Emergency behaviors as a first-class mechanic in video games

Ricardo Sanz Ferrero

Advisor: Dr. Luis Amable García Fernández

Acknowledgments

First of all, I would like to thank my Degree Final Project supervisor, Luis Amable García Fernández for his inspiration before the beginning of the development and all the help provided during the development phase.

Contents

1. Technical proposal 7
1.1. Abstract 7
1.2. Keywords 7
1.3. Related subjects 7
1.4. Work motivation 8
1.5. Objectives 9
1.6. Expected results 10
1.7. Task planning and temporary 10
1.8. Resources evaluation 11
2. Environment and initial state 13
3. System analysis and development 15
3.1. Requirements analysis 15
3.2. Environments element design 15
3.2.1. Cleaning obstacle 15
3.2.2. Collecting obstacle 16
3.2.3. Exploring obstacle 17
3.3. Entities design 19
3.3.1. Cleaners complex behavior 19
3.3.2. Collectors complex behavior 22
3.3.3. Explorers complex behavior 26
3.4. Squad system 31
3.4.1. Squad system structure 32
3.4.2. Squad system actions 32
3.4.3. Squad battles system 34
3.4.4. In-game squad creator 40
3.5. Emergency parameters 42
3.5.1. Cleaners emergency parameters 43
3.5.3. Explorers emergency parameters 44
3.5.4. Squad emergency parameters 44
4. Problems and results 46
4.1. Problems 46
4.2. Results 48
4.3. Hours spent 52
4.4. Project access and download 53

5. Conclusions and future work 54
6. Bibliography 55
7. Annexes 58
7.1. Annex 1 (Selection Process) 58
7.2. Annex 2 (Squad Class) 60
7.3. Annex 3 (NavMesh Unity) 61
7.3.1. Area masks 61
7.3.2. Determine if a destination is valid 62
7.4. Annex 4 (Squad actions) 63
7.4.1. Squad cover 63
7.4.2. Squad returns to the player 63
7.4.3. Squad changes formation 64
7.4.4. Wrestler follows player 65
7.4.5. Wrestler changes formation position 66
7.4.6. Wrestler breaks formation 66
7.4.7. Wrestler assigned as leader 66
7.4.8. Wrestler joins a squad 67
7.5. Annex 5 (Saving system) 69
7.6. Annex 6: Demo Game Design Document (GDD) 71
7.6.1. Narrative design 71
7.6.2. Characters 72
7.6.3. Environment 74
7.6.4. Playability 75
7.6.5. Art 76
7.6.6. HUD and Controls 76
7.7. Code architecture 79
7.7.1. Entities architecture 79
7.7.2. Obstacles architecture 82
7.7.3. UI and Input architecture 83
7.8. Annex 8: Code captures 86
List of figures 96
Videos 99

1. Technical proposal

The Game Design Document of the created Demo is in Annex 6.

1.1. Abstract

This document consists on the project report of the Video Games Design and Development Degree Final Project by Ricardo Sanz Ferrero. The project consists on exploring how to integrate the emergency behaviors applied to multi-agent systems as an active element in a video game with the objective of providing unpredictability and unexpected behaviors for the player. These emergent behaviors will occur when the agents interact between them, with the scenario or with the player. Besides, the project presents a great focus on the squad's coordination and battles between them applying these emergent behaviors. All these emergent behaviors are delimited by a set of parameters which are easily manipulated by anyone allowing scalability and understandability.

1.2. Keywords

Unpredictability, Complex Systems, Emergency Behaviors, Multi-Agent Systems, Emergency Parameters, Squads Coordination and Battles.

1.3. Related subjects

- VJ1231 (Artificial Intelligence (AI)): the AI techniques studied in this subject will help to develop these emergent behaviors and the Squad coordination.
- VJ1234 (Advanced Interaction Techniques): the subject showed us some advanced techniques which can be really useful. Besides, the main purpose of this project is an uncommon technique in the field of Video games.
- VJ1227 (Game Engines): in this subject Unity 3D was explored and studied so it is useful for the development of this project.
- VJ1208 (Programming II): the base of the Object-Oriented Programming in the career and the programming language used for developing this project is studied in this subject.
- VJ1215 (Algorithms and Data Structure): when working with a lot of entities it is important to keep in mind to create efficient algorithms and to use the proper data structures. This is studied in this subject.

1.4. Work motivation

Nowadays, video games are a field that is constantly increasing in society. On the one hand, there are hundreds of thousands of people who play video games and on the other hand there is a big number of people who make these video games. In my case, I love both to play video games and to program it.

So, my work motivation is divided between the player's point of view and the programmer's point of view. First, the player's point of view is going to be explained. The main approach of the video games is to entertain the player, and for accomplishing it, the video game needs to be funny.

For the player who has played the same game many times, it becomes boring and repetitive. This means that the game has lost a very important factor: the unpredictability. Thus, one of the main guidelines in my project is the emergent behaviors applied to multi (AI) agents' systems. This is, the unpredictable behaviors that emerge when some agents interact between them and with the scenario. Usually, these behaviors are unique and unexpected for the player. Applying these behaviors in a video game will ensure different results for the player in each game.

Secondly, the other half of my motivation for doing this project is the designer and programmer point of view. We understand emergent behaviors as the set of behaviors that emerge from agents with certain parameters. The main advantage of this is that if we vary these parameters a bit, the result we obtain will be completely different and this opens a huge range of possibilities. Imagine a video game in which there is a troop enemy and a troop of allies. The objective is to defeat the enemy troop with your troop. Each squad has different types of roles (archers, close range knights, tanks, healers…). The game is divided into some levels and each level has an increase of difficulty. There are some things that you can change in your troop: the position of the troop, the shape of the troop, the health of each troop member, the healer's time delay, etc. So, depending on all these parameters the difficult will vary. The designer and the programmer have to adjust these parameters for the enemy troops in order to get a funny experience for the player. The main advantage of this is that there is no any kind of global AI controller who controls each troop member. They act based on the parameters on their own and the final result emerge of all these acts interacting between them.

Another alternative of doing this would be to program the whole AI behaviors on each troop for each level and to have a controller that controls everything. This is infinitely less efficient and expensive to program.

In the next points I will explain more extensively what the emergency behaviors and multi-agent systems are and how I develop my project around it.

1.5. Objectives

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

- To understand how the emerging behaviors work.
- To think and design some emergent behaviors.
- To achieve complex emerging behaviors based on interaction of simple behaviors.
- To think which parameters will vary the behavior obtained.
- To vary these parameters and see how the behaviors change.
- To organize all these behaviors in order to be easily manipulated by a designer.
- To create squads and apply them these emergent behaviors.
- To create a simple in-game squad creator and store them.
- To get battles between the squads and see how the result changes varying some parameters.
- To divide the set of parameters into configuration files and easily applying them in testing phases.
- To think examples of how to apply these emergent behaviors in a video game.
- To create a demo in which different kind of behaviors are shown.
- To create the base of these emergent-behaviors that can be extended by others.

There are some secondary objectives:

- To learn more about Unity AI agents.
- To learn more about Unity AI System. [Annex 3]
- To learn how to develop a well-scalable code architecture. [Annex 7]
- To learn how to create an organized level manager.
- To learn more about the functionalities of Scriptable Objects.

Besides, must be known that emergency behaviors applied to multi-agent systems are typically used in simulations and other problems. In video games they are commonly used in a different way than the one I am going to use. They are used as a passive element of the game, like crowds of people, flocks of birds, etc. and the main purpose of my project is to use them as an active element on a video game. Thus, the video game could not be played without them.

1.6. Expected results

The expected results for this project are the followings:

- To achieve complex behaviors as the interaction of simple actions.
- To achieve applying these emergent complex behaviors to multi-agent systems.
- To create some of these agents in a video game as an active element.
- To have a squad coordination system done entirely by me.
- To have battles between squads and basing them in emergency behaviors.
- To obtain completely different behaviors varying the emergency parameters.
- To be able to share the project to a designer and allow them to create new behaviors varying the emergency parameters.
- To be able to play a demo in which all the behaviors are shown.

1.7. Task planning and temporary

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. It must be known that all the tasks are between two estimated times. The first one is the one in case everything goes well and the second one is for the case in which there are unforeseen events. As have been said before, some things have been changed from the initial days until now. The planning is the following:

- Reading and learning theory of emerging behaviors: to read and find information about the emergency behaviors and how they are related to the multi-agent systems. This takes between 10 and 15 hours. (Mathivet, 2018)
- Entities behaviors design: to think and design the emergency behaviors and how they will occur, which entities will be involved, how they will interact with the scenario, between them, which parameters are those which emerge behaviors, etc. This takes between 10 and 15 hours.
- Entities behaviors coding: to develop the behaviors of the entities designed. This part contains the “cleaners”, the “explorers” and the “collectors”. In order to able the future expansion of the project, a good code architecture needs to be done to ensure scalability. This takes between 50 and 60 hours.
- Squad system develop: the hardest part of the project. That is, to apply emergent behaviors theory into multi-agent squads and battle them. It is very important in this part to think a good way of represent ing the squads. Besides, the in-game squad creator is contained in this part. This part takes between 100 and 110 hours.

- Organization of the parameters in configuration files: This is a very useful task because it allows the easy manipulation of the parameters by the designers of the game. It allows to test just changing some parameters and see how it affects to the resultant emergent behavior. This part takes between 40 and 50 hours.
- Project report and other documents: the documentation of the project is a very important way of keeping track of the evolution of the project so some documents need to be done. This part will take between 40 and 50 hours.
- Video game demo: a demo in which all the entities developed and the squad interaction is shown. The main objective of the project is not to develop a video game. It is to show how powerful is to use these AI techniques and how well-scalable they are. This part will take between 60 and 70 hours.

The total estimated time oscillates between 310 and 360 hours. In order to end this point, the way of working must be stated. Due to at the same term of these project is being carried out, external internships are being performed there is no possibility of dedicating more than 4 hours per day. These hours will be done in the afternoon. Some days more or less hours can be done depending on the commitments.

