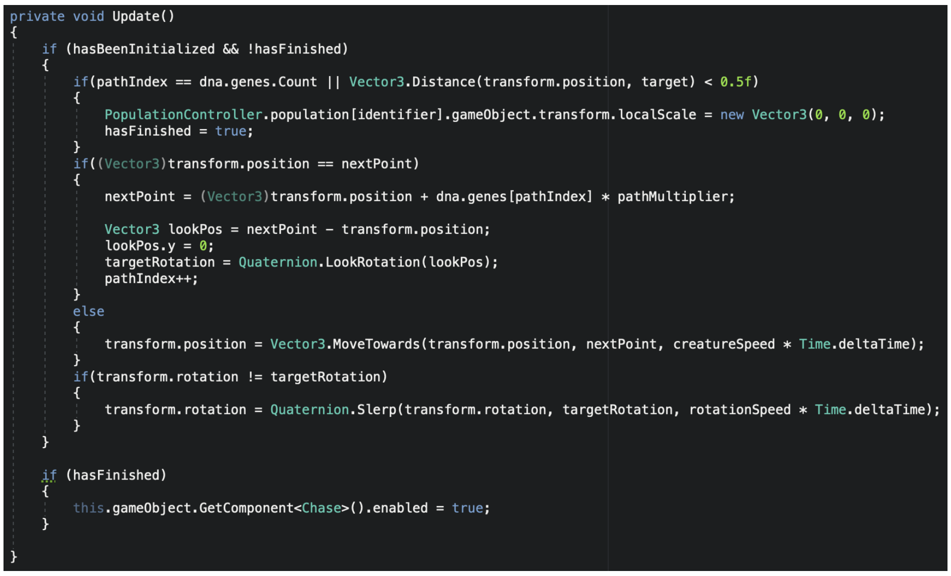
Once the genes have been defined, let's go to the *GeneticPathfinder* class that will control the zombies. First of all we will have a function to initialize the variables and so on.



```
public float creatureSpeed;
public float pathMultiplier;
public float rotationSpeed;
int pathIndex = 0;
public DNA dna;
public bool hasFinished = false;
public int identifier;
public LayerMask obstacleLayer;
bool hasBeenInitialized = false;
bool hasCrashed = false;
Vector3 target;
Vector3 nextPoint;
Quaternion targetRotation;

public void InitCreature(DNA newDna, Vector3 _target, int id)
{
    dna = newDna;
    target = _target;
    nextPoint = transform.position;
    hasBeenInitialized = true;
    identifier = id;
}
```

Figure 4.2: Genetic Pathfinder's variables and *InitCreature()* method.

The *DNA*, *target* and *identifier* will be provided by the *PopulationController* class that we will see later. We'll also have a Vector3, *nextPoint*, that will store the next movement the zombie will make. Of course we'll have a variable that indicates whether the zombie has been initialized so that we can run the rest of the things in the Update.



```
private void Update()
{
    if (hasBeenInitialized && !hasFinished)
    {
        if(pathIndex == dna.genes.Count || Vector3.Distance(transform.position, target) < 0.5f)
        {
            PopulationController.population[identifier].gameObject.transform.localScale = new Vector3(0, 0, 0);
            hasFinished = true;
        }
        if((Vector3)transform.position == nextPoint)
        {
            nextPoint = (Vector3)transform.position + dna.genes[pathIndex] * pathMultiplier;

            Vector3 lookPos = nextPoint - transform.position;
            lookPos.y = 0;
            targetRotation = Quaternion.LookRotation(lookPos);
            pathIndex++;
        }
        else
        {
            transform.position = Vector3.MoveTowards(transform.position, nextPoint, creatureSpeed * Time.deltaTime);
        }
        if(transform.rotation != targetRotation)
        {
            transform.rotation = Quaternion.Slerp(transform.rotation, targetRotation, rotationSpeed * Time.deltaTime);
        }
    }

    if (hasFinished)
    {
        this.gameObject.GetComponent<Chase>().enabled = true;
    }

}
```

Figure 4.3: Genetic Pathfinder's *Update()* method.

As long as the zombie has been initialized and has not finished performing all the "little steps" we talked about before (the movements dictated by its DNA), it will keep moving. There are 3 reasons why the zombie finishes running this script:

- He finishes running his DNA and therefore runs out of instructions to follow.
- He finds the target (the brother), so he can stop running the steps of his DNA.
- He hits some obstacle on the map. If this happens, he can't continue to execute his DNA movements because he would be crossing objects.

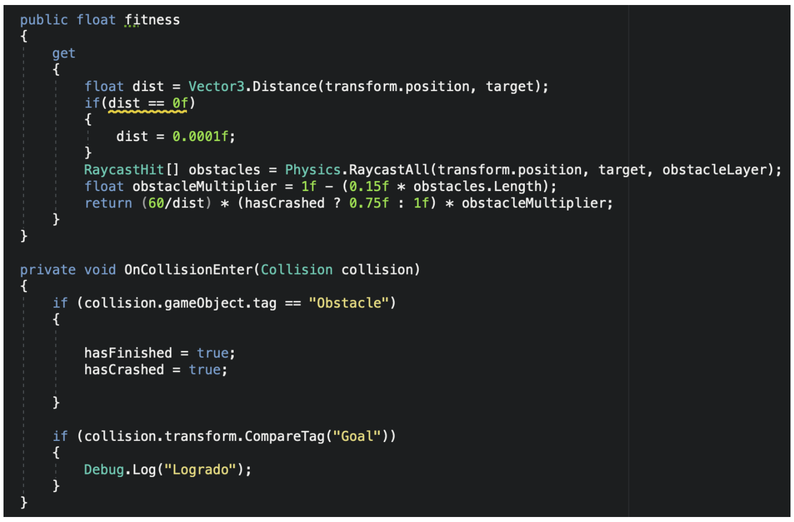
After any of these situations occur, this script is deactivated and another one called *Chase* [1] is activated, which consists of a patrol behavior.

The *PopulationController* class is the one that manages, as its name indicates, the population and therefore the next generations that are created. It uses another method that we have in this *GeneticPathfinder* script, called *fitness()* that returns the "score" of the subject. Fitness function evaluates how close a given solution is to the optimum solution of the desired problem. It determines how fit a solution is. How do we evaluate the fitness function? What parameters do we use?

[1] At the beginning of development, my main purpose was to make this genetic algorithm work, so the procedure was: one generation is created; all individuals finish running their genes; all individuals are destroyed; a new generation is created. And so again and again. After I made it work, it didn't make sense, in relation to gameplay, that zombies destroyed themselves as soon as they were generated, so I added other scripts like Chase.cs and more things. We'll see them later.

The distance from the zombie spawn point and the brother is about 65 units in Unity. So we are going to vary our reward (the value that will return fitness) between 0 and 60. This way, if the distance that separates the zombie from the brother is more than 60, this function will return (we can see that we divide 60 / dist) a number lower than 1.

Let's imagine that the rewards are cookies. Well, since they are zombies they will be *brain cookies*. If a zombie gets very close to his target (the brother) he will get a lot of cookies. On the other hand, if he walks away or stays too far away, he will run out of cookies. But is distance the only thing that matters to us? No. We will also consider two more things. First, whether or not the zombie has hit something. If it has, we'll penalize it with 25 percent fewer cookies. If it hasn't crashed, we don't penalize it in any way.But wait, let's think about this for a moment. If we have two zombies: one stays 20 units away from the brother, without having crashed; and the other stays 10 units away without having crashed either. But this last one has a huge wall in front of him. That means that even if it advanced further, it couldn't get through. And the first one has no obstacle in front of him. As we have assigned the rewards, we would be rewarding the last zombie better, when in fact he has done worse.To solve this, we added an obstacle multiplier. We check, with a raycast, how many obstacles separate the zombie from the brother (in a straight line) and we calculate a multiplier with that data.The multiplier will be 1 (and therefore will not penalize the final reward) if there is no obstacle. The more obstacles there are, the more the multiplier will go down from 1 to -infinity.
And with all this, we would have the reward function ready.As a last note, we have put that if the distance (dist) is equal to 0, change it to 0.0001f because otherwise we would have a 60/0 operation, which would give us problems.

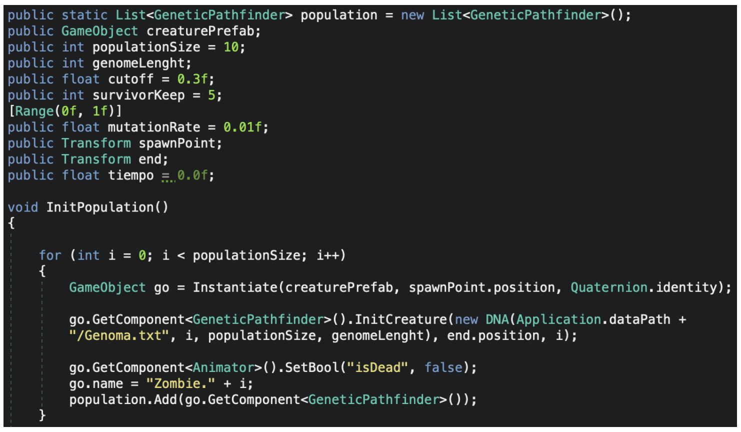


```
public float fitness
{
    get
    {
        float dist = Vector3.Distance(transform.position, target);
        if(dist == 0f)
        {
            dist = 0.0001f;
        }
        RaycastHit[] obstacles = Physics.RaycastAll(transform.position, target, obstacleLayer);
        float obstacleMultiplier = 1f - (0.15f * obstacles.Length);
        return (60/dist) * (hasCrashed ? 0.75f : 1f) * obstacleMultiplier;
    }
}

private void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.tag == "Obstacle")
    {

        hasFinished = true;
        hasCrashed = true;

    }

    if (collision.transform.CompareTag("Goal"))
    {
        Debug.Log("Logrado");
    }
}
```

Figure 4.4: Genetic Pathfinder's *fitness()* method.

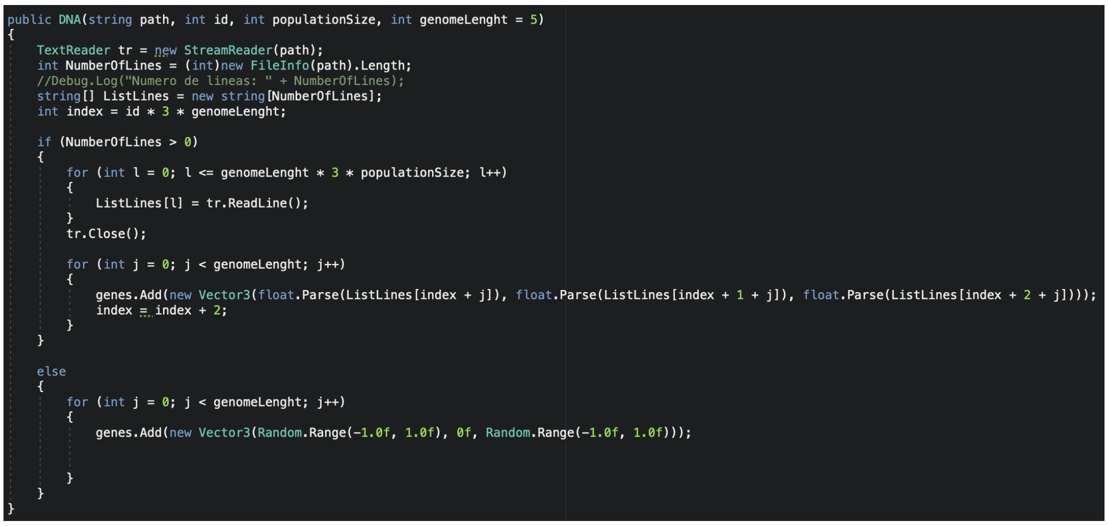
Now let's look at the *PopulationController*. Many of the variables can be modified from the inspector so that the most successful combinations can be tested. In *Init-*

Population() we instantiate n zombies being n the size of the population that we have established. And through the *GeneticPathfinder* script we execute the *InitCreature()*[2] to initialize them.



```
public static List<GeneticPathfinder> population = new List<GeneticPathfinder>();
public GameObject creaturePrefab;
public int populationSize = 10;
public int genomeLenght;
public float cutoff = 0.3f;
public int survivorKeep = 5;
[Range(0f, 1f)]
public float mutationRate = 0.01f;
public Transform spawnPoint;
public Transform end;
public float tiempo = 0.0f;

void InitPopulation()
{
    for (int i = 0; i < populationSize; i++)
    {
        GameObject go = Instantiate(creaturePrefab, spawnPoint.position, Quaternion.identity);

        go.GetComponent<GeneticPathfinder>().InitCreature(new DNA(Application.dataPath +
        "/Genoma.txt", i, populationSize, genomeLenght), end.position, i);

        go.GetComponent<Animator>().SetBool("isDead", false);
        go.name = "Zombie." + i;
        population.Add(go.GetComponent<GeneticPathfinder>());
    }
```

Figure 4.5: PopulationController's variables and *InitPopulation()* method.

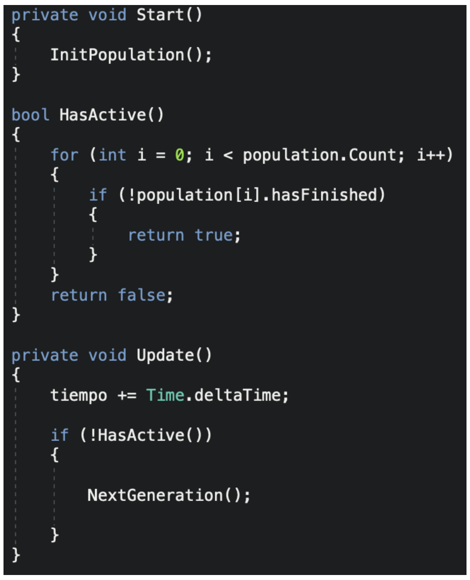


```
public DNA(string path, int id, int populationSize, int genomeLenght = 5)
{
    TextReader tr = new StreamReader(path);
    int NumberOfLines = (int)new FileInfo(path).Length;
    //Debug.Log("Numero de lineas: " + NumberOfLines);
    string[] ListLines = new string[NumberOfLines];
    int index = id * 3 * genomeLenght;

    if (NumberOfLines > 0)
    {
        for (int l = 0; l <= genomeLenght * 3 * populationSize; l++)
        {
            ListLines[l] = tr.ReadLine();
        }
        tr.Close();

        for (int j = 0; j < genomeLenght; j++)
        {
            genes.Add(new Vector3(float.Parse(ListLines[index + j]), float.Parse(ListLines[index + 1 + j]), float.Parse(ListLines[index + 2 + j])));
            index = index + 2;
        }
    }

    else
    {
        for (int j = 0; j < genomeLenght; j++)
        {
            genes.Add(new Vector3(Random.Range(-1.0f, 1.0f), 0f, Random.Range(-1.0f, 1.0f)));


        }
    }
}
```

Figure 4.6: DNA constructor by taking saved information.