1.8. Resources 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.

- Windows PC (i7 CPU, 16 GB RAM and GPU GTX 1080): I will use this PC in a regular way.
- MacBook Pro (i5 CPU, 8 GB RAM): I will use this PC in occasions when I am not in the usual place of the project development. Besides, it will serve to show demos of the project to the supervisor and the final presentation.
- Unity 3D 2018.3f: the engine of the project. It will be developed on it. (Unity Technologies, 2019)
- Visual Studio Community 2017 for Windows and Mac: Attached with Unity 3D is a very useful tool for coding and allows switching between Unity and Visual. Besides, Visual Studio 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. (Microsoft, 2019)
- GitHub (GitHub Desktop): a git repository client where I will upload my work. Besides, this is a good tool for me because as I said, I will be switching between different devices and this is a good way of having everything update. (GitHub, 2019)

- Adobe Photoshop CS6: this will be used to design the user interface (UI) and some of the elements of the games in 2D. (Adobe, 2019)
- Audacity: a powerful tool for sounds and music. (Audacity, 2019)
- Vegas Pro 16.0: to edit the videos that this report contains. (MAGIX Software GmbH, 2019)
- NetLogo: this is a program I will use to see examples of multi-agent and complex systems. Besides you can see how some of them are coded. It is far from the code I will use but it can give you a main idea of how to do it. (Wilensky, 2016)
- 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. (Atlassian, 2019)

2. Environment and initial state

The project's environment is almost as important as its development so it has to be clearly defined. It is important talking about the interests of the author of the project in the sector of video games. I like to program so I want to become a video games Artificial Intelligence programmer. This is probably the main reason that I choose this project. Besides, my Degree Final Project supervisor talked me about the multi-agent systems and I thought it was really interesting. So, the idea of the project is born as a mix of AI gameplay and multi-agent systems.

The environment of the video games industry is continuously increasing their technologies and the AI is becoming a very important part of the games, so I think it is useful to keep working on it and doing research in order to improve or complement what already exists. Talking about the development of the project team, the members of the project are Ricardo Sanz Ferrero as a main responsible for the project and Luis Amable García Fernández as the supervisor.

First of all, it should be known what an emergency behavior is. It is a complex behavior which gives a sense of intelligence and it occurs unexpectedly as a consequence of a series of simple actions. Secondly, a multi-agent system is a system formed by some intelligent agents. They are used to resolve problems that a simple agent would not be able to solve. So, in this project emergent behaviors are going to be applied to multi-agent systems in order to create entities that can be used in a video game.

Before starting talking about the project itself, we are going to see some examples of emergency behaviors. We are going to talk about animals. In particular to the ants: when we see hundreds of ants forming a long line on the street, they are acting by emergency. This is, they need going for food, so they look for it everywhere. When an ant finds some food, she takes it and bring it to the nest. The curious thing is that the ant, while it is going to the nest, is shedding pheromones so when another ant finds that trail of pheromones, it follows it and it ends up where the food is located. This phenomenon, applied to hundreds of ants, produces a complex behavior and it is as simple as:

- Move until finding food.
- Take the food and go back to the nest (leaving pheromones).
- Follow the pheromones when you find the trail.

The persistence value (time until disappear) of these pheromones in the scenario will provide different patterns of ants' complex behavior. These three simple actions lead to a complex behavior. These kind of insects (ants, termites, bees…) are known as eusocial insects and the organization they show is known as eusociality. (Biomolecular Blog, s.f.)

So, now that we have what an emergency behavior applied to a multi-agent system is, we are going to talk about what this project is going to consist. As it has been said before, the emergent behaviors applied to multi-agent systems is a very good way of ensure unpredictability to a video game, creating unexpected behaviors for the player and the main advantage is that it allows big variations on the final result just variating some parameters.

Next, I am going to list the entities I decided to explore applying them emergency behaviors. In section 3.2, they will be explained action by action with some code captures.

- Cleaners: they clean debris for the player. In a video game this can be used to unlock a path that was locked, for example. It is based on the eusociality of termites.
- Collectors: they collect resources from production sources and take it to a destination point. This can be used to collect some in-game resource, like elixir. It is based on the eusociality of ants.
- Explorers: they find a path from one point to another that for the player was unattainable. A priori, it is not based on any eusocial insect. It has been the choice and design of the author.

These entities have the objective of helping the player to complete the level. Without their help is impossible to do it. Apart from that, it has been decided to apply emergency behaviors to squads. A squad is a couple of entities that moves and interact in group and each entity inside the squad adopt a different role. A typical example of squads is a medieval army. Inside the army, there are different types of roles and they do different actions based on it. Nevertheless, the whole army acts together. The objective of the project is to face two or more squads and make them fight acting by emergency.

In this project, the squad chose by the author is formed by the following:

- Archer-like entities.
- Close combat-like entities.
- Tank-like entities.
- A leader.

Besides, a simple in-game squad creator will allow the player to create squads based on the situation. All about the squads is more extensively explained in section 3.4.

3. System analysis and development

3.1. Requirements analysis

One of the main advantages of the emergency complex behaviors is that is formed by some simple behaviors. These do not need a great hardware to work so they can be coded in almost every kind of computer. An example of this is the software NetLogo. This is full of demos with emergent behaviors coded, and it can run in every hardware.

Nonetheless, my purpose in this project is to develop these emergent behaviors applied to multi-agent systems and adapt them in order to include them in a video game, and for doing it some better hardware will be needed.

So, the main requirements come when 3D models appear. The code is well-structured and it will not cause any kind of efficiency problem. Besides, due to my main purpose is to program the emergency behaviors applied to multi-agent systems, most of the art of video game will be taken from external sources. These all will be shown at Annex 6.

Before continuing, some of the explanations will need code captures in order to fully understand how it works. Some of these captures will be referenced in Annex 8.

3.2. Environments element design

In this point, all the elements that form the environment in which the entities will interact to emerge behaviors will be explained. It is important to know that all these elements are going to be called from now to the end “obstacles” and each entity will have a different type of “objective”, which is one of the following obstacles.

3.2.1. Cleaning obstacle

The Cleaning obstacle, also known as Debris obstacle is formed by a set of objects (debris parts) scattered on the floor. They block the road beyond where they are located. In Figure 1 we can see an example of this.

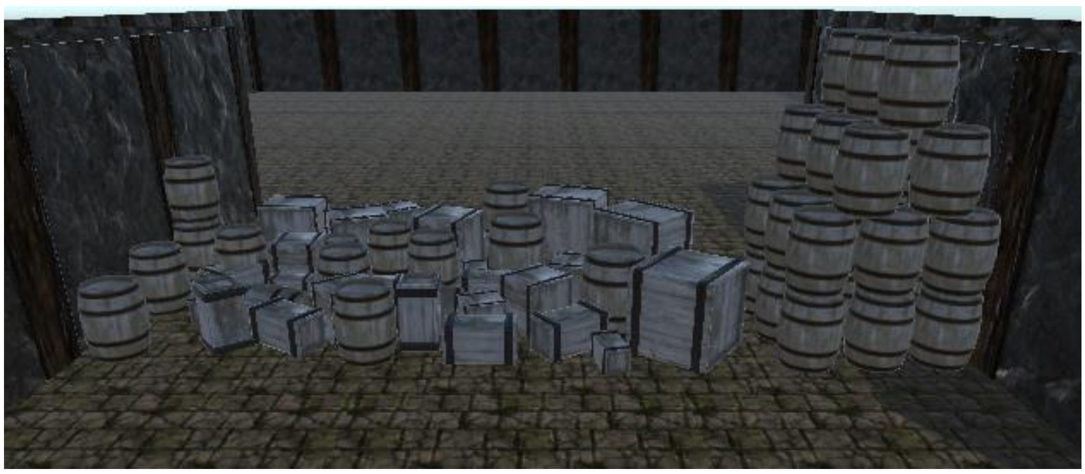


Figure 1: Debris Obstacle

In order to pass it, all the parts must be placed one on top of the other. We understand as a part each box or barrel. This obstacle is the one that the cleaners can solve. So, this is the objective of the cleaners.

As it can be seen in Figure 86 (Annex 8), each debris obstacle has a list of all their parts and another list of all the parts that are potential. This means all the parts that are already placed in the floor and cannot be carried by a cleaner. This is an optional simple trick to avoid accumulate all the debris parts in a point that we do not want. For example, if we do not want to place all the parts in the barrels of the left, we just mark as potential those which are on the center and on the right.

In order to place all the debris parts in the same point, we just mark one of all the potentials as "already placed". Just one. It is done randomly as we can see in the method *ResetSettledDownDebris().*

Figure 87 shows the simple code of each part. It is formed by a reference of the cleaner that is charging it and a bool that indicates if it is already placed. It is important to bear in mind that if it is placed (*settledDown = true*) the cleaner will not be able to carry it.

In section 3.3 the cleaners entities will be explained and shown how the behavior emerges when they interact with it.

3.2.2. Collecting obstacle

The Collecting Obstacle, also known as Elixir/Energy obstacle is formed by a set of objects (stones of elixir) around a collecting machine. The collecting machine is empty and needs to be fulled itself with the resource that is extracted from the stones.

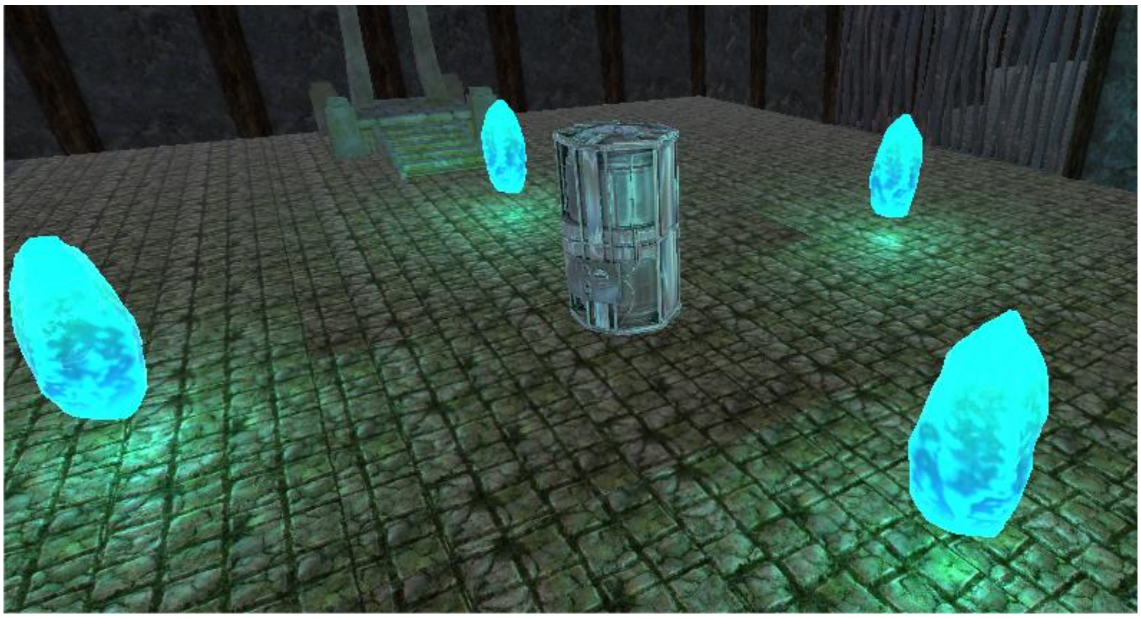


Figure 2: Energy Obstacle

For passing this obstacle, all the energy stored in the stones must be in the collecting machine. This obstacle is the one that collectors can solve. So, this is the objective of the collectors.

We can see in Figure 88 that it consists of a list of all the stones around the machine and the total amount of resource available. These values are filled automatically at the beginning.

In Figure 6 we can see how simple is the code of each stone. It just consists of the initial elixir in and how many collectors can be collecting at the same time. It also has a type, which can be Elixir or Energy. From the point of view of a programmer it is the same, both types have exactly the same value and the only reason that exists is to provide variety.

We can see that in the *Update()* method it is checked if the stone is empty. So, it will disable the emission in that case.

3.2.3. Exploring obstacle

The Exploring obstacle, also known as *Multipath*, is probably the most complex obstacle. It is formed by many sets of corridors. It is recommended to watch Figure 3 in order to figure out how this obstacle is.

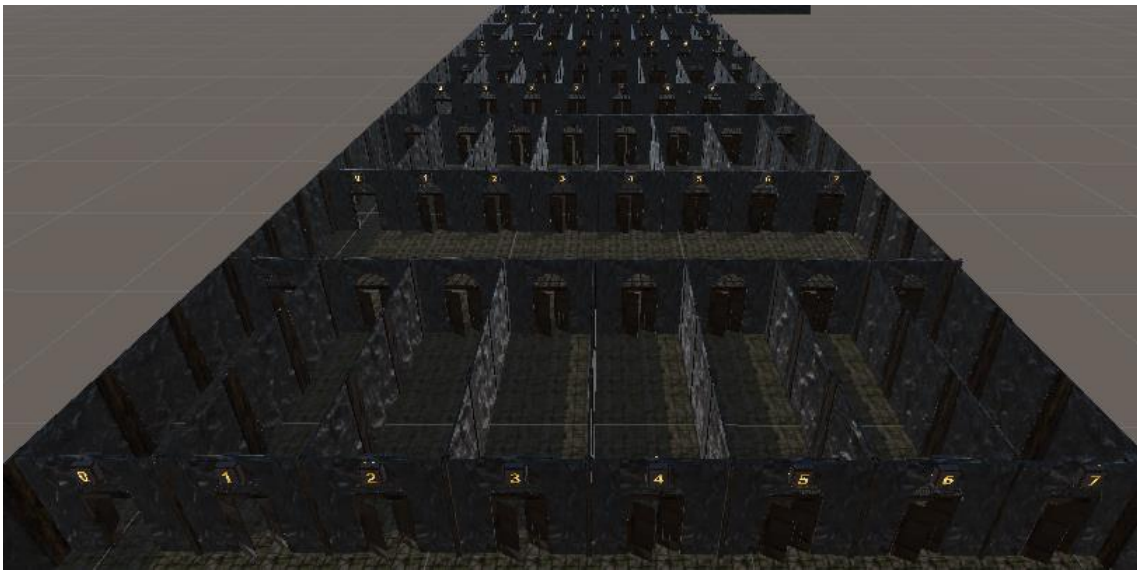


Figure 3: Multipath

For passing this obstacle, the combination of consecutive corridors must be known. For example, if the combination is 0-1-4-6-2, this means that the first corridor will be the 0, the second one will be the 1, the third one the 4, the fourth one the 6 and the last one the 2. If at any point this combination is broken, the position of the player will be restarted to the beginning. This is, for example, if the player takes the 0 corridor at the beginning, nothing will happen. At this point, the player should take the corridor 1, instead of it, he decides to go through the corridor 2. As soon as he passes the corridor exit door, he will be at the beginning again.

If a player decides to try randomly the correct combination, he will probably waste a lot of time, so this is why the explorers exist. This is the obstacle than the explorers can solve, so this is the objective of them.

The class *MultipathObstacle.cs* is empty. Nonetheless, it exists because of a well-scalable code architecture (this will be seen in the Code Architecture Point). The code of the multipath is in another class because it is more efficient to access to the children objects than access to the grandchildren. For understanding this, it is important to see Figure 8.

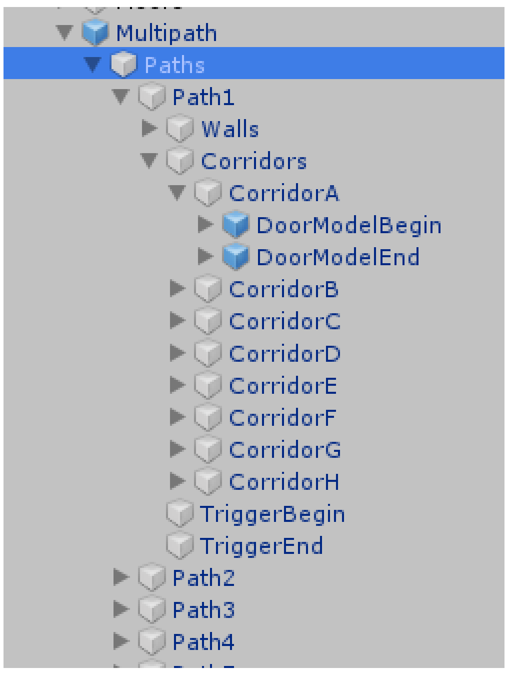


Figure 4: Multipath Hierarchy

As we can see in Figure 8, the *MultipathObstacle.cs* class is in the multipath object and the Paths object contains all the logic that allows that the multipath works. This is because there is direct access to the children of this object. From the script attached on Paths we can access every path; and from every path, we can access all the content inside each of them. The hierarchy is going to be explained:

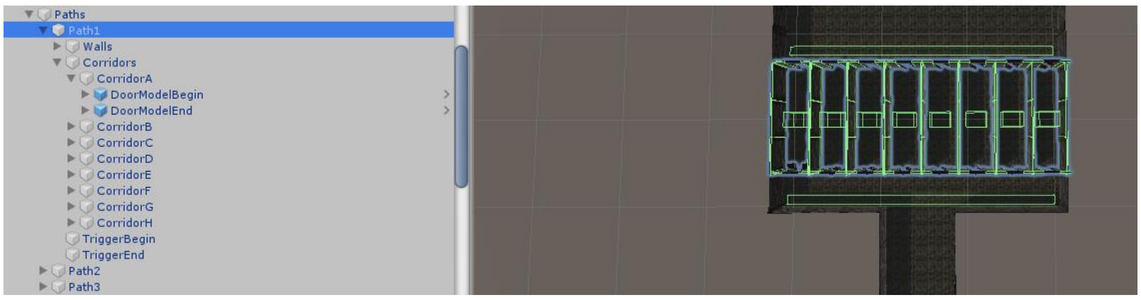


Figure 5: Set of corridors

As we can see in Figure 9, each set of corridors is formed by the walls of each of them, a trigger at the begin, a trigger at the end, and a trigger inside each corridor. A trigger is an empty object with a trigger collider that serve to know if any game element is touching it.

The begin trigger (Figure 90) detects if the player was in the correct corridor, the middle trigger (Figure 91) sets which in corridor is the player passing through and the end trigger (Figure 92) checks if the multipath is finished. In case of yes, nothing happens. In case of no, the position of the player is restarted to the beginning.
Before finishing with the multipath, it is important to see this demo in which the player is trying to pass the multipath without the explorer's help.

Figure 6: Multipath demo

[Video 1]

3.3. Entities design

At this point, it is going to be explained the set of actions of each type of entity (cleaners, explorers and collectors) and how they interact with the environment and between them to emerge the complex behavior.

3.3.1. Cleaners complex behavior

The cleaners clean debris efficiently and they pile them in one point. This complex behavior is based on the eusociality of termites. These insects can build enormous termite mounds known as cathedrals.