[2]We can see that here the DNA constructor is different from the ones I have shown before, in this constructor we take the DNA from a text file, *Genes.txt*, where we have saved some trained genes. If this file is empty, it creates the DNA randomly as we have seen in the previous constructors. This helps us to pre-train our zombies and save an acceptable result, so that when the game starts the zombies already walk a more or less precise path. This means that they start from a solid base. (see Figure 4.6)

We have a method called *HasActive()* that checks whether all individuals in the population have finished or some are still active. So if all have finished[3], in the *Update()* function we run *NextGeneration()* to create the new generation of individuals.

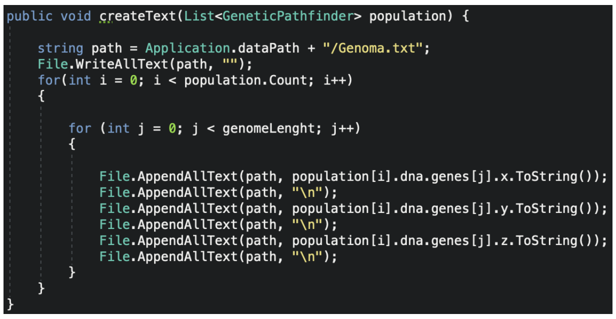


```
private void Start()
{
    InitPopulation();
}

bool HasActive()
{
    for (int i = 0; i < population.Count; i++)
    {
        if (!population[i].hasFinished)
        {
            return true;
        }
    }
    return false;
}

private void Update()
{
    tiempo += Time.deltaTime;

    if (!HasActive())
    {

        NextGeneration();

    }
}
```

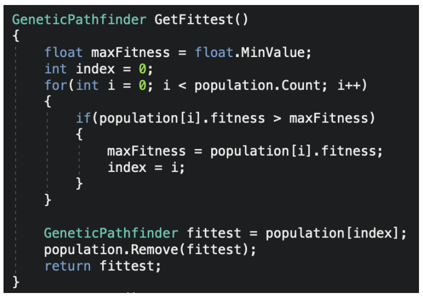
Figure 4.7: PopulationController's *Start()*, *HasActive()* and *Update()* methods.

Before we move on to *NextGeneration()* we're going to look at two more methods. *GetFittest()* and *createText()*. This last one what it does is to save the information (the genes) of the last population created in the text file that I have commented before. While *GetFittest()* is used to take the genes of the individual with the best score achieved in this last generation. We can't forget to delete it from the list, because if we call more than once to the function to get for example the 3 best individuals and we don't delete them, it will always return the same set of genes.

[3]That code shows how the creations of generations worked while I was developing the algorithm. When I got the genetic algorithm working, I added another condition for NextGeneration() to run. You can see it in the final code in Appendix A. But basically, apart from all the zombies having finished running their genes, there is a time restriction. After 180, 280, 350 and 500 seconds NextGeneration() can be executed. There will be a maximum of 5 generations in total during the game.



```
public void createText(List<GeneticPathfinder> population) {

    string path = Application.dataPath + "/Genoma.txt";
    File.WriteAllText(path, "");
    for(int i = 0; i < population.Count; i++)
    {

        for (int j = 0; j < genomeLenght; j++)
        {

            File.AppendAllText(path, population[i].dna.genes[j].x.ToString());
            File.AppendAllText(path, "\n");
            File.AppendAllText(path, population[i].dna.genes[j].y.ToString());
            File.AppendAllText(path, "\n");
            File.AppendAllText(path, population[i].dna.genes[j].z.ToString());
            File.AppendAllText(path, "\n");
        }
    }
}
```



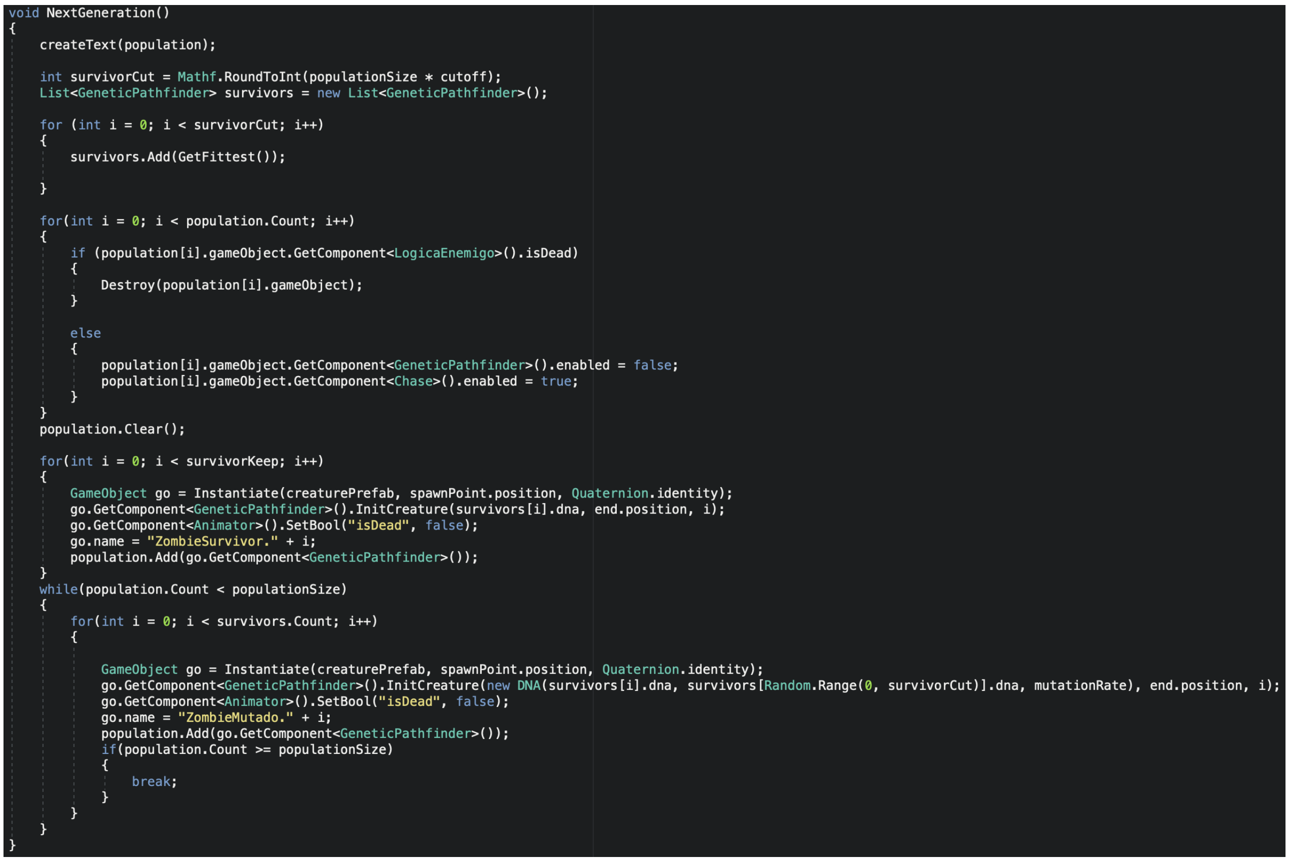
```
GeneticPathfinder GetFittest()
{
    float maxFitness = float.MinValue;
    int index = 0;
    for(int i = 0; i < population.Count; i++)
    {
        if(population[i].fitness > maxFitness)
        {
            maxFitness = population[i].fitness;
            index = i;
        }
    }

    GeneticPathfinder fittest = population[index];
    population.Remove(fittest);
    return fittest;
}
```

Figure 4.8: PopulationController's *createText()* and *GetFittest()* methods.

In *NextGeneration()* what we do is with the variable *cutoff* get the number of survivors we want to keep and put them in a list. Before we continue, there is one important thing to explain. When the player kills a zombie that is running the *GeneticPathfinder* script, we don't destroy his *gameObject*. Why? Because we will need his data, his "score", so that later in the new generation we can use his genes if he enters the list of survivors. So, knowing this, now in *NextGeneration()* we destroy the *gameObject* of all the individuals who have died. If they haven't died we activate the *Chase script* and deactivate the *GeneticPathfinder* script[4]. We also clear the *population* list.

Then we add to the list the survivors of the previous population and we instantiate them. We can modify the variable *survivorKeep* in case we want to keep more survivors or less survivors. We fill the remaining population with mutated individuals, using the constructor we saw for it.

[4]In principle these two modifications should have been done already from a script called *LogicaEnemigo()* that controls the zombie life issue and some other things. But just in case something happens to malfunction, here we make sure to turn those scripts on and off.



```
void NextGeneration()
{
    createText(population);

    int survivorCut = Mathf.RoundToInt(populationSize * cutoff);
    List<GeneticPathfinder> survivors = new List<GeneticPathfinder>();

    for (int i = 0; i < survivorCut; i++)
    {
        survivors.Add(GetFittest());

    }

    for(int i = 0; i < population.Count; i++)
    {
        if (population[i].gameObject.GetComponent<LogicaEnemigo>().isDead)
        {
            Destroy(population[i].gameObject);
        }

        else
        {
            population[i].gameObject.GetComponent<GeneticPathfinder>().enabled = false;
            population[i].gameObject.GetComponent<Chase>().enabled = true;
        }
    }
    population.Clear();

    for(int i = 0; i < survivorKeep; i++)
    {
        GameObject go = Instantiate(creaturePrefab, spawnPoint.position, Quaternion.identity);
        go.GetComponent<GeneticPathfinder>().InitCreature(survivors[i].dna, end.position, i);
        go.GetComponent<Animator>().SetBool("isDead", false);
        go.name = "ZombieSurvivor." + i;
        population.Add(go.GetComponent<GeneticPathfinder>());
    }
    while(population.Count < populationSize)
    {
        for(int i = 0; i < survivors.Count; i++)
        {

            GameObject go = Instantiate(creaturePrefab, spawnPoint.position, Quaternion.identity);
            go.GetComponent<GeneticPathfinder>().InitCreature(new DNA(survivors[i].dna, survivors[Random.Range(0, survivorCut)].dna, mutationRate), end.position, i);
            go.GetComponent<Animator>().SetBool("isDead", false);
            go.name = "ZombieMutado." + i;
            population.Add(go.GetComponent<GeneticPathfinder>());
            if(population.Count >= populationSize)
            {
                break;
            }
        }
    }
}
```

Figure 4.9: PopulationController's *NextGeneration()* method.

Well, what I have told you here in a few minutes took me several weeks to think about and apply correctly. But after those weeks of frustration I got what I just showed. And it works? That's a good question. Yes, I made a scene in unity, I created a prefab (it wasn't even a Zombie, was a capsule), I put a cube (the target) and several walls to serve as an obstacle. I put a population size of 200 individuals and *creatureSpeed* variable, the speed of the individuals, at 100. After about 20 generations, most of the individuals reached the target.

So, my first mini-goal achieved! I already had some sort of genetic/pathfinding algorithm working.

So the next step was to find models for the game. I found a unity package of an FPS [15] with an animated look. I already had what would be the player: the model of the arms, the gun, the camera, etc. I modified a bit the scripts it brought and I already had the player in motion and totally controllable. All that was missing was a suitable environment, a stage.

I found in the unity asset store low poly nature models [16] [17] with cartoon look, ideal for the perception I had in mind. I designed a kind of labyrinthine forest and, after finding models that visually appealed to me and fit the look I was looking for, I started to assemble everything. After having the whole map of the game assembled, I looked for some zombie model that would fit the aesthetics that the game was starting to have. And after finding it [18], I started to add the scripts that I had made of the genetic pathfinding.

I already had the scene set up, the player could move through it and the zombies (see Figure 4.10) were able to find the target (which was still a cube). Until that moment in the zombie prefab I only had the *DNA*, *GeneticPathfinder* and *PopulationController* scripts. So when I started the game I realized two things. The first, that the player had no ammunition. Obviously I could put it on from the inspector, but as far as the context of the game is concerned, I hadn't created the ammunition boxes that were going to be scattered all over the map. The other one was that the zombies, after having finished running their entire genome, didn't do anything else. They would just stand there. So I just had the next two things to work on.

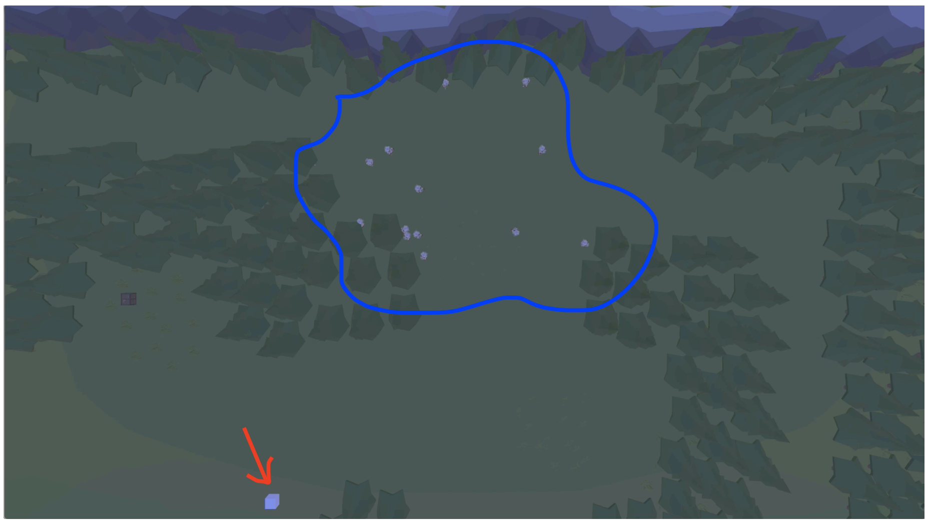


Figure 4.10: What's inside the blue line is the zombies working with the genetic algorithm, while what the red arrow points to is the target, which is still a cube.