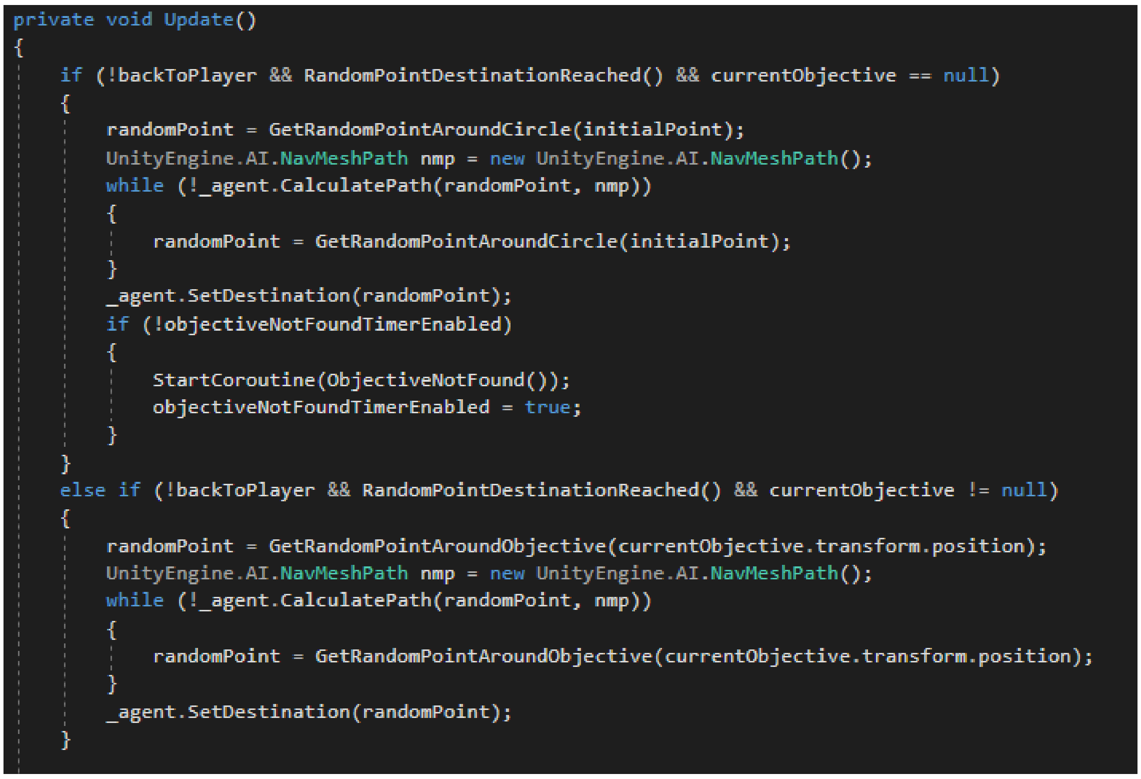
As every complex behavior, it must be divided into simple actions. The cleaners have the following actions associated:

1. Move randomly.
2. If they collide with an obstacle or debris, they grab it.
3. If they collide with another obstacle or debris while they are carrying one, they place the carried one over which they have just hit.

And that is all. These three simple actions can produce incredible complex behaviors. It will be seen later. But first, these actions are going to be translated to code.

The first action is the movement one. All these entities are AI agents from Unity so the movement system is to set a destination and the AI system creates a path between the current point and the destination point. They move randomly because this is the base of the emerging behaviors. They do not reason: their behavior emerges while they perform some actions and interact between them and the scenario.

As it can be seen in Figure 7, these entities need to have an objective in order to start doing their job. This is needed because as they form part of a video game, if the player decides to spawn them in a place where there are not debris to clean, they have to detect it and go back to player. This code is inside an *Update()* method, which means that is executing every frame.

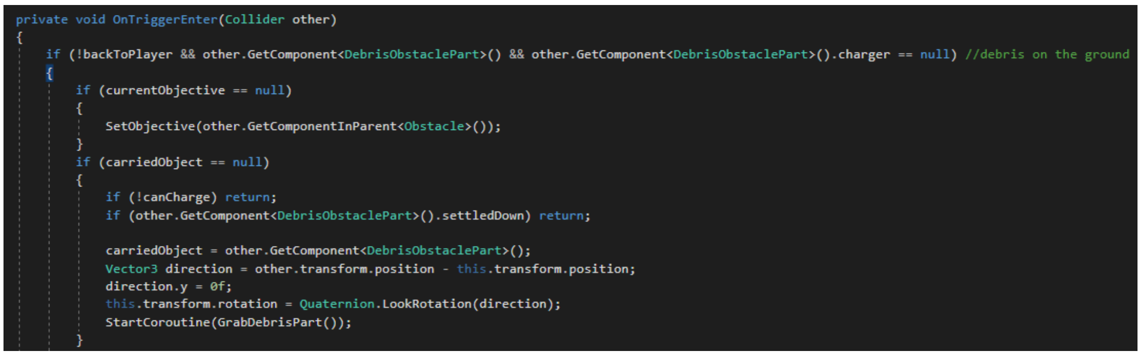


```
private void Update()
{
    if (!backToPlayer && RandomPointDestinationReached() && currentObjective == null)
    {
        randomPoint = GetRandomPointAroundCircle(initialPoint);
        UnityEngine.AI.NavMeshPath nmp = new UnityEngine.AI.NavMeshPath();
        while (!_agent.CalculatePath(randomPoint, nmp))
        {
            randomPoint = GetRandomPointAroundCircle(initialPoint);
        }
        _agent.SetDestination(randomPoint);
        if (!objectiveNotFoundTimerEnabled)
        {
            StartCoroutine(ObjectiveNotFound());
            objectiveNotFoundTimerEnabled = true;
        }
    }
    else if (!backToPlayer && RandomPointDestinationReached() && currentObjective != null)
    {
        randomPoint = GetRandomPointAroundObjective(currentObjective.transform.position);
        UnityEngine.AI.NavMeshPath nmp = new UnityEngine.AI.NavMeshPath();
        while (!_agent.CalculatePath(randomPoint, nmp))
        {
            randomPoint = GetRandomPointAroundObjective(currentObjective.transform.position);
        }
        _agent.SetDestination(randomPoint);
    }
```

Figure 7: Cleaners action 1

The first condition is in case that the current objective is null, which means they have not found where the first debris is. At the exact moment when they find a part of a debris, they already will have an objective: The Debris obstacle. So, in the case they have not found the objective (is null), they will move randomly inside a circle around the spawn initial position (*GetRandomPointAroundCircle()).*

The second condition is in the case that the current objective is not null. Thus, they will move randomly the whole objective (*GetRandomPointAroundObjective()*).

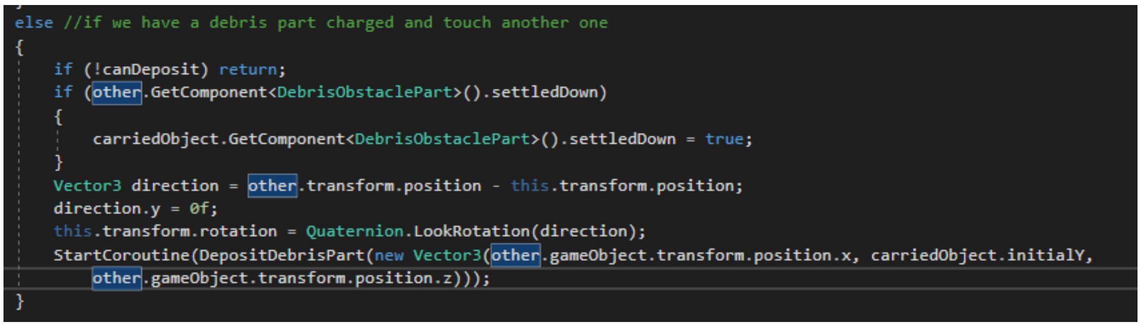


```
private void OnTriggerEnter(Collider other)
{
    if (!backToPlayer && other.GetComponent<DebrisObstaclePart>() && other.GetComponent<DebrisObstaclePart>().charger == null) //debris on the ground
    {
        if (currentObjective == null)
        {
            SetObjective(other.GetComponentInParent<Obstacle>());
        }
        if (carriedObject == null)
        {
            if (!canCharge) return;
            if (other.GetComponent<DebrisObstaclePart>().settledDown) return;

            carriedObject = other.GetComponent<DebrisObstaclePart>();
            Vector3 direction = other.transform.position - this.transform.position;
            direction.y = 0f;
            this.transform.rotation = Quaternion.LookRotation(direction);
            StartCoroutine(GrabDebrisPart());
        }
```

Figure 8: Cleaners action 2

The second action that the cleaners perform is to carry a debris part if they collide with it and they are not carrying another part in that moment. If their current objective was null, now they have an objective. We can see in Figure 8 the code of this.

In the case they were already charging a debris part and they collide with another one placed on the floor, they deposit it in that point. If the debris part in which the cleaner has collided was settled down, the carried one will be too. This means that this part will not be carried by any other cleaner anymore. It is shown in Figure 9.



```
else //if we have a debris part charged and touch another one
{
    if (!canDeposit) return;
    if (other.GetComponent<DebrisObstaclePart>().settledDown)
    {
        carriedObject.GetComponent<DebrisObstaclePart>().settledDown = true;
    }
    Vector3 direction = other.transform.position - this.transform.position;
    direction.y = 0f;
    this.transform.rotation = Quaternion.LookRotation(direction);
    StartCoroutine(DepositDebrisPart(new Vector3(other.gameObject.transform.position.x, carriedObject.initialY,
        other.gameObject.transform.position.z)));
}
```

Figure 9: Cleaners action 3

When they accomplish their work (checking that all the parts are *settleDown = true)*, the objective is completed.

Here is a short demo (Figure 10) about how the cleaners do these actions and how the cleaning behavior emerges. Keep in mind that the main purpose of this project was to get the emergency behaviors and apply them to multi-agent systems.

Figure 10: Cleaners demo

[Video 2]

3.3.2. Collectors complex behavior

The collectors have to collect some kind of resource from some "resource sources" and bring it to a collecting machine. In this case we are going to refer these sources as elixir stone. If the movement was completely random, it should take a long time to collect all the available elixir in the stones. Nonetheless, due to the emergency behavior created by the actions performed by the collectors, it is quite fast.

This complex behavior is based in the eusociality of the ants, explained at the beginning of this point.

The actions performed by the collectors are the following:

1. Move randomly until they find an elixir stone or a valid path of remains of elixir.
2. If they collide a stone, start collecting its elixir.
3. When the collecting process is done, go looking for the collecting machine. While they are moving looking for the collecting machine, they are leaving a trail of elixir (like the ant's pheromones).
4. If they collide a path of remains of elixir and they are carrying collected elixir, follow that trail in direction through the collecting machine. If they were not carrying elixir, follow that trail in direction through the stone.
5. When they find the collecting machine and they were carrying elixir, they deposit it in the machine and all the path they were leaving from the stone to the current position become a valid path so now the rest of explorers can follow that path if they find it.