I started with the ammunition and with the first aid kits [19] that should also be around the map. Based on the design (see Figure 4.11) I made for the distribution of these objects, I looked for a couple of assets [20], programmed some simple scripts and had both objects working. So what I had to do now was to add that the zombies, after finishing their genome execution, would not stay still. I programmed a script, *Chase.cs*, where the zombie is walking around randomly and if it sees the player goes to attack him. I don't consider this script relevant, as it's something simple, so I'm not going to explain it.

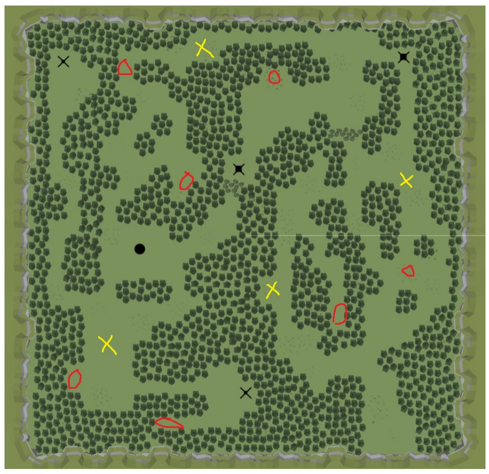


Figure 4.11: Map Design and objects distribution. Yellow crosses are first aid kits and red circles ammunition packs.

Let's go over the things I had already completed: a player who could move around the map, who could collect ammunition and fire the weapon; zombies with a DNA capable of guiding them towards their target and able to enter a patrol mode after finishing their genome. Yet there was still no interaction between the player and the zombies. I needed a controller for their life and that they could damage each other. For this I programmed 3 scripts: *Vida.cs*, *LogicaEnemigo.cs* and *LogicaJugador.cs*. The *Vida.cs* script is basically to define the initial health of the individual and a couple of functions to subtract or add health. The other two scripts changed as other NPCs were developed so we will see them later to be able to understand them completely.

I decided to put the UI [21] in order to have already on screen the data of the ammunition and the health. And since I was designing that I started designing the main menu as well. So, to the things we already had finished we could add the possibility to kill the zombies, the possibility to die if the zombies attacked us and a working main menu.

Then I realized that there was one problem that hadn't been considered. Zombies were able to go on for generation after generation to learn the way and find the brother. This is a static environment, a thing that was extremely important for the algorithm to work. But what happens when the player goes into this equation? Let's take an example where there are two zombies, A and B. Zombie A, because of his genes, ends up getting very close to the target (the brother). Zombie B, on the other hand, seems to be going in the right direction at first, but ends up deviating after a while. If the player kills Zombie A at the beginning of his generation, when evaluating the fitness function of both, it will be concluded that Zombie B is better (because he will have gone further than A). But this is not correct, is it? We can say that the interaction of the player with these zombies totally alters the system and turns it into a dynamic system. All the work thrown away? Well, not really. All that was needed was to redesign the concept of the enemy that would work with this algorithm.

Then a new idea emerged, a new kind of enemy (see Figure 4.12). A Business-Zombie [22] capable of summoning flies [23], so that the flies would go in search of some prey. This way, the flies would work with the genetic algorithm. This is an improvement for several reasons. One of them is that the flies don't interact with the player, which allows them to stay in a static system. The number of flies in each generation can be very high, as they do not make the gameplay difficult and do not bother the screen because they are small. And finally, the speed of the flies can be very high, so approximately every second there is a new generation. This allows around 400 generations to be generated during the course of the game, and therefore, more probability that some fly will end up finding the target. Its operation would be based on what has just been explained but with some small changes in the scripts. The most relevant ones[5]:

[5]Appendix A shows the scripts of the final version of the game, so you can see the changes I mention now.

About DNA.cs

- The DNA constructor from data stored in a text file is no longer required.

About GeneticPathfinder.cs

- Once it has finished running its genome or it has crashed, no further scripts are activated (as was done before, when *Chase.cs* was activated to get the zombies into patrol mode). In the end the *Chase.cs* script is not needed in the whole project.
- If a fly finds the target it starts following him and is always close to him.

About PopulationController.cs

- Now this script is attached to the Business-Zombie that invokes the flies.
- In each new generation the flies of the previous generation are eliminated.
- When a fly finds the target, the Business-Zombie is given the information of the target's position so that it can go and attack him. This is done with a navMesh. All flies are destroyed except de "winner".

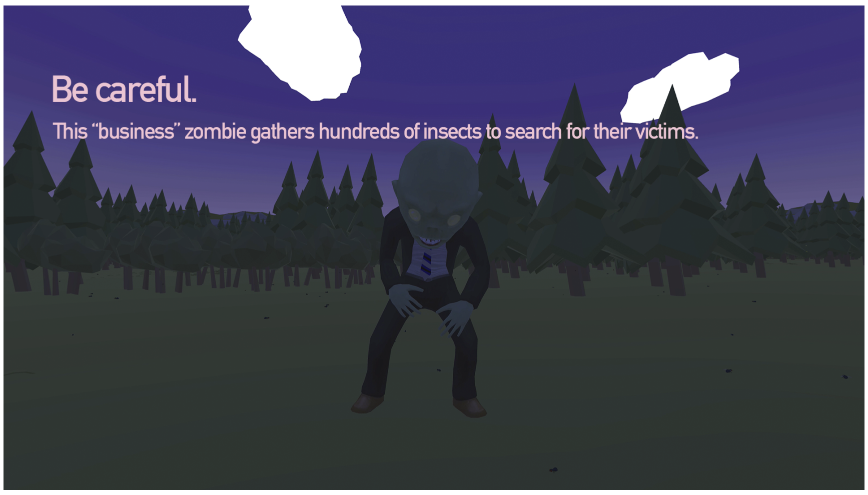
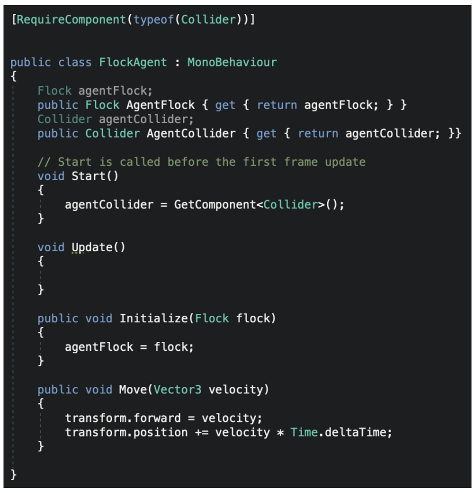

Be careful.
This "business" zombie gathers hundreds of insects to search for their victims.

Figure 4.12: An image that comes out while the levels are loading, that shows the new type of zombie.

After all that, it was time to create the other types of zombies. I needed zombies to be there with the goal of killing the player. Otherwise, during the first few minutes of the game, since the zombie who invokes the flies is still without attacking anyone, the player didn't have any threat. So it was time to learn something about emergent behaviors. I found that there were a few algorithms based on different swarming behaviors. Two of them caught my attention. The first, which I finally ruled out, was Antz colony optimization. I found it quite interesting, it is a probabilistic technique to solve computer problems that can be reduced to looking for the best paths or routes in graphs. I could have used it even for the other zombie behavior, the one of looking for the player's brother. But I wanted something more basic, I didn't necessarily want a search for paths to the player. Imagining what real life could be, even though we haven't had any zombie apocalypse yet (although with how this 2020 is going you never know), I wanted a group of individuals to go together and walk around the map and if anyone of them saw the player then they would go for him. The second algorithm that caught my attention was the flocking algorithm. It sounded to me like something I had seen in the A.I. subject (so I took advantage and relied on the book that this subject used as a reference, Artificial Intelligence For Games [24]). Based on three simple rules: separation, alignment and cohesion, I could achieve behavior that resembled what I was looking for. Daniel Shiffman's example/article [25] also helped me understand how these three rules worked.

Let's start by looking at how we applied the flocking algorithm to our game. The first thing we have is 3 main scripts, *Flock.cs*, *FlockAgent.cs* and *FlockBehavior.cs*. We're going to start with *FlockAgent.cs* which handles each agent individually. The remarkable thing about this script is the initialization function, which assigns it a *Flock*, and the *Move()* function that allows the agent to move given a *Vector3*.

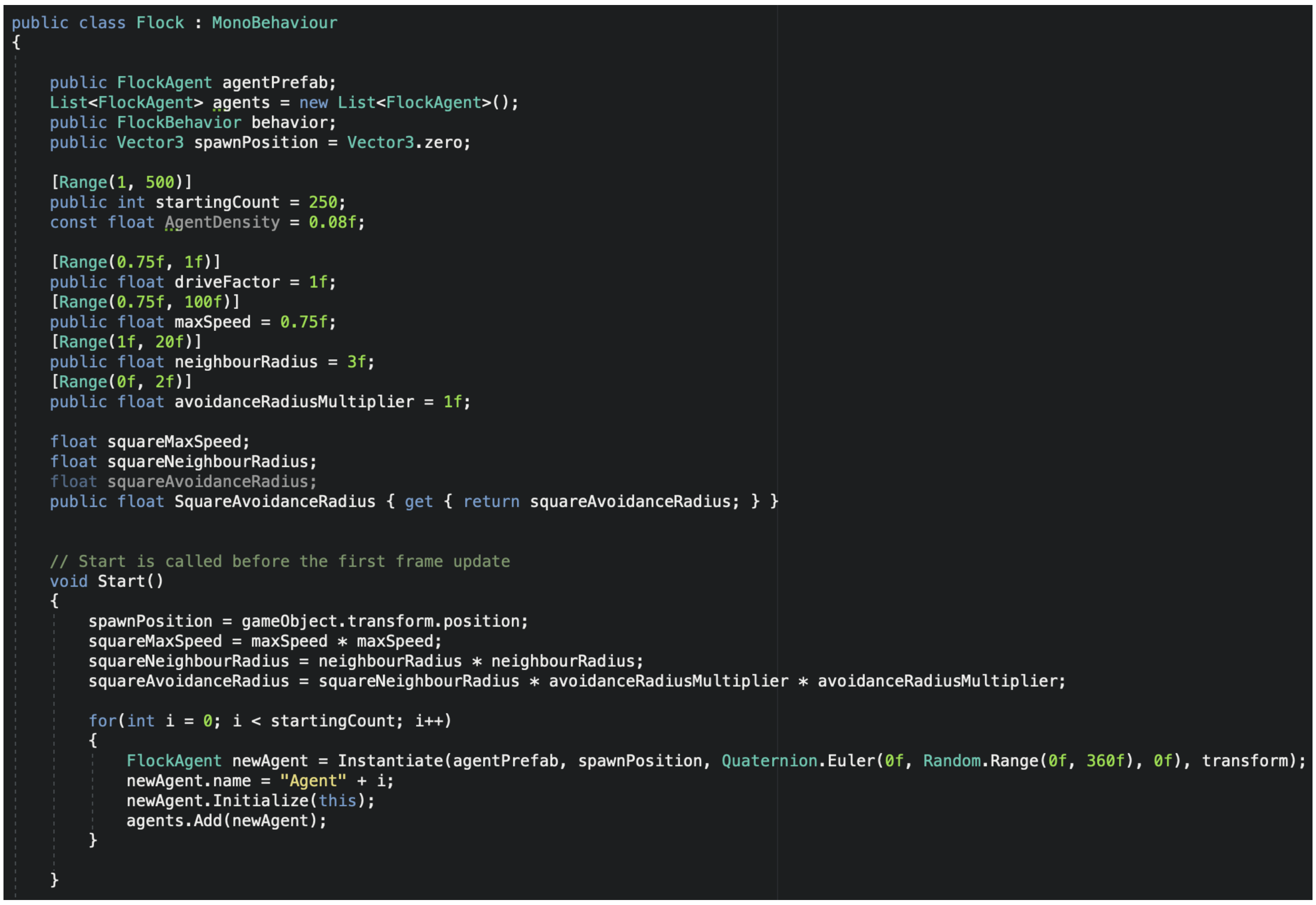


```
[RequireComponent(typeof(Collider))]


public class FlockAgent : MonoBehaviour
{
    Flock agentFlock;
    public Flock AgentFlock { get { return agentFlock; } }
    Collider agentCollider;
    public Collider AgentCollider { get { return agentCollider; }}

    // Start is called before the first frame update
    void Start()
    {
        agentCollider = GetComponent<Collider>();
    }

    void Update()
    {

    }

    public void Initialize(Flock flock)
    {
        agentFlock = flock;
    }

    public void Move(Vector3 velocity)
    {
        transform.forward = velocity;
        transform.position += velocity * Time.deltaTime;
    }

}
```

Figure 4.13: FlockAgent.cs

With this seen, we move on to *Flock.cs*. It basically handles all the agents based on the behaviors we assign to them. It's going to be similar to the *PopulationController* in the sense that when we apply it to a *gameObject* in the inspector, we'll be able to adjust some variables like the number of agents (zombies), the speed, etc. Let's see the declaration of variables and the *Start()* function.