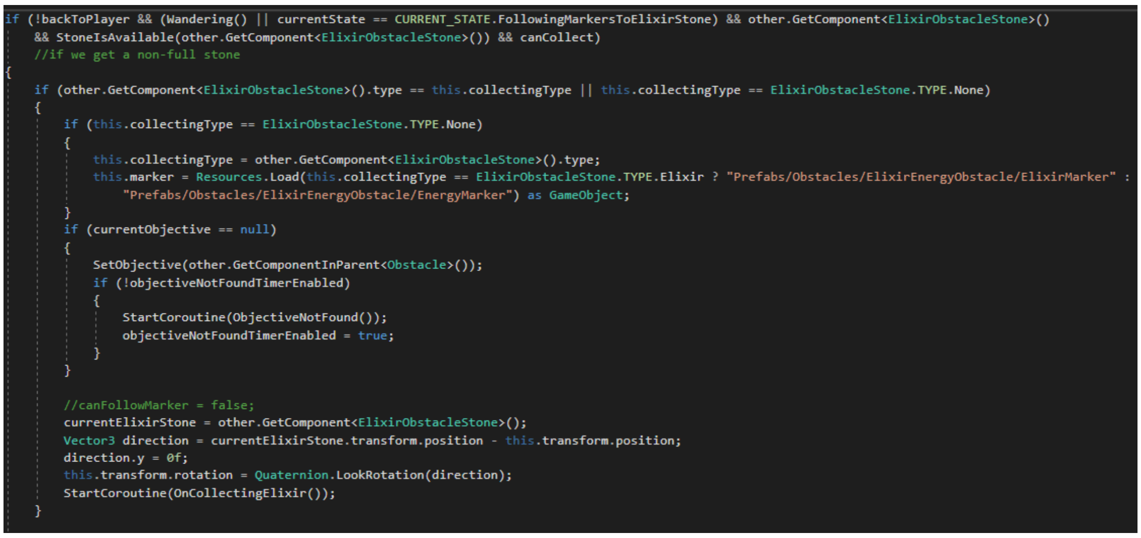
It is important to explain how the trail works. It starts as soon as the collecting process is done. While they move looking for the collecting machine in order to deposit the carried elixir, they are leaving a trail. In the moment they find the collecting machine, this trail become a valid trail. At this moment, there is a trail that communicates an elixir stone with the collecting machine. This means that every explorer that finds that trail can go direction to the stone or direction to the collecting machine. They just have to follow the trail.

Now, these actions are going to be explained showing some code so it will be clearly explained. First of all, must be known that the movement they follow while there is not any objective assigned is following the already seen function *GetRandomPointAroundCircle()* and if there is some objective and they are moving random they are following the *GetRandomPointAroundObjective().* It is important to remember that the objective is assigned as soon as they interact (collide or touch) with one of their obstacles. In this case, the Collecting Obstacle.

In order to simplify the code of the collectors, it has been decided to create CURRENT_STATE variable. It will help the development of all the actions. We are going to see which values can have this variable.

- *GoingToCollectorMachine*: this is the state of the collectors when they collect elixir and they are looking for the elixir collecting machine.
- *WanderingAroundObjective*: when they are looking for an elixir stone. Here they are using the function *WanderingAroundObjective()*.
- *WanderingAroundCircle*: when they are looking for an elixir stone but they do not have an objective assigned. They are using the function *WanderingAroundCircle*().
- *CollectingElixir*: when they are collecting elixir. The time it takes depends on a emergency behavior parameter.
- *FollowingMarkersToElixirStone*: when they were wandering around and they find a trail of elixir.
- *FollowingMarkersToCollectorMachine*: when they were looking for the collecting machine and they find a trail of elixir.
- *DepositingElixir*: when they find the collecting machine.
- *GoingToPlayer*: when they finish their work or the player orders the collectors to come back.

There is an important parameter in the configuration of the collectors. It is called *BoogiesKnowCollectorMachinePosition*. This is a Boolean parameter. In the case it is true, as soon as they finish collecting the elixir, they will go to the collecting machine. In the case it is false, when they complete the collecting process, they start moving randomly because they do not know where is the collecting machine, they would act like if they were blind. In the demo at the end of this point, it is shown in the case they do not know where the machine is located. In the author's opinion, this case allows more emergent behaviors because they will arrive if they follow the trails. Otherwise, the trails are not as useful.



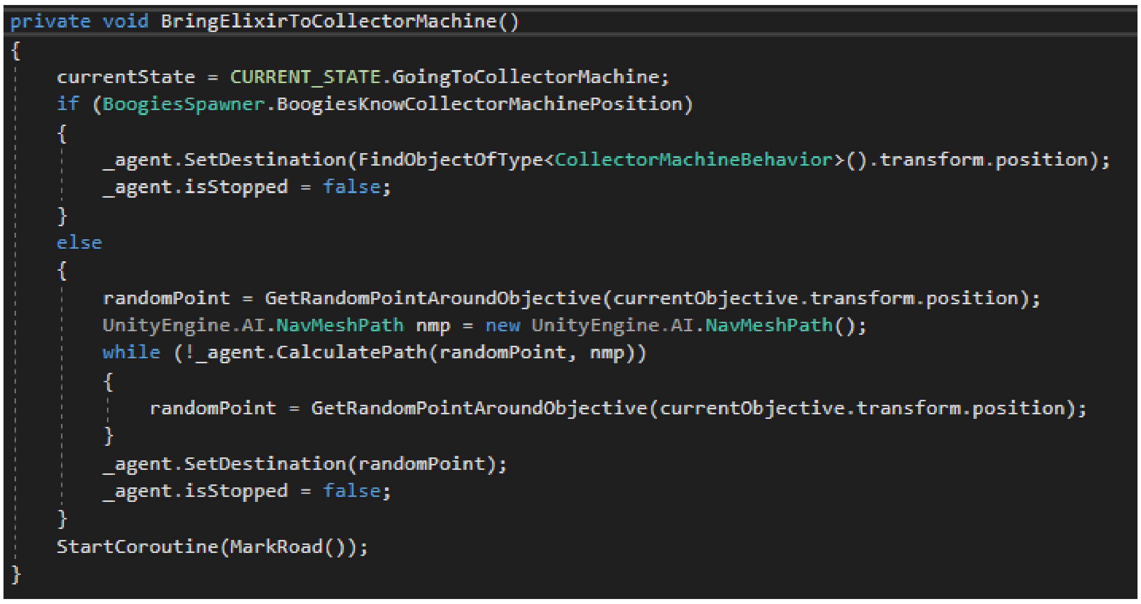
```
if (!backToPlayer && (Wandering() || currentState == CURRENT_STATE.FollowingMarkersToElixirStone) && other.GetComponent<ElixirObstacleStone>()
    && StoneIsAvailable(other.GetComponent<ElixirObstacleStone>()) && canCollect)
    //if we get a non-full stone
{
    if (other.GetComponent<ElixirObstacleStone>().type == this.collectingType || this.collectingType == ElixirObstacleStone.TYPE.None)
    {
        if (this.collectingType == ElixirObstacleStone.TYPE.None)
        {
            this.collectingType = other.GetComponent<ElixirObstacleStone>().type;
            this.marker = Resources.Load(this.collectingType == ElixirObstacleStone.TYPE.Elixir ? "Prefabs/Obstacles/ElixirEnergyObstacle/ElixirMarker" :
                "Prefabs/Obstacles/ElixirEnergyObstacle/EnergyMarker") as GameObject;
        }
        if (currentObjective == null)
        {
            SetObjective(other.GetComponentInParent<Obstacle>());
            if (!objectiveNotFoundTimerEnabled)
            {
                StartCoroutine(ObjectiveNotFound());
                objectiveNotFoundTimerEnabled = true;
            }
        }

        //canFollowMarker = false;
        currentElixirStone = other.GetComponent<ElixirObstacleStone>();
        Vector3 direction = currentElixirStone.transform.position - this.transform.position;
        direction.y = 0f;
        this.transform.rotation = Quaternion.LookRotation(direction);
        StartCoroutine(OnCollectingElixir());
    }
```

Figure 11: Collectors action 2

As we can see in Figure 11, when they collide a stone, if the stone still has elixir available, they start collecting it. And in case that the objective has not been already assigned, they assign it. There are some conditions in order to be able to start collecting. These are the following:

- The Stone type must be the same than our collecting type. Remember that this could be Energy or Elixir.
- The collector is not carrying elixir in that moment.
- The collector is wandering around or following the trail in the stone direction.
- The stone does not exceed the stablished number of maximum collectors.

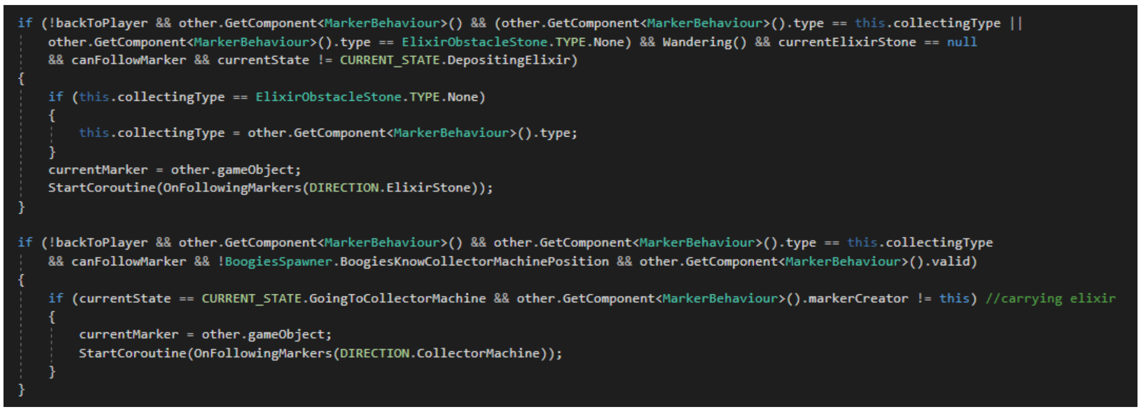
In Figure 12, we can see the action performed when this collecting process is finished. In the case the collectors know where the collecting machine is, they set their destination where it is located. Otherwise, they start moving randomly until they find the collecting machine or a trail of elixir which they can follow direction to the collecting machine.



```
private void BringElixirToCollectorMachine()
{
    currentState = CURRENT_STATE.GoingToCollectorMachine;
    if (BoogiesSpawner.BoogiesKnowCollectorMachinePosition)
    {
        _agent.SetDestination(FindObjectOfType<CollectorMachineBehavior>().transform.position);
        _agent.isStopped = false;
    }
    else
    {
        randomPoint = GetRandomPointAroundObjective(currentObjective.transform.position);
        UnityEngine.AI.NavMeshPath nmp = new UnityEngine.AI.NavMeshPath();
        while (!_agent.CalculatePath(randomPoint, nmp))
        {
            randomPoint = GetRandomPointAroundObjective(currentObjective.transform.position);
        }
        _agent.SetDestination(randomPoint);
        _agent.isStopped = false;
    }
    StartCoroutine(MarkRoad());
}
```

Figure 12. Collectors action 3

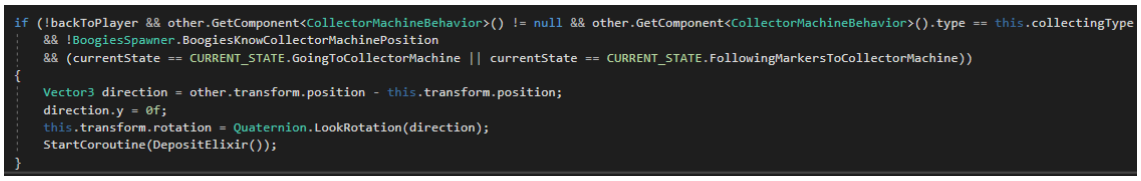
Figure 13 shows how the collectors check which direction taking in the case they find a trail of elixir in the floor. This decision will depend on if they are carrying elixir in that moment or not. In affirmative case, they will take the direction towards the collecting machine (because they need to deposit the carried elixir). If not, the direction will be towards the elixir stone (because they need to collect elixir).



```
if (!backToPlayer && other.GetComponent<MarkerBehaviour>() && (other.GetComponent<MarkerBehaviour>().type == this.collectingType ||
    other.GetComponent<MarkerBehaviour>().type == ElixirObstacleStone.TYPE.None) && Wandering() && currentElixirStone == null
    && canFollowMarker && currentState != CURRENT_STATE.DepositingElixir)
{
    if (this.collectingType == ElixirObstacleStone.TYPE.None)
    {
        this.collectingType = other.GetComponent<MarkerBehaviour>().type;
    }
    currentMarker = other.gameObject;
    StartCoroutine(OnFollowingMarkers(DIRECTION.ElixirStone));
}

if (!backToPlayer && other.GetComponent<MarkerBehaviour>() && other.GetComponent<MarkerBehaviour>().type == this.collectingType
    && canFollowMarker && !BoogiesSpawner.BoogiesKnowCollectorMachinePosition && other.GetComponent<MarkerBehaviour>().valid)
{
    if (currentState == CURRENT_STATE.GoingToCollectorMachine && other.GetComponent<MarkerBehaviour>().markerCreator != this) //carrying elixir
    {
        currentMarker = other.gameObject;
        StartCoroutine(OnFollowingMarkers(DIRECTION.CollectorMachine));
    }
}
```

Figure 13: Collectors action 4

There is just one action left to be explained and it is the simplest of all. When the collectors are in the collector machine and they were carrying some elixir, they deposit it in the machine. This is shown in Figure 14.



```
if (!backToPlayer && other.GetComponent<CollectorMachineBehavior>() != null && other.GetComponent<CollectorMachineBehavior>().type == this.collectingType
    && !BoogiesSpawner.BoogiesKnowCollectorMachinePosition
    && (currentState == CURRENT_STATE.GoingToCollectorMachine || currentState == CURRENT_STATE.FollowingMarkersToCollectorMachine))
{
    Vector3 direction = other.transform.position - this.transform.position;
    direction.y = 0f;
    this.transform.rotation = Quaternion.LookRotation(direction);
    StartCoroutine(DepositElixir());
}
```

Figure 14: Collectors action 5

For ending this part, it is important to see the following demonstration of how the complex collecting behavior emerge of these simple actions explained above.

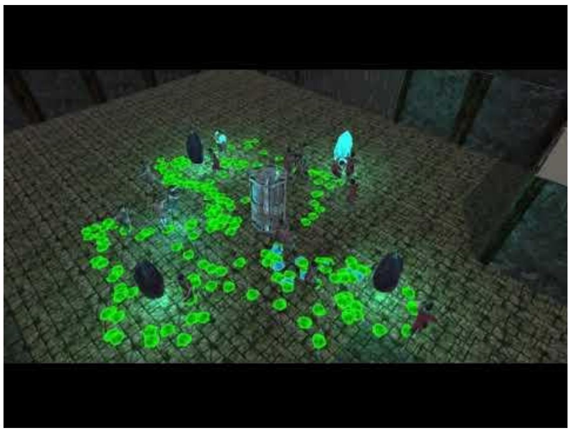


Figure 15: Collectors demo

[Video 3]

3.3.3. Explorers complex behavior

The explorers have to explore a path between two points through a multipath. Their objective is the multipath explained above so they need to find the combination of corridors in order to pass it. Nonetheless, as they have the purpose of helping the player, they need to communicate to the player which is that combination in order to the player be able to complete it.

The actions performed by the explorers are the following:

1. Move randomly until they find the beginning of a corridor.
2. If there is no clue at the beginning of the multipath that indicates which corridor is the correct one, they select randomly a corridor. Otherwise, they go through the correct corridor.
3. If that corridor was the correct one, when they pass through the exit corridor door, they drop a clue which indicates that corridor is the correct one.
4. In case that was not the correct corridor, they restart their position to the beginning of the multipath.
5. If they find a clue in the floor, they pick it and they place at the beginning of the multipath.
6. When they are in the beginning of the multipath and they were carrying a clue, they deposit it.

These are the simple actions performed by the explorers and when a multi-agent system performs it, a complex exploring behavior emerges and the result of it is the full combination of paths indicated by the clues position at the beginning of the multipath. Before start explaining the actions showing the code, it is going to be shown what are the clues and how they work.

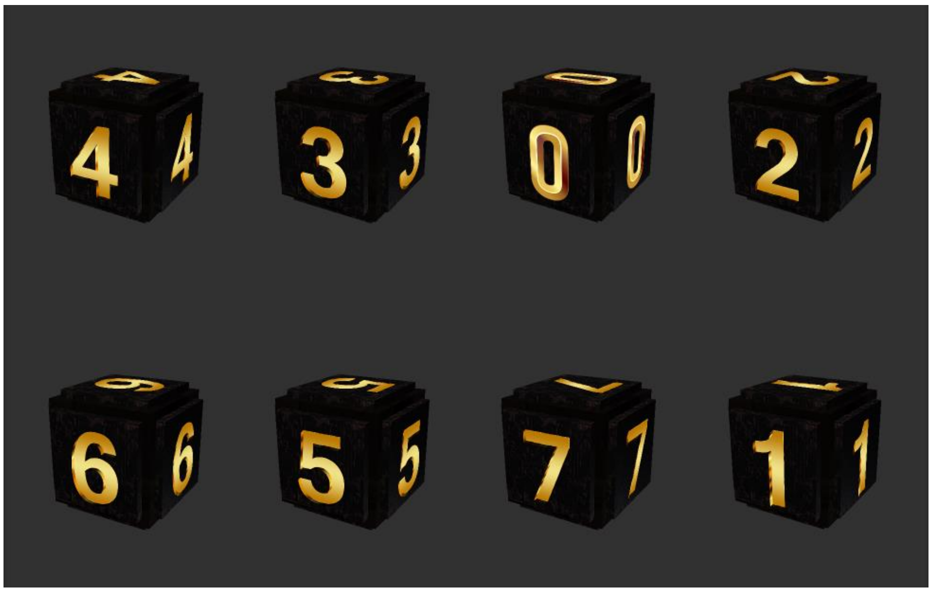


Figure 16: Clues

As we can see in Figure 16, the Clues are objects that contains a number. This number indicates the number of the corridor. As it has been said before, these clues are dropped by the explorers and taken by them too. When they pick a clue, they go to the beginning of the multipath carrying it and they place it in a point. Thus, when some clues are placed, they will form a sequence of numbers which is the exact combination. In Figure 17 we can see the final result of the combination 2-3-4-6-0.

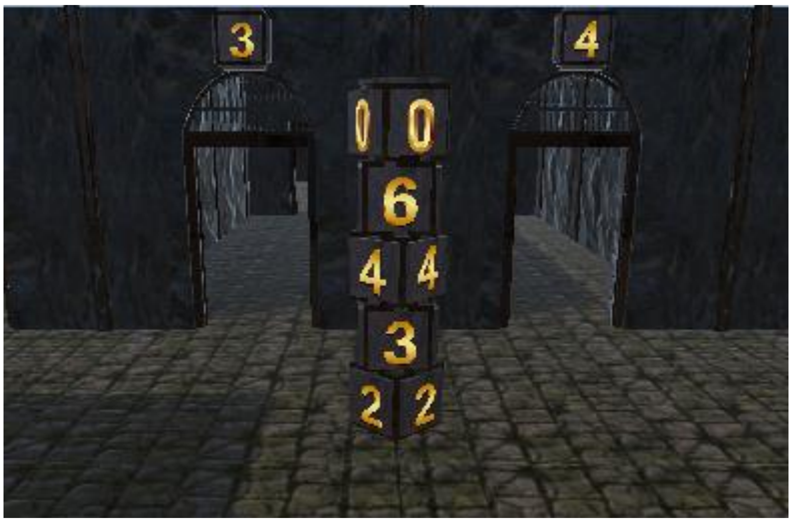
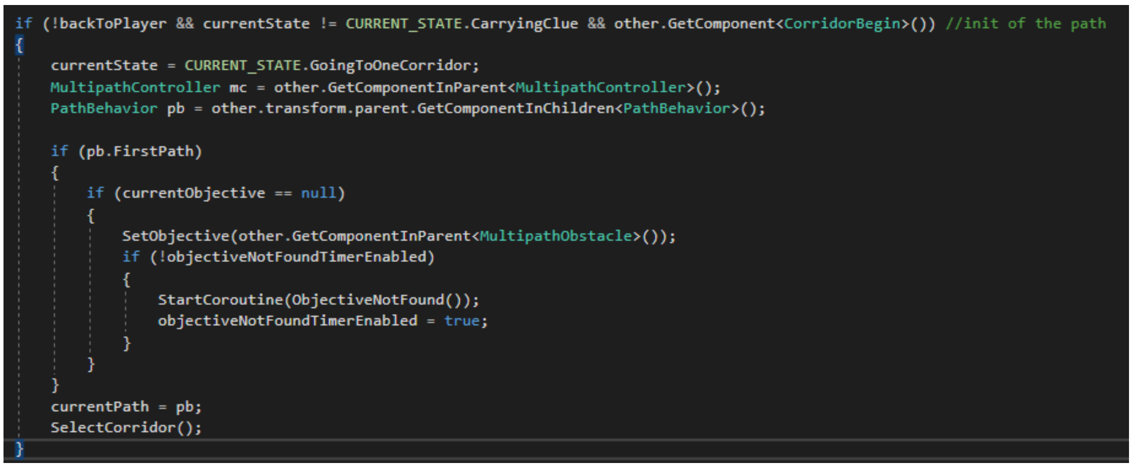


Figure 17: All clues placed

So, now is a good moment to focus on the behaviors of the cleaners, because of there are some actions to be performed. The explorers have a simple parameter called CURRENT_STATE which indicates which action is performing at any moment. Now, we are going to have a look of it.