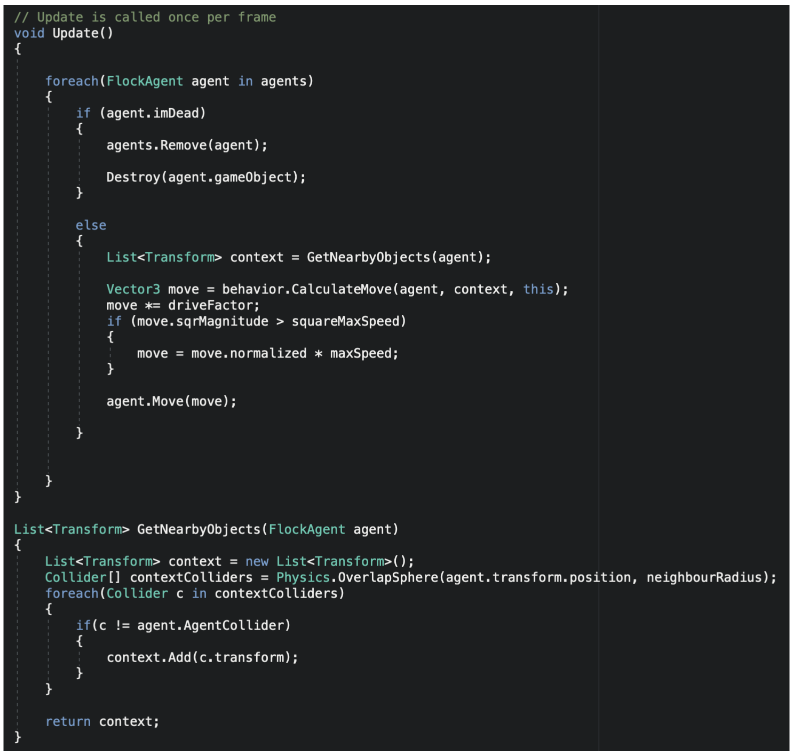
We define several parameters such as the position of the spawn, the speed of the agents, the radius that encompasses the neighboring agents, etc. In the *Start()* what we do is initialize some variables and instantiate the agents from a prefab.



```
public class Flock : MonoBehaviour
{

    public FlockAgent agentPrefab;
    List<FlockAgent> agents = new List<FlockAgent>();
    public FlockBehavior behavior;
    public Vector3 spawnPosition = Vector3.zero;

    [Range(1, 500)]
    public int startingCount = 250;
    const float AgentDensity = 0.08f;

    [Range(0.75f, 1f)]
    public float driveFactor = 1f;
    [Range(0.75f, 100f)]
    public float maxSpeed = 0.75f;
    [Range(1f, 20f)]
    public float neighbourRadius = 3f;
    [Range(0f, 2f)]
    public float avoidanceRadiusMultiplier = 1f;

    float squareMaxSpeed;
    float squareNeighbourRadius;
    float squareAvoidanceRadius;
    public float SquareAvoidanceRadius { get { return squareAvoidanceRadius; } }


    // Start is called before the first frame update
    void Start()
    {
        spawnPosition = gameObject.transform.position;
        squareMaxSpeed = maxSpeed * maxSpeed;
        squareNeighbourRadius = neighbourRadius * neighbourRadius;
        squareAvoidanceRadius = squareNeighbourRadius * avoidanceRadiusMultiplier * avoidanceRadiusMultiplier;

        for(int i = 0; i < startingCount; i++)
        {
            FlockAgent newAgent = Instantiate(agentPrefab, spawnPosition, Quaternion.Euler(0f, Random.Range(0f, 360f), 0f), transform);
            newAgent.name = "Agent" + i;
            newAgent.Initialize(this);
            agents.Add(newAgent);
        }

    }
```

Figure 4.14: Flock.cs

In the *Update()*, for each agent that we have created, first we see if it has died to destroy its *gameObject*, and if it has not, we get with the *GetNearbyObjects* function its context. These are the agents that are close to it and that we consider as neighbours. Then we calculate, through behavior (now we'll see *FlockBehavior.cs*) the movement that corresponds to that agent. And we end up calling the *Move()* function that this agent has in its *FlockAgent* script.



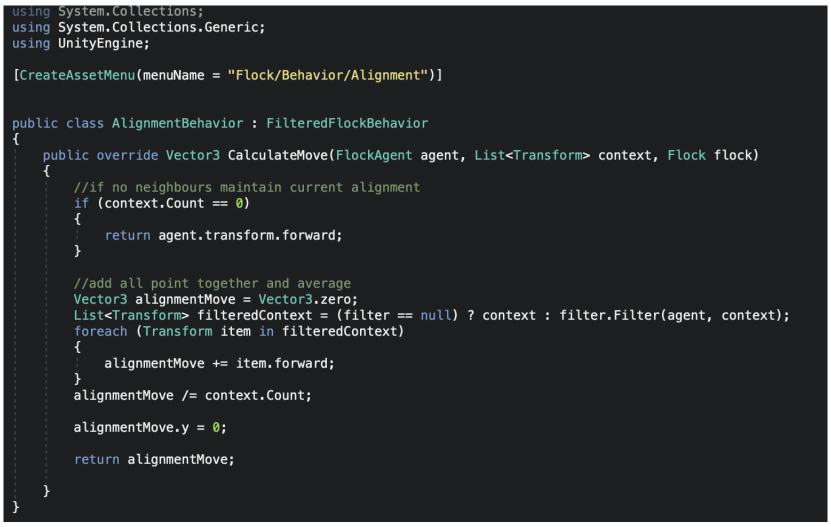
```
// Update is called once per frame
void Update()
{

    foreach(FlockAgent agent in agents)
    {
        if (agent.imDead)
        {
            agents.Remove(agent);

            Destroy(agent.gameObject);
        }

        else
        {
            List<Transform> context = GetNearbyObjects(agent);

            Vector3 move = behavior.CalculateMove(agent, context, this);
            move *= driveFactor;
            if (move.sqrMagnitude > squareMaxSpeed)
            {
                move = move.normalized * maxSpeed;
            }

            agent.Move(move);

        }


    }
}

List<Transform> GetNearbyObjects(FlockAgent agent)
{
    List<Transform> context = new List<Transform>();
    Collider[] contextColliders = Physics.OverlapSphere(agent.transform.position, neighbourRadius);
    foreach(Collider c in contextColliders)
    {
        if(c != agent.AgentCollider)
        {
            context.Add(c.transform);
        }
    }

    return context;
}
```

Figure 4.15: Flock.cs

Perhaps we'll get to the most complicated part of explaining and understanding. This is because the part of the behaviors, their scripts and so on, started as something very basic but I added more things. So there are many scripts and many references between them. So, I'm going to explain above how these behaviors work but without explaining step by step all the scripts that compose it. I will rely on some code, but not on everything. Even so, all these scripts will be in Appendix A if you want to go deeper into the code.

We have 3 behaviors as we said a while ago. They all have scripts that define the movement they'll return.

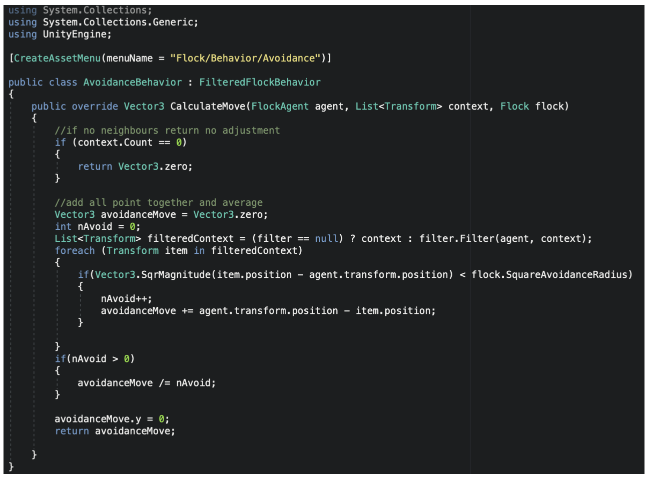
Let's talk about the alignment first. We check where our neighbours are looking and we add up all those positions (which are *Vector3* type). Then we divide that sum by the number of neighbours we have, so we're calculating as an "average" of where to look. And then we return that alignment movement.



```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu(menuName = "Flock/Behavior/Alignment")]


public class AlignmentBehavior : FilteredFlockBehavior
{
    public override Vector3 CalculateMove(FlockAgent agent, List<Transform> context, Flock flock)
    {
        //if no neighbours maintain current alignment
        if (context.Count == 0)
        {
            return agent.transform.forward;
        }

        //add all point together and average
        Vector3 alignmentMove = Vector3.zero;
        List<Transform> filteredContext = (filter == null) ? context : filter.Filter(agent, context);
        foreach (Transform item in filteredContext)
        {
            alignmentMove += item.forward;
        }
        alignmentMove /= context.Count;

        alignmentMove.y = 0;

        return alignmentMove;

    }
}
```

Figure 4.16: Alignment.cs

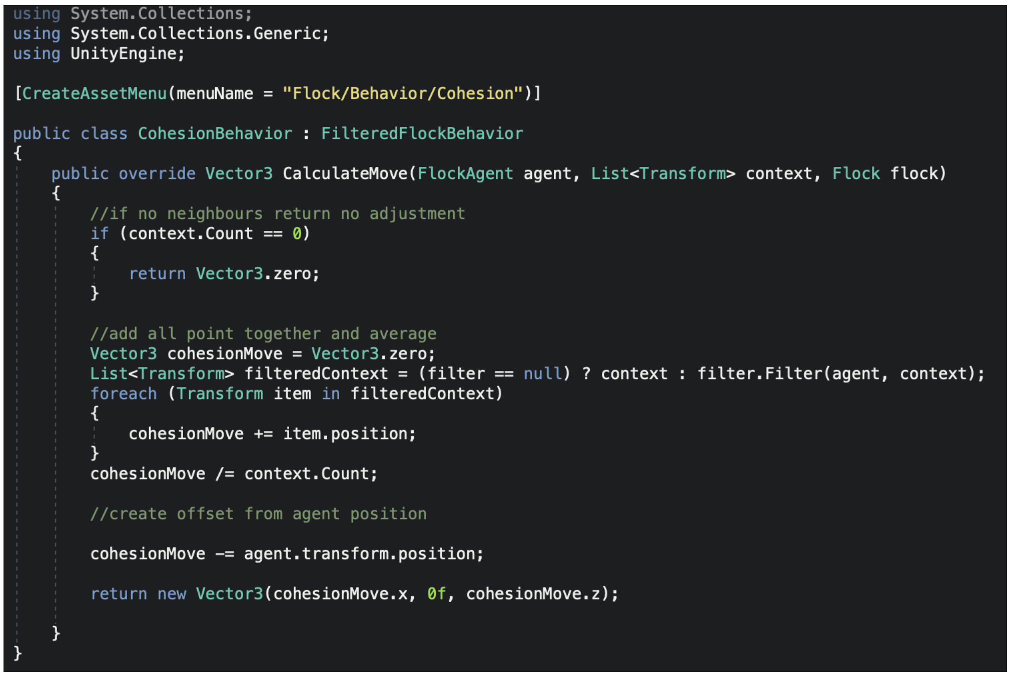
As for the separation, it's similar. For every neighboring agent we have if the distance that separates us from it is less than the one we have defined in *Flock.cs*, *squareAvoidanceRadius*, we add that distance to a variable: *avoidanceMove*. We also are adding 1 to a variable called nAvoid to have the account of the neighbor agents that fulfill this. Then we divide this sum of distances between nAvoid and we obtain the separation movement vector.



```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu(menuName = "Flock/Behavior/Avoidance")]

public class AvoidanceBehavior : FilteredFlockBehavior
{
    public override Vector3 CalculateMove(FlockAgent agent, List<Transform> context, Flock flock)
    {
        //if no neighbours return no adjustment
        if (context.Count == 0)
        {
            return Vector3.zero;
        }

        //add all point together and average
        Vector3 avoidanceMove = Vector3.zero;
        int nAvoid = 0;
        List<Transform> filteredContext = (filter == null) ? context : filter.Filter(agent, context);
        foreach (Transform item in filteredContext)
        {
            if(Vector3.SqrMagnitude(item.position - agent.transform.position) < flock.SquareAvoidanceRadius)
            {
                nAvoid++;
                avoidanceMove += agent.transform.position - item.position;
            }

        }
        if(nAvoid > 0)
        {
            avoidanceMove /= nAvoid;
        }

        avoidanceMove.y = 0;
        return avoidanceMove;

    }
}
```

Figure 4.17: Avoidance.cs

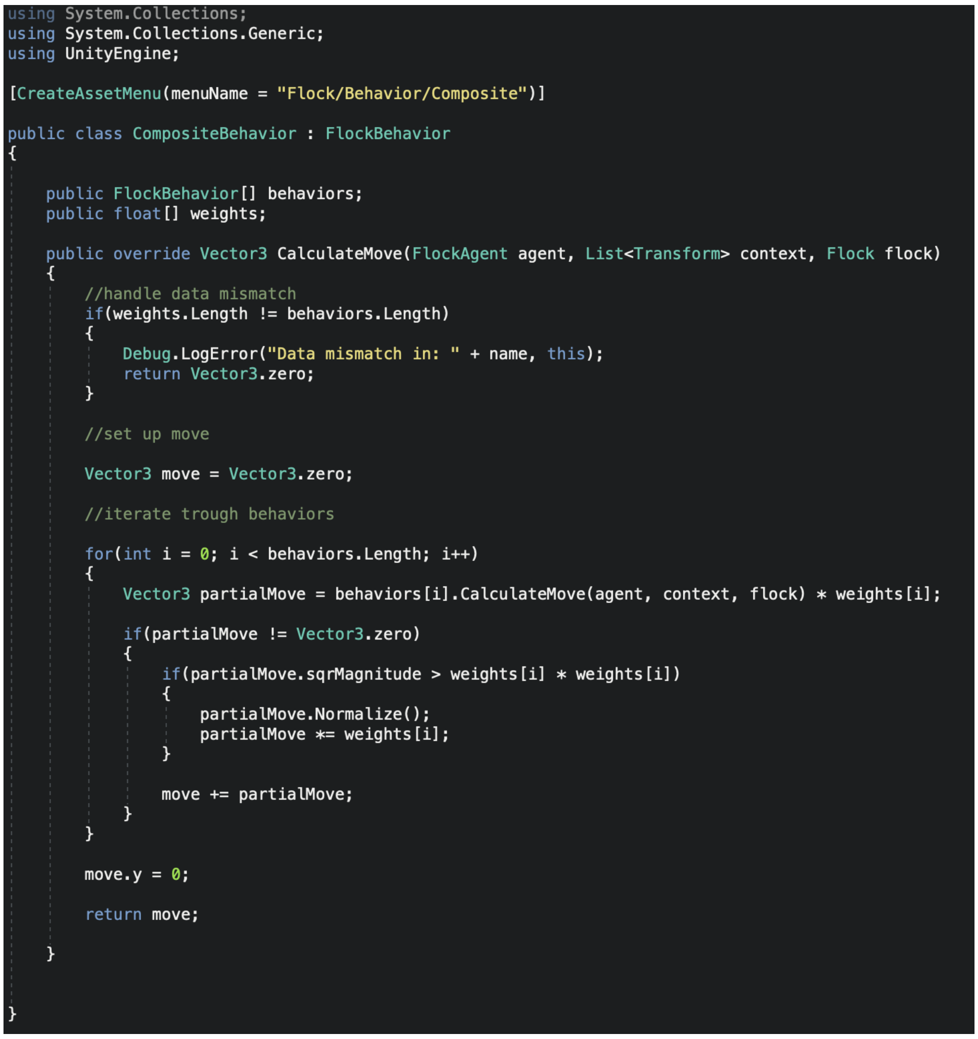
Then we have the cohesion. For each neighboring agent we add to a variable, *cohesionMove*, the position of that neighbor. Then we divide it by the number of neighbors we have and subtract our own position from *cohesionMove*. And we already have the movement vector of cohesion.



```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu(menuName = "Flock/Behavior/Cohesion")]

public class CohesionBehavior : FilteredFlockBehavior
{
    public override Vector3 CalculateMove(FlockAgent agent, List<Transform> context, Flock flock)
    {
        //if no neighbours return no adjustment
        if (context.Count == 0)
        {
            return Vector3.zero;
        }

        //add all point together and average
        Vector3 cohesionMove = Vector3.zero;
        List<Transform> filteredContext = (filter == null) ? context : filter.Filter(agent, context);
        foreach (Transform item in filteredContext)
        {
            cohesionMove += item.position;
        }
        cohesionMove /= context.Count;

        //create offset from agent position

        cohesionMove -= agent.transform.position;

        return new Vector3(cohesionMove.x, 0f, cohesionMove.z);

    }
}
```

Figure 4.18: Cohesion.cs

Finally we have another script that is in charge of joining these 3 behaviors and the movements that we have calculated, in a single movement. This script is called *CompositeBehavior.cs*. To each behavior we assign a weight. With this what we get is, with different configurations of the weights, to obtain different behavior results. For example if we give the separation a very high weight, the agents will end up separating a lot and it will give the feeling that they are more independent, that they don't go almost in a group. For each behavior, its assigned movement is calculated and considered as a partial movement. These partial movements are added up and a final movement is obtained.