- *FindingMultipathBegin:* as soon as they are spawned, they move around the circle using the function seen before: *GetRandomPointAroundCircle().*
- *GoingToOneCorridor:* when they find the beginning of a set of corridors, they go to one of the corridors.
- *GoingToEndPath*: when they are passing through a corridor, they need to go to the end of the same.
- *GoingToInitNextPath*: after reaching the end of a corridor, it is moment to go to the next path.
- *CarryingClue:* when they find a clue and carry it to the beginning of the Multipath.
- *GoingToEnd*: when they pass through the last corridor door and there are no more corridors.
- *GoingToPlayer:* when the player passes the multipath or the player orders the explorers to come back.

Once understood this, the first action is the movement one. As soon as they are spawned, they move randomly around the circle (state *FindingMultipathBegin*), exactly as the cleaners and the collectors do. They will change their destination in the moment they find the beginning of the multipath. As we can see in Figure 18, they will do it when collide with the *CorridorBegin* trigger object. When they do it, their state will change to *GoingToOneCorridor.*



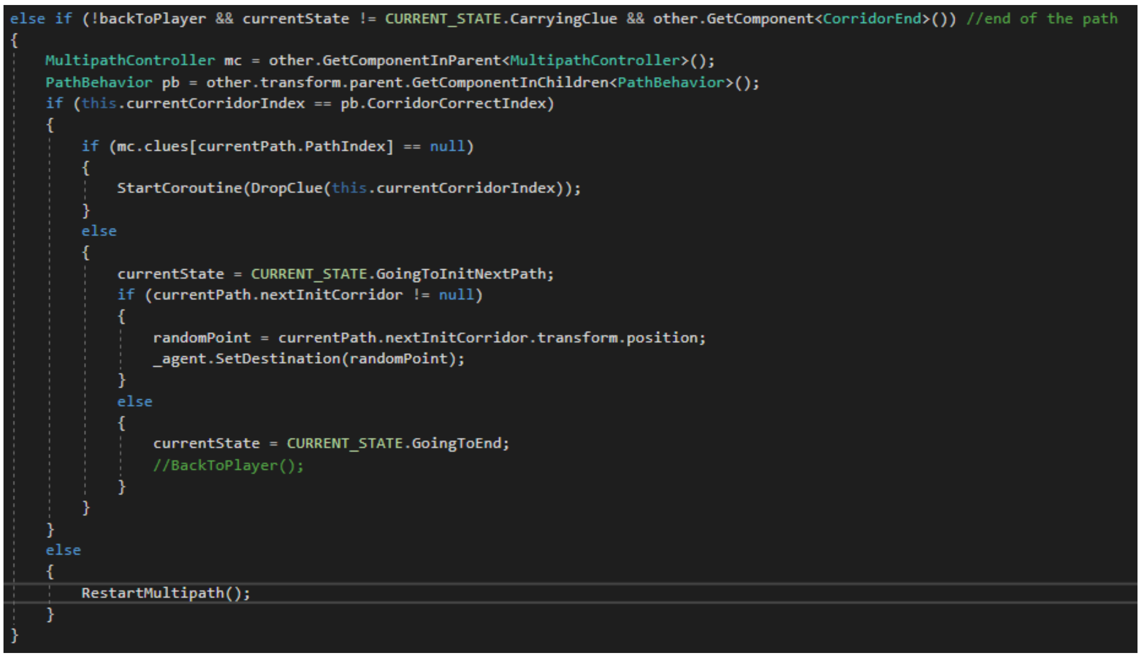
```
if (!backToPlayer && currentState != CURRENT_STATE.CarryingClue && other.GetComponent<CorridorBegin>()) //init of the path
{
    currentState = CURRENT_STATE.GoingToOneCorridor;
    MultipathController mc = other.GetComponentInParent<MultipathController>();
    PathBehavior pb = other.transform.parent.GetComponentInChildren<PathBehavior>();

    if (pb.FirstPath)
    {
        if (currentObjective == null)
        {
            SetObjective(other.GetComponentInParent<MultipathObstacle>());
            if (!objectiveNotFoundTimerEnabled)
            {
                StartCoroutine(ObjectiveNotFound());
                objectiveNotFoundTimerEnabled = true;
            }
        }
    }
    currentPath = pb;
    SelectCorridor();
}
```

Figure 18: Explorers action 2

As we can see in Figure 18, there are two cases when they find the *CorridorBegin.* On the one hand, if it was the first path (beginning of the multipath) they will assign the multipath as the objective so from now to the end, when they move randomly they will move using the function *GetRandomPointAroundObjective().* After doing this, they select a corridor. This function is going to be explained now.

When the explorers have to select a corridor, they see if there is any clue that indicates which corridor to select. If there is no clue, they pick a corridor randomly. It is important to pay attention to the value *probabilityFollowClue*, this is a parameter of the emergency behavior. The higher the parameter, the more probable will be to go to the clue's corridor. If this parameter is 0, clues would be useless.

In the next action, it is going to be explained how the explorers drop a clue in the case that was the correct corridor in the combination. They do it when they touch the *CorridorEnd* trigger and they were not carrying a clue.

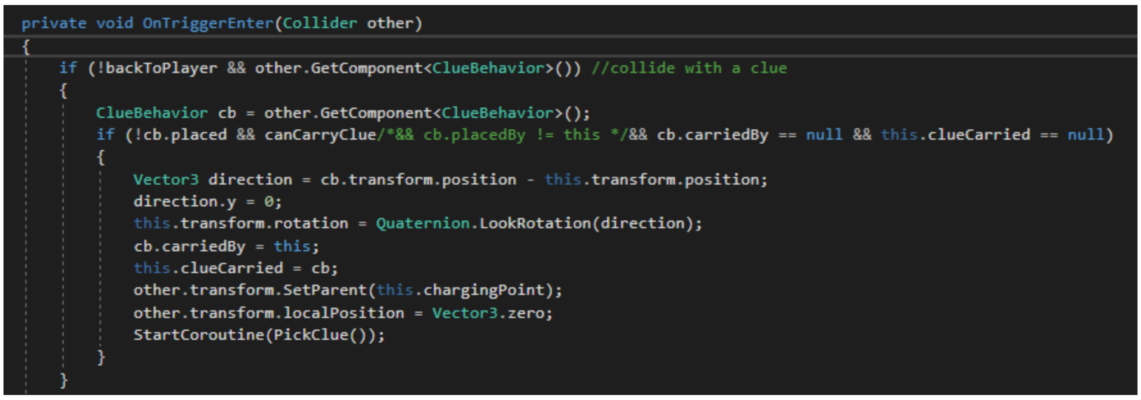


```
else if (!backToPlayer && currentState != CURRENT_STATE.CarryingClue && other.GetComponent<CorridorEnd>()) //end of the path
{
    MultipathController mc = other.GetComponentInParent<MultipathController>();
    PathBehavior pb = other.transform.parent.GetComponentInChildren<PathBehavior>();
    if (this.currentCorridorIndex == pb.CorridorCorrectIndex)
    {
        if (mc.clues[currentPath.PathIndex] == null)
        {
            StartCoroutine(DropClue(this.currentCorridorIndex));
        }
        else
        {
            currentState = CURRENT_STATE.GoingToInitNextPath;
            if (currentPath.nextInitCorridor != null)
            {
                randomPoint = currentPath.nextInitCorridor.transform.position;
                _agent.SetDestination(randomPoint);
            }
            else
            {
                currentState = CURRENT_STATE.GoingToEnd;
                //BackToPlayer();
            }
        }
    }
    else
    {
        RestartMultipath();
    }
}
```

Figure 19: Explorers action 3 and 4

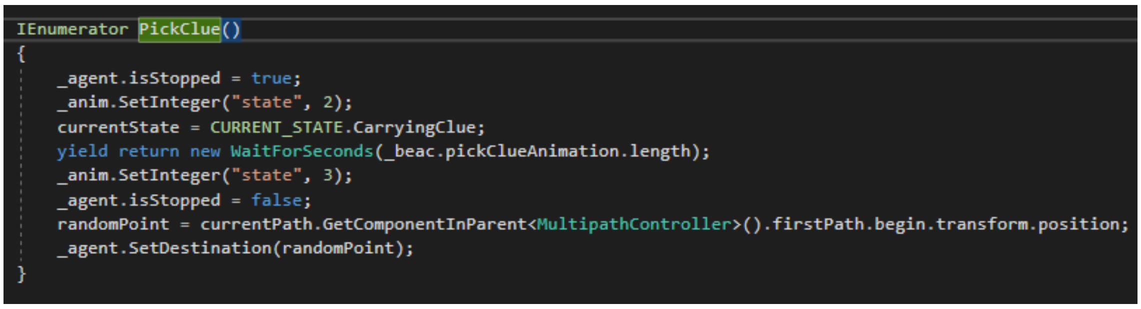
As it can be seen in Figure 19, if they collide with *CorridorEnd,* it is checked if that was the correct corridor. On the one hand (action 3), if it is the correct corridor, they check if there already is a clue of that corridor at the beginning. In that case they set their destination to the next corridors. Otherwise, they place a clue which indicates that it was the correct corridor and keep moving randomly to the next corridor. On the other hand, if it is not the correct corridor (action 4), their position is restarted to the beginning of the multipath.