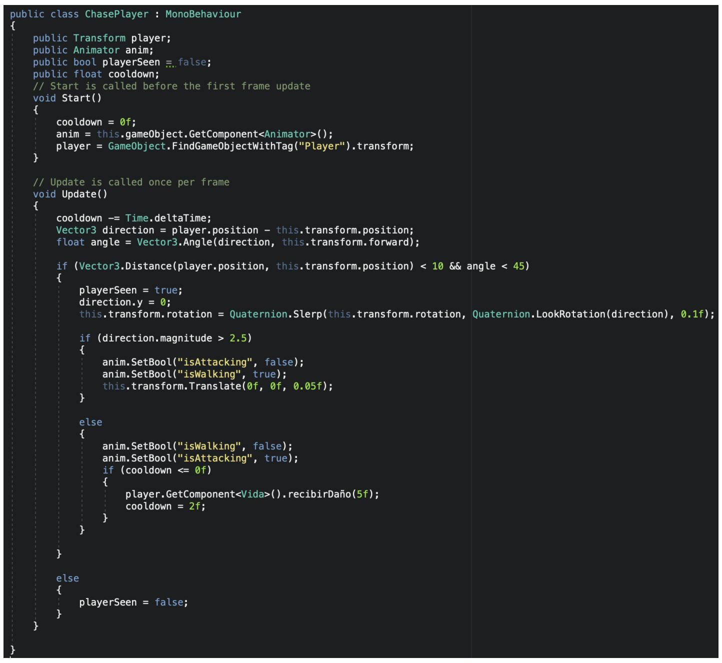


```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu(menuName = "Flock/Behavior/Composite")]

public class CompositeBehavior : FlockBehavior
{

    public FlockBehavior[] behaviors;
    public float[] weights;

    public override Vector3 CalculateMove(FlockAgent agent, List<Transform> context, Flock flock)
    {
        //handle data mismatch
        if(weights.Length != behaviors.Length)
        {
            Debug.LogError("Data mismatch in: " + name, this);
            return Vector3.zero;
        }

        //set up move

        Vector3 move = Vector3.zero;

        //iterate trough behaviors

        for(int i = 0; i < behaviors.Length; i++)
        {
            Vector3 partialMove = behaviors[i].CalculateMove(agent, context, flock) * weights[i];

            if(partialMove != Vector3.zero)
            {
                if(partialMove.sqrMagnitude > weights[i] * weights[i])
                {
                    partialMove.Normalize();
                    partialMove *= weights[i];
                }

                move += partialMove;
            }
        }

        move.y = 0;

        return move;

    }


}
```

Figure 4.19: CompositeBehavior.cs

Before I finish with the flocking, I also want to mention that there are some filters for the contexts. There are two of them specifically. One filter is in charge of taking into account, in the context, only the agents that belong to the same flock. That is, if I create two *Flock* objects and execute them, if the agents of both *Flock* would get close, they would not start mixing and following each other. They'll work independently. The other filter that takes into account is an obstacle layer. This is used to apply it to the separation behavior so that it can also avoid colliding with the obstacles on stage.

This would end the entire flocking algorithm. We'd already have a group of zombies moving around the map together. But if we figure that out, we've only got the zombies walking around. We haven't added anything about attacking the player or his brother. This is easily fixed with a couple of scripts, *ChasePlayer.cs* and *ChaseBrother.cs*. I'm going to show only the code of one of them, since the other one is the same, the only thing that changes is the target (the player or the brother). In this script there is a boolean variable, playerSeen or brotherSeen, with which we control if the zombie has seen any of these. If the variable is set to false, the zombie will keep on working and moving thanks to its *FlockAgent.cs* and its *Move()* function. But, we have made a small modification, and if the variable is set to true, we don't execute that function. Instead the zombie does the thing related to the *ChasePlayer.cs* or *ChaseBrother.cs* script that we can see below.



```csharp
public class ChasePlayer : MonoBehaviour
{
    public Transform player;
    public Animator anim;
    public bool playerSeen = false;
    public float cooldown;
    // Start is called before the first frame update
    void Start()
    {
        cooldown = 0f;
        anim = this.gameObject.GetComponent<Animator>();
        player = GameObject.FindGameObjectWithTag("Player").transform;
    }

    // Update is called once per frame
    void Update()
    {
        cooldown -= Time.deltaTime;
        Vector3 direction = player.position - this.transform.position;
        float angle = Vector3.Angle(direction, this.transform.forward);

        if (Vector3.Distance(player.position, this.transform.position) < 10 && angle < 45)
        {
            playerSeen = true;
            direction.y = 0;
            this.transform.rotation = Quaternion.Slerp(this.transform.rotation, Quaternion.LookRotation(direction), 0.1f);

            if (direction.magnitude > 2.5)
            {
                anim.SetBool("isAttacking", false);
                anim.SetBool("isWalking", true);
                this.transform.Translate(0f, 0f, 0.05f);
            }

            else
            {
                anim.SetBool("isWalking", false);
                anim.SetBool("isAttacking", true);
                if (cooldown <= 0f)
                {
                    player.GetComponent<Vida>().recibirDaño(5f);
                    cooldown = 2f;
                }
            }

        }

        else
        {
            playerSeen = false;
        }
    }

}
```

Figure 4.20: ChasePlayer.cs

We have two different scripts, one to attack the player and one to attack the brother, because finally in the game design, I wanted there to be 3 groups of zombies. One, the genetic algorithm, to look for the brother. Others that, through flocking, would attack only the player. And others that, also through flocking, would attack only the brother. In this way the player has to prioritize some things over others. Maybe he meets a horde of zombies, but he doesn't want to spend his few bullets on killing them, because they are the ones who go after the brother, for example, and ignore him. Maybe it makes more sense to run away and keep looking for his brother before the zombies get him.

After finishing the development of these other zombies, there was less to do to have the game finished. I thought that maybe there was a lack of history or context in the game. So I got inspired and edited an intro video for the game. This video was mentioned in a previous section, is played when pressing the "Intro" button on the main menu. I also created a couple more videos about when the player would lose or win the game. When I made these last videos, I realized that I hadn't implemented anything related to the winning condition. I hadn't even implemented the brother. So I looked for a cartoon model [26] in the asset store and applied a couple of scripts to it. Basic things like life and some animations. I added a house [27] to the map and thought that the goal of the game could be to find the brother and guide him home.

So the brother, besides the script of life, has another script in which when he sees the player begins to follow him. If the player goes too fast, the brother will be a little behind and therefore lose sight of him and stay still again. So the player once he finds his brother, he will have to guide him home slowly and of course controlling that the zombies do not kill them.

Well, after all the foundations on which the game was going to work were finished, it was time to give it some more shape. To do that, I asked several people to test the game and give me feedback. This would help me find bugs and know what things were missing to make the game fun. After multiple and continuous tests, bug fixes, suggestions to include other enemies and obstacles, we ended up with a much more complete video game than in its initial version.

These are some of the additions:

- A new type of zombie [28] was added that is capable of going through objects and becoming invisible as it goes through them. It only attacks the player, but it's faster and does more damage.It behaves like a basic patrol and attacks the player when it enters their range of vision. (If you have heart problems, be careful, because this zombie gives some interesting scares)

- Another type of enemy [29] was included that "protects" the Business-Zombie. This is because, since this zombie is inactive while the flies are searching, it is very easy to accumulate ammunition and go to kill it without taking any risks. These other zombies are standing still near the Business-Zombie and if the player gets close they will attack him. If the player walks away, these zombies will go back to the Business-Zombie and stay by his side again.

- Obstacles were added to complicate the search for the brother. For example, there is an area that prevents passage because there is a fire [30] in the middle. To move forward, the player must find water and put out the fire.
- A second level has been included with a city [31] theme. After the player rescues his brother in the first level and escapes from the forest, both of them reach a city where there are also zombies. The goal is to look for survivors [32] to help. This level includes a new enemy, a bat [33] that attacks by throwing fireballs.
- A third level has been added, set in a desert [34]. The premise is that, after finding a couple of survivors in level 2, they tell the player that they know in which area the zombies started to originate. And they tell him that there's a curse [35]. So the player goes (this time alone, the survivors stay to look after their brother) to the desert to break the curse. This level includes another unique and different enemy. It is a skeleton that wanders in the desert and runs away from the player when it sees him. But once the player succeeds in damaging it, it starts to defend itself and attack by throwing magic spheres. It is mandatory to kill it because it drops a key [36] needed to finish the level.

4.1.1 Problems

In this section I would like to mention the problems I have had throughout the development of the project.

The main problem I had was designing the genetic algorithm. Initially, the zombies that were going to work with that algorithm were going to spawn, complete the steps that their genes tell them to do and disappear. When they finished their genome, since they didn't know what else to do, I wanted them to disappear and the next generation to be created. The problem is that this was not sustained in terms of narrative/gameplay. What was the point of zombies dying on their own after a minute of gameplay? That's when I decided that the zombies would stay on patrol after finishing their genome. When I wanted to save the genes in a file I also had problems, because if I killed a zombie it would be destroyed. So at the end of that generation I was missing individuals with their respective gene information. Apparently I couldn't destroy their gameObjects until I saved their data. So I wanted to make that when I killed a zombie its gameObject would only be deactivated, but it gave me problems anyway, it didn't always work and sometimes even if the zombie was dead it kept appearing but with the animations broken. In the end the solution was to scale them to 0 when they were killed and disable their scripts so they wouldn't keep moving and then, when creating the next generation and after saving the information of all the genes, destroy the gameObject of those who had died.

Clearly this was later discarded. Realizing that those zombies, that that genetic algorithm, was in a dynamic environment and therefore would not progress well I changed the design of NPC.

I had some stupid problems with the zombie animations. Apparently I declared static the variable of the animator, so every time a zombie changed its animation they all changed it. It was a simple thing to fix but I had a hard time figuring out what the problem was.

I put the objects around the map like trees, bushes, plants and so on in a new layer called "obstacles". This was so that in the genetic algorithm, when checking the "scores" of the zombies, I could throw a raycast and it would only count the obstacles that it hit. Well, the action of putting objects in another layer caused me a visual problem. The camera didn't render those objects well. With certain parameters in the camera, the objects didn't appear, and with others the objects were duplicated. It turned out to be a Clear Flags problem, the solution was to put it in "Depth Only" instead of "Skybox".

I think I just had one more problem and it was related to the flocking algorithm. Once implemented, finding a good combination of weights between alignment, avoidance and cohesion was a complicated task. Sometimes zombies were too far apart, other times when they encountered an obstacle they wanted to turn around and walk away, but since they had another zombie behind them they also wanted to stay away from it so it kept spinning on itself. Finally I found a combination that works well but it would still be interesting to keep trying different combinations to see what other results I could find.

4.2 Results

Based on the objectives cited in point 1.2, I can ensure that all the main objectives have been completed including also the secondary objectives

Thanks to the development of this project I have managed to learn how the genetic algorithms and AI swarming techniques work. And as we have seen throughout section 4.1 it seems that I have also learned how to implement them. The penultimate point may not have been developed as I thought. At first I imagined that having two groups of zombies with different behaviors I would have to program or configure something to make them look cohesive. But it hasn't happened. When I started the project, in the game, we had 3 types of zombies with 3 different colors, as we mentioned at the beginning of this document. This was done because I wanted the player to know that there were different zombies with different behaviors. That is, I wanted the player to be able to differentiate them. But I realize that this is silly and it's also an advantage for the player. If, in terms of appearance, all zombies are the same, the player won't know what each zombie is capable of, so it'll be harder for him to decide on his moves. Besides, all zombies, even if there are groups that act differently, will seem to belong to the same group. So, in short, that penultimate goal was already working without me having to apply anything special.

My last objective was to have a demo available that would show the result of having achieved the above objectives. Basically a demo where both artificial intelligence techniques were implemented and working. The demo[6] is available for both MacOs and Windows here:

https://drive.google.com/drive/folders/1Oexb1e3s6hhiVT1C-4C5xD6jQ_7Yy8x_?usp=sharing

You can also access the entire project from Git:

https://github.com/Jonho27/DFP

In addition to the objectives in section 1.2, when writing the GDD I also made it a personal goal of having a finished, though short, game. I wanted to have a game with 3 levels where each level takes place in a different scenario. And finally I achieved it. The game could obviously be improved a lot more in terms of the narrative aspect and the overall design. But I'm still proud to finish what I've finished. To close this chapter, I'm going to add some pictures of the game within the different levels so that you can visually see what you've been reading.

[6]I'm still working on the project. So I will be updating the builds as well as the gitHub repository.

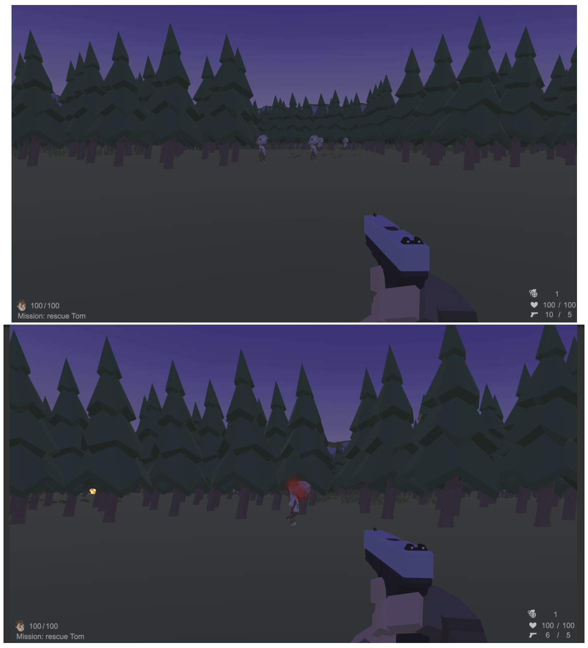
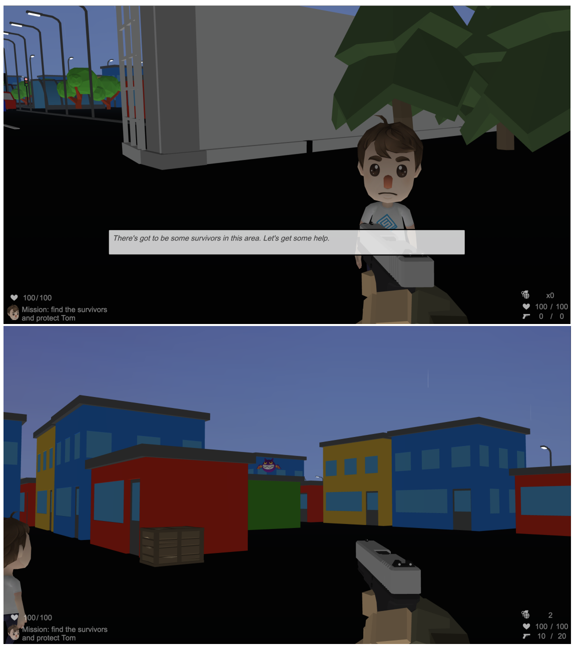


Figure 4.21: Killing a Zombie on Level 1.


There's got to be some survivors in this area. Let's get some help.

100 / 100
Mission: find the survivors and protect Tom
x0
100 / 100
0 / 0
100 / 100
Mission: find the survivors and protect Tom
2
100 / 100
10 / 20

Figure 4.22: Starting the Level 2.

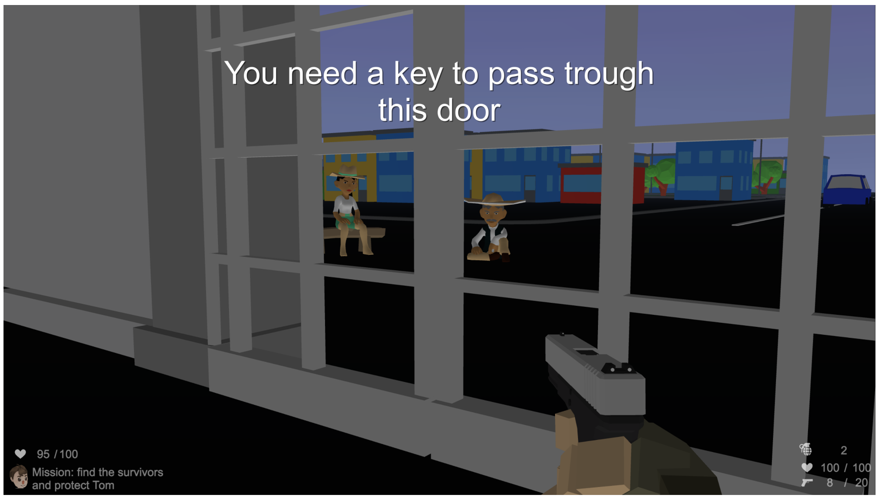

You need a key to pass trough this door
95 / 100
Mission: find the survivors and protect Tom
2
100 / 100
8 / 20

Figure 4.23: The survivors of Level 2.

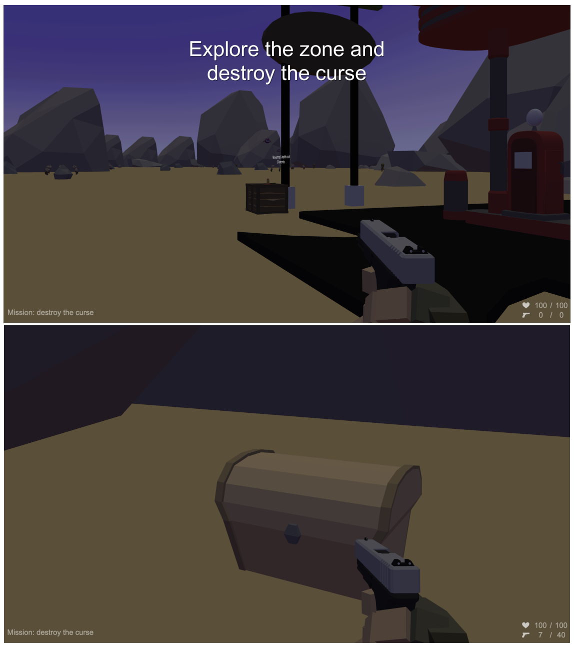
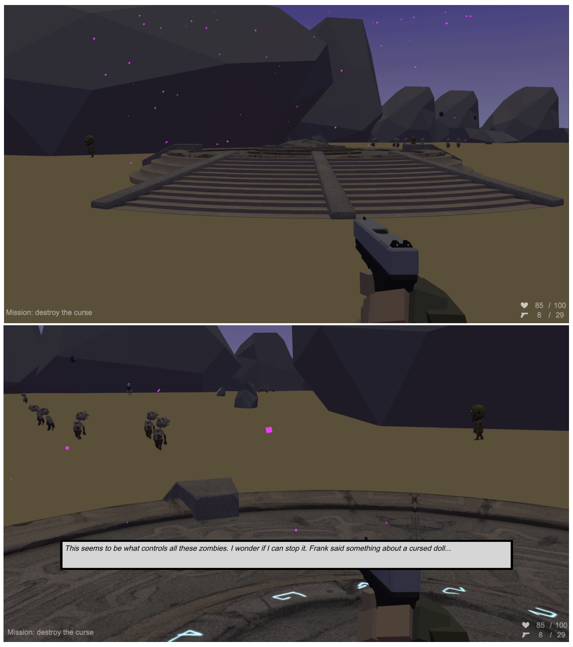

Explore the zone and destroy the curse
Mission: destroy the curse
100 / 100
0 / 0
Mission: destroy the curse
100 / 100
7 / 40

Figure 4.24: Start of the Level 3 and the chest which contains the cursed doll.



This seems to be what controls all these zombies. I wonder if I can stop it. Frank said something about a cursed doll...

Figure 4.25: A Skull Platform which is responsible for this whole catastrophe.

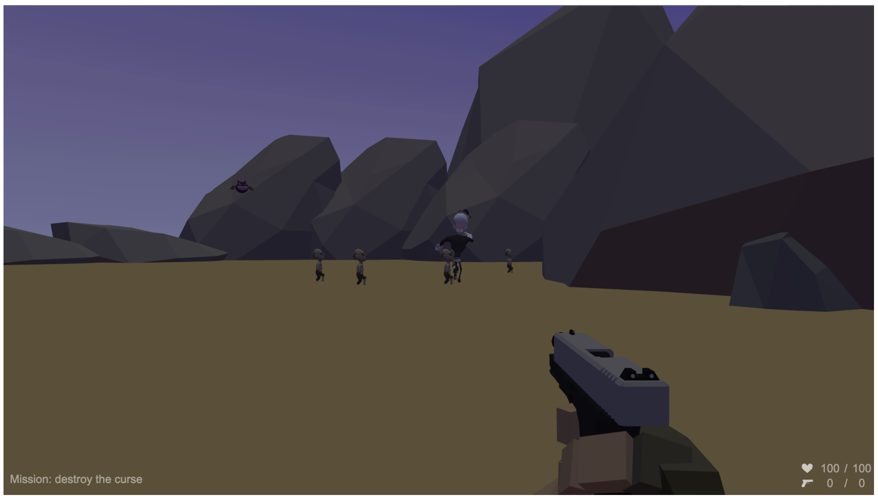


Figure 4.26: The skeleton that holds the key to the chest. Running away, of course.

Chapter 5

Conclusions and Future Work

Contents

In this chapter, the conclusions of the work, as well as its future extensions are shown.

5.1 Conclusions

I think the DFP is a great opportunity to do what you really want. After 4 years of career, or more, you can finally have an idea of your own and end up executing it. It's true that during the course of your degree you will carry out projects in which you can express an idea, but these projects are usually very marked/guided by the subject to which they belong.

What I am clear about is that artificial intelligence applied to video games has a huge future. More specifically, that which is related to machine learning or deep learning. What I have developed in this project is just an appetizer, I have seen really incredible things/projects on the Internet. This project has introduced me to a new world of videogames. It is clear that there are many different branches and all of them are important, but the branch of artificial intelligence has caught all my attention. I'd like to continue learning more about it.

5.2 Future work

The truth is, I like the way the video game finally looks. I'd like to keep working on it, adding more levels and maybe implementing other kinds of enemies. Whenever we've developed a video game, it's always been a demo. We've never gone beyond that to finish it. I think this is because working for a whole semester on the development of a video game for X subject ends up burning you out. It's not the same to develop something calmly and completely free, as it is to develop something with a specific schedule, with deadlines, and with guidelines in terms of goals or design that limit you when developing the videogame. So, for once, it would be good to continue with the development of a project, even if it is in a slow way.

In fact, I'm still improving the game. With the help of friends who test the game, I'm correcting errors, removing or adding things, etc. The main work involved in this project, genetic algorithms and swarm behaviors, is already done. So now I'm polishing the game, making it entertaining and getting a flow in the gameplay.

Bibliography

[1] James McCaffrey. Introduction to q-learning using c#. https://docs.microsoft.com/en-us/archive/msdn-magazine/2018/august/test-run-introduction-to-q-learning-using-csharp. Accessed: 2020-05-16.

[2] Alexandre Thomas. Ganttproject: free project management tool for windows, macos and linux. (2020). https://www.ganttproject.biz/download. Accessed: 2020-05-16.

[3] Unity Technologies. Unity (2019). https://unity3d.com/es/get-unity/download. Accessed: 2020-05-16.

[4] Microsoft. Visual studio (2019). https://visualstudio.microsoft.com/es/downloads/. Accessed: 2020-05-16.

[5] Microsoft. Github (2019). https://github.com/. Accessed: 2020-05-16.

[6] Adobe. Photoshop (2019). https://www.adobe.com/es/products/photoshop/free-trial-download.html. Accessed: 2020-05-16.

[7] Adobe. Premiere pro (2017). https://www.adobe.com/es/products/premiere/free-trial-download.html. Accessed: 2020-05-16.

[8] Joel Spolsky (Atlassian). Trello (2020). https://trello.com. Accessed: 2020-05-16.

[9] Unity Technologies. Unity asset store. https://assetstore.unity.com. Accessed: 2020-05-16.

[10] Adobe. Mixamo (2020). https://www.mixamo.com/. Accessed: 2020-05-16.

[11] Unity Technologies. Unity documentation. https://docs.unity3d.com/Manual/index.html . Accessed: 2020-05-16.

[12] Misc. References for the dfp development. https://www.youtube.com/playlist?list=PLSzlIVtJ1C-p7orhj5JGXcg9a5iGKqRyX. Accessed: 2020-05-16.

[13] V Mathivet. *Inteligencia artificial para desarrolladores : conceptos e implementación en C#*. Cornellà de Llobregat, Barcelona, Spain: ENI editions, 2015.

[14] Ben Fry & Casey Reas. (github) ai-maze-genetic-algorithm-smart-agents. (2018). https://github.com/nj-AllAboutCode/AI-Maze-Genetic-algorithm-smart-agents. Accessed: 2020-05-16.

[15] David Stenfors. Low poly fps pack - free (sample) | 3d weapons. (2019, june 23). https://assetstore.unity.com/packages/3d/props/weapons/low-poly-fps-pack-free-sample-144839. Accessed: 2020-05-16.

[16] Gigel. Rpg poly pack - lite | 3d landscapes. (2019, july 6). https://assetstore.unity.com/packages/3d/environments/landscapes/rpg-poly-pack-lite-148410. Accessed: 2020-05-16.

[17] Broken Vector. Low poly cliff pack | 3d landscapes. (2016, november 29). https://assetstore.unity.com/packages/3d/environments/landscapes/low-poly-cliff-pack-67289. Accessed: 2020-05-16.

[18] Supercyan. Character pack: Zombie sample | 3d humanoids. (2018, november 11). https://assetstore.unity.com/packages/3d/characters/humanoids/character-pack-zombie-sample-131604. Accessed: 2020-05-16.

[19] cookiepopworks.com. Survival game tools | 3d tools. (2019, august 1). https://assetstore.unity.com/packages/3d/props/tools/survival-game-tools-139872. Accessed: 2020-05-16.

[20] Solum Night. Low poly pack - environment lite (2017, october 19). https://assetstore.unity.com/packages/3d/props/exterior/low-poly-pack-environment-lite-102039. Accessed: 2020-05-29.

[21] MadFireOn. Simple ui elements | 2d icons. (2016, october 1). https://assetstore.unity.com/packages/2d/gui/icons/simple-ui-elements-53276. Accessed: 2020-05-16.

[22] 3DcwCurto. Cartoon business zombie. (2014, october 30). https://assetstore.unity.com/packages/3d/characters/cartoon-business-zombie-24298. Accessed: 2020-05-29.

[23] blujay. low poly fly. (2018, january 22). https://sketchfab.com/3d-models/low-poly-fly-2a48c1d30ebb47fab89fc9868ee1d263. Accessed: 2020-05-29.

[24] J. Millington, I. & Funge. *Artificial Intelligence for Games*. Taylor & Francis, 2009.

[25] Ben Fry & Casey Reas. Flocking examples (2020). https://processing.org/examples/flocking.html. Accessed: 2020-05-16.

[26] Supercyan. Character pack: Free sample | 3d humanoids. (2017, march 22). https://assetstore.unity.com/packages/3d/characters/humanoids/character-pack-free-sample-79870. Accessed: 2020-05-16.

[27] Johnny Kasapi. Furnished cabin | 3d urban. (2019, august 12). https://assetstore.unity.com/packages/3d/environments/urban/furnished-cabin-71426. Accessed: 2020-05-16.

[28] M.eye. Cartoon zombie [sample] | 3d characters. (2014, july 20). https://assetstore.unity.com/packages/3d/characters/cartoon-zombie-sample-17622. Accessed: 2020-05-16.

[29] ESsplashkid. Zcharacter. (2019, november 13). https://assetstore.unity.com/packages/3d/characters/zcharacter-157331. Accessed: 2020-05-29.

[30] Unity Technologies. Unity particle pack 5.x | asset packs. (2016, october 22). https://assetstore.unity.com/packages/essentials/asset-packs/unity-particle-pack-5-x-73777. Accessed: 2020-05-16.

[31] Mixaill. Simple city - low poly assets. (2019, december 2). https://assetstore.unity.com/packages/3d/environments/urban/simple-city-low-poly-assets-61541. Accessed: 2020-05-29.

[32] New Vision Studios. Free western wild west characters. (2019, december 9). https://assetstore.unity.com/packages/3d/characters/humanoids/free-western-wild-west-characters-146422. Accessed: 2020-05-29.

[33] amusedART. Free monster bat. (2019, november 27). https://assetstore.unity.com/packages/3d/characters/free-monster-bat-158125. Accessed: 2020-05-29.

[34] Streakbyte. Low poly ultimate environment pack. (2020, march 18). https://assetstore.unity.com/packages/3d/environments/low-poly-ultimate-environment-pack-164923#version-current. Accessed: 2020-05-29.

[35] Kubiqual. Skull platform. (2017, december 20). https://assetstore.unity.com/packages/3d/props/skull-platform-105664. Accessed: 2020-05-29.

[36] RoboCG. Handpainted keys (2015, july 24). https://assetstore.unity.com/packages/3d/handpainted-keys-42044. Accessed: 2020-05-29.

Appendix A

Source code

This appendix compiles the most relevant scripts in the development of the video game. The project has about 25 scripts, so it is understandable that not all of them are included here. Only those related to the behavior of the Zombies and the player and sibling controllers are included. If you are interested in seeing all the code of the project, you can find the link to the repository that contains it in section 4.2.

Genetic Algorithm Scripts

Listing A.1: DNA.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.IO;

public class DNA
{

  public List<Vector3> genes = new List<Vector3>();
  public DNA(int genomeLenght = 5)
  {
    //Debug.Log("Lo hago random");
    for (int i = 0; i < genomeLenght; i++)
    {
      genes.Add(new Vector3(Random.Range(-1.0f, 1.0f), 0f, Random.Range(-1.0f, 1.0f)));
    }
  }

  public DNA(DNA parent, DNA partner, float mutationRate = 0.01f)
  {
    for (int i = 0; i < parent.genes.Count; i++)
    {
      float mutationChance = Random.Range(0.0f, 1.0f);
      if (mutationChance <= mutationRate)
      {
        genes.Add(new Vector3(Random.Range(-1.0f, 1.0f), 0f, Random.Range(-1.0f, 1.0f)));
      }
      else
      {
        int chance = Random.Range(0, 2);
        if (chance == 0)
        {
          genes.Add(parent.genes[i]);
        }
        else
        {
          genes.Add(partner.genes[i]);
        }

      }
    }
  }


}
```

Listing A.2: GeneticPathfinder.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class GeneticPathfinder : MonoBehaviour
{
    public float creatureSpeed;
    public float pathMultiplier;
    public float rotationSpeed;
    int pathIndex = 0;
    public DNA dna;
    public bool hasFinished = false;
    public int identifier;
    public LayerMask obstacleLayer;
    bool hasBeenInitialized = false;
    bool hasCrashed = false;
    Vector3 target;
    Vector3 nextPoint;
    Quaternion targetRotation;
    public bool targetFound;
    public Transform brother;

    private void Start()
    {
        brother = GameObject.FindGameObjectWithTag("Brother").transform;
    }

    public void InitCreature(DNA newDna, Vector3 _target, int id)
    {
        dna = newDna;
        target = _target;
        nextPoint = transform.position;
        hasBeenInitialized = true;
        identifier = id;
    }

    private void Update()
    {
        if (targetFound)
        {
            Vector3 direction = brother.position - this.transform.position;
            float angle = Vector3.Angle(direction, this.transform.forward);
            direction.y = 0;
            this.transform.rotation = Quaternion.Slerp(this.transform.rotation, Quaternion.
                 LookRotation(direction), 0.1f);
            if (direction.magnitude > 1)
            {

                this.transform.Translate(0f, 0f, 0.25f);
            }
        }

        else if (hasBeenInitialized && !hasFinished)
        {
            if (pathIndex == dna.genes.Count || Vector3.Distance(transform.position, target) < 0.5f)
            {
                hasFinished = true;
            }
            if ((Vector3)transform.position == nextPoint)
            {
                nextPoint = (Vector3)transform.position + dna.genes[pathIndex] * pathMultiplier;

                Vector3 lookPos = nextPoint - transform.position;
                lookPos.y = 0;
                targetRotation = Quaternion.LookRotation(lookPos);
                pathIndex++;
            }
            else
            {
                transform.position = Vector3.MoveTowards(transform.position, nextPoint, creatureSpeed * Time.deltaTime);
            }
            if (transform.rotation != targetRotation)
            {
                transform.rotation = Quaternion.Slerp(transform.rotation, targetRotation, rotationSpeed * Time.deltaTime);
            }
        }

    }

    public float fitness
    {
        get
        {
            float dist = Vector3.Distance(transform.position, target);
            if (dist == 0f)
            {
                dist = 0.0001f;
            }
            RaycastHit[] obstacles = Physics.RaycastAll(transform.position, target, obstacleLayer);
            float obstacleMultiplier = 1f - (0.15f * obstacles.Length);
            return (60 / dist) * (hasCrashed ? 0.75f : 1f) * obstacleMultiplier;
        }
    }
    private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.layer == 8)
        {
            hasFinished = true;
            hasCrashed = true;
        }

        if (other.transform.CompareTag("Brother"))
        {
            targetFound = true;

        }
    }

    private void OnCollisionEnter(Collision collision)
    {
        if (collision.gameObject.tag == "Obstacle")
        {

            hasFinished = true;
            hasCrashed = true;

        }

        if (collision.transform.CompareTag("Brother"))
        {
            targetFound = true;

        }
    }
}
```

Listing A.3: PopulationController.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI;

public class PopulationController : MonoBehaviour
{

    public List<GeneticPathfinder> population = new List<GeneticPathfinder>();
    public GameObject creaturePrefab;
    public int populationSize = 10;
    public int genomeLenght;
    public float cutoff = 0.3f;
    public int survivorKeep = 5;
    [Range(0f, 1f)]
    public float mutationRate = 0.01f;
    public Transform spawnPoint;
    public Vector3 spawnVector;
    public Transform end;
    public float tiempo = 0.0f;
    public string subname;
    public bool saveData;
    bool haveWin;
    public Transform brother;
    public Animator anim;
    public bool playerSeen = false;
    public float cooldown;
public int numGeneracion = 1;
NavMeshAgent navMeshAgent;

void InitPopulation()
{


    for (int i = 0; i < populationSize; i++)
    {
        GameObject go = Instantiate(creaturePrefab, new Vector3(spawnVector.x, spawnVector.y + Random.Range(0f, 1.5f), spawnVector.z), Quaternion.identity);
        go.GetComponent<GeneticPathfinder>().InitCreature(new DNA(genomeLenght), end.position, i);
        go.name = "Fly." + i;
        population.Add(go.GetComponent<GeneticPathfinder>());
    }


}

void NextGeneration()
{
    numGeneracion++;
    int survivorCut = Mathf.RoundToInt(populationSize * cutoff);
    List<GeneticPathfinder> survivors = new List<GeneticPathfinder>();

    for (int i = 0; i < survivorCut; i++)
    {
        survivors.Add(GetFittest());
    }

    for (int i = 0; i < population.Count; i++)
    {
        Destroy(population[i].gameObject);
    }
    population.Clear();

    for (int i = 0; i < survivorKeep; i++)
    {

        GameObject go = Instantiate(creaturePrefab, new Vector3(spawnVector.x, spawnVector.y + Random.Range(0f, 1.5f), spawnVector.z), Quaternion.identity);
        go.GetComponent<GeneticPathfinder>().InitCreature(survivors[i].dna, end.position, i);
        go.name = "FlySurvivor." + i;
        population.Add(go.GetComponent<GeneticPathfinder>());

    }
    while (population.Count < populationSize)
    {
        for (int i = 0; i < survivors.Count; i++)
```

```
            {

                GameObject go = Instantiate(creaturePrefab, new Vector3(spawnVector.x,
                    spawnVector.y + Random.Range(0f, 1.5f), spawnVector.z), Quaternion.
                    identity);
                go.GetComponent<GeneticPathfinder>().InitCreature(new DNA(survivors[i].dna,
                    survivors[Random.Range(0, survivorCut)].dna, mutationRate), end.position
                    , i);
                go.name = "FlyMutate." + i;
                population.Add(go.GetComponent<GeneticPathfinder>());
                if (population.Count >= populationSize)
                {
                    break;
                }
            }
        }


        for (int i = 0; i < survivors.Count; i++)
        {
            Destroy(survivors[i].gameObject);

        }
    }

    private void Start()
    {
        cooldown = 0f;
        anim = this.gameObject.GetComponent<Animator>();
        brother = GameObject.FindGameObjectWithTag("Brother").transform;
        navMeshAgent = GetComponent<NavMeshAgent>();
        spawnVector = spawnPoint.position;

        if (ChooseDifficulty.difficulty == 0)
        {
            populationSize = 100;

        }

        else if (ChooseDifficulty.difficulty == 1)
        {
            populationSize = 150;
        }

        else if (ChooseDifficulty.difficulty == 2)
        {
            populationSize = 200;
        }
        InitPopulation();
    }

    private void Update()
    {
        tiempo += Time.deltaTime;
        if(haveWin)
        {
            cooldown -= Time.deltaTime;
            Vector3 direction = brother.position - this.transform.position;
            float angle = Vector3.Angle(direction, this.transform.forward);

            if (this.GetComponent<Vida>().valor > 0)
            {
                if (Vector3.Distance(brother.position, this.transform.position) < 10 &&
                     angle < 45)
                {
                    navMeshAgent.isStopped = true;
                    playerSeen = true;
                    direction.y = 0;
                    this.transform.rotation = Quaternion.Slerp(this.transform.rotation,
                         Quaternion.LookRotation(direction), 0.1f);



                    if (direction.magnitude > 3)
                    {
                        anim.SetBool("isAttacking", false);
                        anim.SetBool("isWalking", true);
                        this.transform.Translate(0f, 0f, 0.1f);
                    }

                    else
                    {

                        anim.SetBool("isWalking", false);
                        anim.SetBool("isAttacking", true);
                        if (cooldown <= 0f)
                        {
                            StartCoroutine(kickAttack());
                            cooldown = 3.8f;
                        }
                    }

                }

                else
                {
                    navMeshAgent.isStopped = false;
                    anim.SetBool("isAttacking", false);
                    anim.SetBool("isIdle", false);
                    anim.SetBool("isWalking", true);
                    navMeshAgent.SetDestination(brother.position);
                }
            }
        }

        else if (!HasActive() && !haveWin)
        {
            NextGeneration();
        }
    }

    IEnumerator kickAttack()
    {
        yield return new WaitForSeconds(0.6f);
        brother.GetComponent<Vida>().recibirDaño(40f);

    }

    GeneticPathfinder GetFittest()
    {
        float maxFitness = float.MinValue;
        int index = 0;
        for (int i = 0; i < population.Count; i++)
        {
            if (population[i].fitness > maxFitness)
            {
                maxFitness = population[i].fitness;
                index = i;
            }
        }

        GeneticPathfinder fittest = population[index];
        population.Remove(fittest);
        return fittest;
    }

    bool HasActive()
    {
        for (int i = 0; i < population.Count; i++)
        {
            if (population[i].targetFound)
            {
                haveWin = true;
                destroyPopulation(i);
                return true;
            }

            if (!population[i].hasFinished || population[i].targetFound)
            {
                return true;
            }
        }
        return false;
    }

    void destroyPopulation(int winner)
    {
        for (int i = 0; i < population.Count; i++)
        {
                if(i != winner)
                {
                    Destroy(population[i].gameObject);
                }

            }
        }
    }
```

Flocking Algorithm Scripts

Listing A.4: FlockAgent.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[RequireComponent(typeof(Collider))]

public class FlockAgent : MonoBehaviour
{
    Flock agentFlock;
    public bool imDead;
    public Flock AgentFlock { get { return agentFlock; } }
    Collider agentCollider;
    public Collider AgentCollider { get { return agentCollider; }}
    public string myName;
    public bool playerSeen;
    public ChaseBrother chaseBrother;
    public ChasePlayer chasePlayer;

    // Start is called before the first frame update
    void Start()
    {
        if(this.gameObject.tag == "Blue")
        {
            chaseBrother = gameObject.GetComponent<ChaseBrother>();
        }

        else if (this.gameObject.tag == "Orange")
        {
            chasePlayer = gameObject.GetComponent<ChasePlayer>();
        }
        imDead = false;
        agentCollider = GetComponent<Collider>();
    }

    public void Initialize(Flock flock)
    {
        agentFlock = flock;
    }

    public void Move(Vector3 velocity)
    {
        if(chaseBrother != null)
        {
            if (!chaseBrother.playerSeen)
            {
                transform.forward = velocity;
                transform.position += velocity * Time.deltaTime;
            }
        }

        else
        {
            if (!chasePlayer.playerSeen)
            {
                transform.forward = velocity;
                transform.position += velocity * Time.deltaTime;
            }
        }
    }
}
```

Listing A.5: FlockBehavior.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public abstract class FlockBehavior : ScriptableObject
{
    public abstract Vector3 CalculateMove(FlockAgent agent, List<Transform> context, Flock
         flock);
}
```

Listing A.6: Flock.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Flock : MonoBehaviour
{

  public FlockAgent agentPrefab;
  List<FlockAgent> agents = new List<FlockAgent>();
  public FlockBehavior behavior;
        public Vector3 spawnPosition = Vector3.zero;

  [Range(1, 500)]
  public int startingCount = 250;
  const float AgentDensity = 0.08f;

  [Range(0.75f, 1f)]
  public float driveFactor = 1f;
  [Range(0.75f, 100f)]
  public float maxSpeed = 0.75f;
  [Range(1f, 20f)]
  public float neighbourRadius = 3f;
  [Range(0f, 2f)]
  public float avoidanceRadiusMultiplier = 1f;

  float squareMaxSpeed;
  float squareNeighbourRadius;
float squareAvoidanceRadius;
      public float SquareAvoidanceRadius { get { return squareAvoidanceRadius; } }


// Start is called before the first frame update
void Start()
  {
      spawnPosition = gameObject.transform.position;
  squareMaxSpeed = maxSpeed * maxSpeed;
  squareNeighbourRadius = neighbourRadius * neighbourRadius;
  squareAvoidanceRadius = squareNeighbourRadius * avoidanceRadiusMultiplier *
       avoidanceRadiusMultiplier;

      for(int i = 0; i < startingCount; i++)
  {
          FlockAgent newAgent = Instantiate(agentPrefab, spawnPosition, Quaternion.Euler(0
               f, Random.Range(0f, 360f), 0f), transform);
          newAgent.name = "Agent" + i;
          newAgent.Initialize(this);
          agents.Add(newAgent);
      }

}

  // Update is called once per frame
  void Update()
  {
      foreach(FlockAgent agent in agents)
      {
          if (agent.imDead)
          {
              agents.Remove(agent);

              Destroy(agent.gameObject);
          }

          else
          {
              List<Transform> context = GetNearbyObjects(agent);

              Vector3 move = behavior.CalculateMove(agent, context, this);
              move *= driveFactor;
              if (move.sqrMagnitude > squareMaxSpeed)
              {
                  move = move.normalized * maxSpeed;
              }

              agent.Move(move);

          }
      }
  }

```

```
    List<Transform> GetNearbyObjects(FlockAgent agent)
    {
        List<Transform> context = new List<Transform>();
        Collider[] contextColliders = Physics.OverlapSphere(agent.transform.position,
             neighbourRadius);
        foreach(Collider c in contextColliders)
        {
            if(c != agent.AgentCollider)
            {
                context.Add(c.transform);
            }
        }
        return context;
    }
}
```

Listing A.7: ContextFilter.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public abstract class ContextFilter : ScriptableObject
{
    public abstract List<Transform> Filter(FlockAgent agent, List<Transform> original);
}
```

Listing A.8: FilteredFlockBehavior.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public abstract class FilteredFlockBehavior : FlockBehavior
{
    public ContextFilter filter;
}
```

Listing A.9: PhysicsLayerFilter

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu(menuName = "Flock/Filter/PhysicsLayer")]

public class PhysicsLayerFilter : ContextFilter
{
    public LayerMask mask;

    public override List<Transform> Filter(FlockAgent agent, List<Transform> original)
    {
        List<Transform> filtered = new List<Transform>();
        foreach (Transform item in original)
        {
            if(mask == (mask | (1 << item.gameObject.layer)))
            {
                filtered.Add(item);
            }
        }

        return filtered;
    }
}
```

Listing A.10: SameFlockFilter.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu(menuName = "Flock/Filter/SameFlock")]

public class SameFlockFilter : ContextFilter
{
    public override List<Transform> Filter(FlockAgent agent, List<Transform> original)
    {
        List<Transform> filtered = new List<Transform>();
        foreach(Transform item in original)
        {
            FlockAgent itemAgent = item.GetComponent<FlockAgent>();
            if(itemAgent != null && itemAgent.AgentFlock == agent.AgentFlock)
            {
                filtered.Add(item);
            }
        }

        return filtered;
    }
}
```

Listing A.11: AlignmentBehavior.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu(menuName = "Flock/Behavior/Alignment")]


public class AlignmentBehavior : FilteredFlockBehavior
```

```
{
    public override Vector3 CalculateMove(FlockAgent agent, List<Transform> context, Flock
         flock)
    {
        //if no neighbours maintain current alignment
        if (context.Count == 0)
        {
            return agent.transform.forward;
        }

        //add all point together and average
        Vector3 alignmentMove = Vector3.zero;
        List<Transform> filteredContext = (filter == null) ? context : filter.Filter(agent,
             context);
        foreach (Transform item in filteredContext)
        {
            alignmentMove += item.forward;
        }
        alignmentMove /= context.Count;

        alignmentMove.y = 0;

        return alignmentMove;

    }
}
```

Listing A.12: AvoidanceBehavior.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu(menuName = "Flock/Behavior/Avoidance")]

public class AvoidanceBehavior : FilteredFlockBehavior
{
    public override Vector3 CalculateMove(FlockAgent agent, List<Transform> context, Flock
         flock)
    {
        //if no neighbours return no adjustment
        if (context.Count == 0)
        {
            return Vector3.zero;
        }

        //add all point together and average
        Vector3 avoidanceMove = Vector3.zero;
        int nAvoid = 0;
        List<Transform> filteredContext = (filter == null) ? context : filter.Filter(agent,
             context);
        foreach (Transform item in filteredContext)
        {
            if(Vector3.SqrMagnitude(item.position - agent.transform.position) < flock.
                SquareAvoidanceRadius)
            {
                nAvoid++;
                avoidanceMove += agent.transform.position - item.position;
            }

        }
        if(nAvoid > 0)
        {
            avoidanceMove /= nAvoid;
        }

        avoidanceMove.y = 0;
        return avoidanceMove;

    }
}
```

Listing A.13: SteerCohesionBehavior.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu(menuName = "Flock/Behavior/SteeredCohesion")]

public class SteerCohesionBehavior : FilteredFlockBehavior
{

    Vector3 currentVelocity;
    public float agentSmoothTime = 2f;

    public override Vector3 CalculateMove(FlockAgent agent, List<Transform> context, Flock
        flock)
    {
        //if no neighbours return no adjustment
        if (context.Count == 0)
        {
            return Vector3.zero;
        }

        //add all point together and average
        Vector3 cohesionMove = Vector3.zero;
        List<Transform> filteredContext = (filter == null) ? context : filter.Filter(agent,
            context);
        foreach (Transform item in filteredContext)
        {
            cohesionMove += item.position;
        }
        cohesionMove /= context.Count;

        //create offset from agent position

        cohesionMove -= agent.transform.position;
        cohesionMove = Vector3.SmoothDamp(agent.transform.forward, cohesionMove, ref
             currentVelocity, agentSmoothTime);
        cohesionMove.y = 0;
        return cohesionMove;

    }
}
```

Listing A.14: CompositeBehavior.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu(menuName = "Flock/Behavior/Composite")]

public class CompositeBehavior : FlockBehavior
{

    public FlockBehavior[] behaviors;
    public float[] weights;

    public override Vector3 CalculateMove(FlockAgent agent, List<Transform> context, Flock
         flock)
    {
        //handle data mismatch
        if(weights.Length != behaviors.Length)
        {
            Debug.LogError("Data mismatch in: " + name, this);
            return Vector3.zero;
        }

        //set up move

        Vector3 move = Vector3.zero;

        //iterate trough behaviors

        for(int i = 0; i < behaviors.Length; i++)
        {
            Vector3 partialMove = behaviors[i].CalculateMove(agent, context, flock) *
                 weights[i];

            if(partialMove != Vector3.zero)
            {
                if(partialMove.sqrMagnitude > weights[i] * weights[i])
                {
                    partialMove.Normalize();
                    partialMove *= weights[i];
                }

                move += partialMove;
            }
        }

        move.y = 0;
        return move;
    }
}
```

General Enemy Controller

Listing A.15: LogicaEnemigo.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine.AI;
using UnityEngine;

public class LogicaEnemigo : MonoBehaviour
{
    private Vida vida;
    private Animator animator;
    private Collider collider;
    private Vida vidaJugador;
    private LogicaJugador logicaJugador;
    public bool vida0 = false;
    public bool estaAtacando = false;
    public float daño = 25;
    public FlockAgent myFlockAgent;
    public GeneticPathfinder myGeneticPathfinder;
    public Chase myChase;
    public bool isDead;

    // Start is called before the first frame update
    void Start()
    {

        myFlockAgent = this.gameObject.GetComponent<FlockAgent>();
        myGeneticPathfinder = this.gameObject.GetComponent<GeneticPathfinder>();
        myChase = this.gameObject.GetComponent<Chase>();
        vida = GetComponent<Vida>();
        animator = GetComponent<Animator>();
        collider = GetComponent<Collider>();
        isDead = false;
    }

    // Update is called once per frame
    void Update()
    {
        RevisarVida();

    }
    void RevisarVida()
    {
        if (vida0) return;
        if (vida.valor <= 0)
        {
            isDead = true;
            vida0 = true;
            if(myFlockAgent != null)
            {

                gameObject.GetComponent<Animator>().SetBool("isDead", true);
                StartCoroutine(FlockMuerto());
            }

            else if(myGeneticPathfinder != null)
            {
                if(myChase.enabled == false)
                {
                    myGeneticPathfinder.hasFinished = true;
                    myGeneticPathfinder.enabled = false;
                    gameObject.GetComponent<Animator>().SetBool("isDead", true);
                    StartCoroutine(Desaparecer());
                }

                else
                {
                    gameObject.GetComponent<Animator>().SetBool("isDead", true);
                    StartCoroutine(Desaparecer());
                }


            }

            else
            {
                gameObject.GetComponent<Animator>().SetBool("isDead", true);
                StartCoroutine(Morir());
            }

        }
    }

    IEnumerator Morir()
    {
        yield return new WaitForSeconds(1.75f);
        Destroy(gameObject);

    }

    IEnumerator Desaparecer()
    {
        yield return new WaitForSeconds(1.75f);
        gameObject.transform.localScale = new Vector3(0, 0, 0);

    }

    IEnumerator FlockMuerto()
    {
        yield return new WaitForSeconds(1.25f);
        myFlockAgent.imDead = true;

    }

    private void OnCollisionEnter(Collision collision)
    {
        if(collision.collider.tag == "Bala")
        {
            gameObject.GetComponent<Vida>().recibirDaño(50f);


        }
    }


}
```

Listing A.16: ChasePlayer.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ChasePlayer : MonoBehaviour
{
    public Transform player;
    public Animator anim;
    public bool playerSeen = false;
    public float cooldown;
    // Start is called before the first frame update
    void Start()
    {
        cooldown = 0f;
        anim = this.gameObject.GetComponent<Animator>();
        player = GameObject.FindGameObjectWithTag("Player").transform;
    }

    // Update is called once per frame
    void Update()
    {
        cooldown -= Time.deltaTime;
        Vector3 direction = player.position - this.transform.position;
        float angle = Vector3.Angle(direction, this.transform.forward);

        if (Vector3.Distance(player.position, this.transform.position) < 10 && angle < 45)
        {
            playerSeen = true;
            direction.y = 0;
            this.transform.rotation = Quaternion.Slerp(this.transform.rotation, Quaternion.LookRotation(direction), 0.1f);



            if (direction.magnitude > 2.5)
            {
                anim.SetBool("isAttacking", false);
                anim.SetBool("isWalking", true);
                this.transform.Translate(0f, 0f, 0.05f);
            }

            else
            {

                anim.SetBool("isWalking", false);
                anim.SetBool("isAttacking", true);
                if (cooldown <= 0f)
                {
                    player.GetComponent<Vida>().recibirDaño(5f);
                    cooldown = 2f;
                }



            }

        }

        else
        {
            playerSeen = false;

        }
    }
}
```

Player Controller

Listing A.17: LogicaJugador.cs

```

using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.SceneManagement;

public class LogicaJugador : MonoBehaviour
{
    public Vida vida;
    public GameObject hermano;

    public Text currentLifeText;
    public Text totalLifeText;
    // Start is called before the first frame update
    void Start()
    {
        vida = GetComponent<Vida>();
        totalLifeText.text = "100";

    }

    // Update is called once per frame
    void Update()
    {
        RevisarVida();
        currentLifeText.text = vida.valor.ToString();

    }

    void RevisarVida()
    {


        if (vida.valor <= 0)
        {

            SceneManager.LoadScene("GameOver1");

        }
    }

    private void OnTriggerEnter(Collider other)
    {
        if(other.gameObject.tag == "Casa" && hermano.GetComponent<brotherController>().
             playerSeen)
        {
            Debug.Log("Has ganado");
            SceneManager.LoadScene("Win");
        }
    }

    private void OnCollisionEnter(Collision collision)
    {
        if(collision.gameObject.tag == "Blue" || collision.transform.tag == "Orange" ||
             collision.transform.tag == "Genetic")
        {
            if (Input.GetButton("KnifeAttack"))
            {
                collision.gameObject.GetComponent<Vida>().recibirDaño(20f);
            }
        }
    }
}
```

Brother Controller

Listing A.18: brotherController.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class brotherController : MonoBehaviour
{
    public Transform player;
    public Animator anim;
    public bool playerSeen = false;

    // Start is called before the first frame update
    void Start()
    {
        anim = this.gameObject.GetComponent<Animator>();
        player = GameObject.FindGameObjectWithTag("Player").transform;
    }

    // Update is called once per frame
    void Update()
    {
        if(gameObject.GetComponent<Vida>().valor <= 0)
        {
            Debug.Log("Has perdido");
            SceneManager.LoadScene("GameOver2");
        }

        Vector3 direction = player.position - this.transform.position;
        float angle = Vector3.Angle(direction, this.transform.forward);

        if (Vector3.Distance(player.position, this.transform.position) < 12 && angle < 45)
        {
            anim.SetBool("isWalking", false);
            anim.SetBool("isIdle", false);
            anim.SetBool("isHello", true);
            playerSeen = true;
            direction.y = 0;
            this.transform.rotation = Quaternion.Slerp(this.transform.rotation, Quaternion.
                LookRotation(direction), 0.1f);



            if (direction.magnitude > 3.5 && direction.magnitude < 10)
            {
                anim.SetBool("isIdle", false);
                anim.SetBool("isHello", false);
                anim.SetBool("isWalking", true);
                this.transform.Translate(0f, 0f, 0.05f);
            }

            else if(direction.magnitude > 8)
            {
                anim.SetBool("isWalking", false);
                anim.SetBool("isIdle", false);
                anim.SetBool("isHello", true);
            }

            else
            {
                anim.SetBool("isWalking", false);
                anim.SetBool("isHello", false);
                anim.SetBool("isIdle", true);
            }



        }

        else
        {
            anim.SetBool("isWalking", false);
            anim.SetBool("isHello", false);
            anim.SetBool("isIdle", true);
            playerSeen = false;
        }
    }

    private void OnTriggerEnter(Collider other)
    {
        if(other.gameObject.tag == "Path")
        {
            Debug.Log("Collider lol");
            transform.position = new Vector3(transform.position.x, 1.5f, transform.position.
                z);
        }
    }

    private void OnCollisionEnter(Collision collision)
    {
        if (collision.gameObject.tag == "Path")
        {
            Debug.Log("Collider lol");
            transform.position = new Vector3(transform.position.x, 1.5f, transform.position.
                z);
        }
    }
}
```

List of Figures