There is just one action left, when they pick a clue on the floor and bring it to the beginning of the multipath.



```
private void OnTriggerEnter(Collider other)
{
    if (!backToPlayer && other.GetComponent<ClueBehavior>()) //collide with a clue
    {
        ClueBehavior cb = other.GetComponent<ClueBehavior>();
        if (!cb.placed && canCarryClue/*&& cb.placedBy != this */&& cb.carriedBy == null && this.clueCarried == null)
        {
            Vector3 direction = cb.transform.position - this.transform.position;
            direction.y = 0;
            this.transform.rotation = Quaternion.LookRotation(direction);
            cb.carriedBy = this;
            this.clueCarried = cb;
            other.transform.SetParent(this.chargingPoint);
            other.transform.localPosition = Vector3.zero;
            StartCoroutine(PickClue());
        }
    }
```

Figure 20: Explorers action 5 part 1

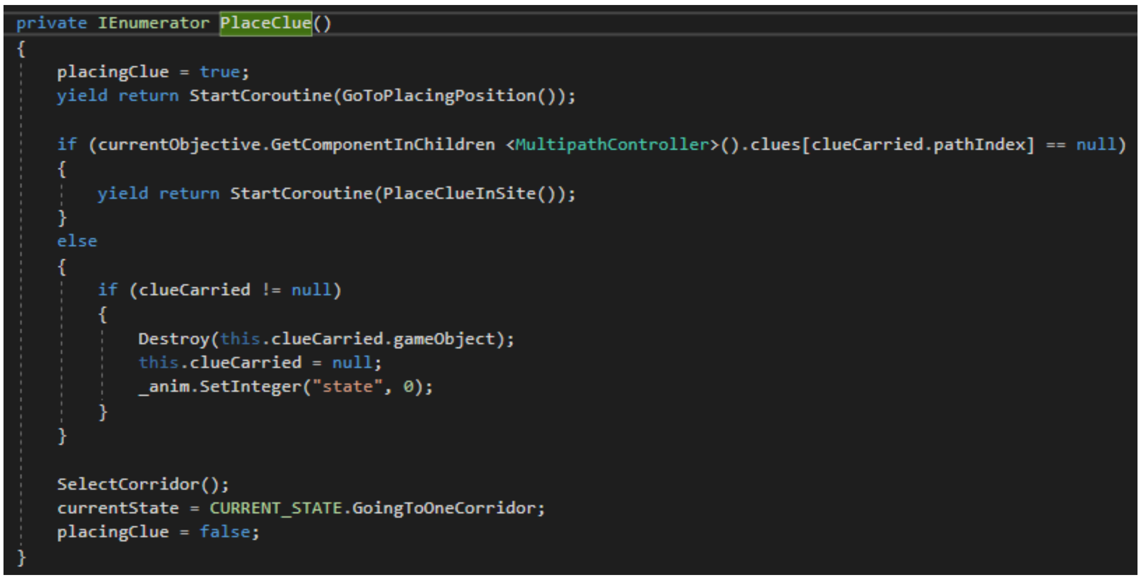


```
IEnumerator PickClue()
{
    _agent.isStopped = true;
    _anim.SetInteger("state", 2);
    currentState = CURRENT_STATE.CarryingClue;
    yield return new WaitForSeconds(_beac.pickClueAnimation.length);
    _anim.SetInteger("state", 3);
    _agent.isStopped = false;
    randomPoint = currentPath.GetComponentInParent<MultipathController>().firstPath.begin.transform.position;
    _agent.SetDestination(randomPoint);
}
```

Figure 21: Explorers action 5 part 2

As it can be seen in Figure 20, when they find a clue in the floor, they do some assignments and start the coroutine shown in Figure 21. In *PickClue()* we can see that they set as destination the beginning of the multipath.

In order to conclude this part, it is important to see how they place the Clue in the beginning. We can see it in Figure 22. First, what they do is to move to the "placing position". Once there, they check there is not a clue of that corridor already placed. In that case, they place it. Otherwise, the clue is destroyed.



```
private IEnumerator PlaceClue()
{
    placingClue = true;
    yield return StartCoroutine(GoToPlacingPosition());

    if (currentObjective.GetComponentInChildren <MultipathController>().clues[clueCarried.pathIndex] == null)
    {
        yield return StartCoroutine(PlaceClueInSite());
    }
    else
    {
        if (clueCarried != null)
        {
            Destroy(this.clueCarried.gameObject);
            this.clueCarried = null;
            _anim.SetInteger("state", 0);
        }
    }

    SelectCorridor();
    currentState = CURRENT_STATE.GoingToOneCorridor;
    placingClue = false;
}
```

Figure 22: Explorers action 6

For ending with explorers complex behavior. In Figure 23 it is shown a demo of how they traverse a multipath.

Figure 23: Explorers demo

[Video 4]

3.4. Squad system

In this point, it is going to be shown and explained the squad system built for this project. It is a multi-agent system formed by some entities with different roles. Each role has a different set of actions. Nonetheless, they all act as a squad and the emergency behaviors appear when two or more squads face each other.

It has been decided to separate this point from the last because in the last point the multi-agent systems were formed by some entities of the same type. Here, although they all are "wrestlers", each of them has a different role. Before starting the explanation, we are going to identify these roles.

- Distance: they attack from afar with low damage and are faster than the rest.
- Close: they attack close-combat with low damage.
- Giant: they attack close-combat with high damage. Besides, their help is higher than the rest.
- Commander: he attacks close-combat with high-damage. He coordinates the whole squad.

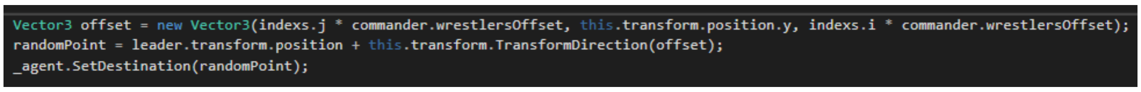
First of all, is is going to be explained how this point is going to be divided. Firstly, the main structure of the squad system will be explained. In the second place, the whole set of actions that the squad can perform will be shown. For ending this point, the battle system will be explained with demos and code captures.

3.4.1. Squad system structure

The squad internally is represented as a matrix. Each position of the matrix has two indexes (typically known in matrix as rows and columns). The Commander has the indexes 0,0. The rest of the wrestlers are positioned relative to this Commander. Besides, there is a parameter called *offsetBetweenIndexs* that is the separation between these wrestlers. So, calculating the position of each of them is as simple as follow this equation:

$$\begin{aligned} Vector3\ position \\ &= new\ Vector3(commander.position.x\ +\ indexes.columns \\ &* offsetBetweenIndexs, Yposition, commander.position.z \\ &+ indexes.rows\ *\ offsetBetweenIndexs; \end{aligned}$$

This is the same than calculating first the offset position and add it to the commander's position. It can be seen in Figure 24.



```
Vector3 offset = new Vector3(indexs.j * commander.wrestlersOffset, this.transform.position.y, indexs.i * commander.wrestlersOffset);
randomPoint = leader.transform.position + this.transform.TransformDirection(offset);
_agent.SetDestination(randomPoint);
```

Figure 24: Position calculation

The rotation of each troop is the same of the commander, so it gives more impression of squad coordination. Besides, each entity inside the squad has two references. One is for the Commander, and the other one is for the Leader. At the beginning they both Commander and Leader are the same. Nonetheless, there are some actions (which will be explained in the next point) that can change this. They move relative to the Leader.

There is a reason because there exists both references: the player can be covered by the squad, so the squad will move relative to the player. In this case, the player would be the Leader and the Commander would be another wrestler covering the player. Besides, any wrestler can act as the Leader so it is necessary to have a distinction between the Leader and the Commander.

3.4.2. Squad system actions

This point is going to be divided into three groups; the actions performed by a squad, the actions performed by a wrestler inside a squad and the actions performed by a wrestler without squad. All these actions have a lot of code so it will be referenced as Annex 4, explained at the end of the document.

3.4.2.1. Squad actions

This set of actions is shown when the user clicks with the left button of the mouse at any member of the squad.

- <u>Move</u>: the whole squad will move to a point. Internally, the commander selects the destiny point and each wrestler will select his/her position relative to that point. This is, they are going to select the position where they will be once the squad arrives. As soon as it is selected, they set their destination to that point.

- Cover: the whole squad will cover a selected body. For this to be possible, a position of the squad must be assigned to what it is going to cover. This position is selected in the squad in-game creator. It will be seen later. The squad can cover everything of the type *InteractableBody*. This includes the player, the clues, the elixir stone, all the entities, etc. When they cover them, they set that body as the Leader, because now the whole squad is going to move relative to this body. They will have to adapt their velocity to the covered body's.
- Rotate: the leader rotates so the whole squad will position relative to him.
- Back to player: if the player orders the squad to go back to him/her, they will go.
- Change formation: this is a very interesting action. There can be stored some configurations of the squad (the concept configuration will be seen later). This is, different formations of each entity inside the squad. The squad will be able to change the formation to another as long as both have the same number of wrestlers in each role both allow the *Cover action* or not.

3.4.2.2. Individual entity being in a squad actions

This set of actions is shown when the user clicks with the right button of the mouse at a wrestler inside a squad.

- Follow player: the wrestler will follow the player moving around a circle with the player as center. As soon as they start following the player, they break the formation.
- Change position: the wrestler is able to change his position with another wrestler of the same squad. This is, they exchange positions. This is very useful if one of them has low health, etc.
- Break the formation: the wrestler can break the formation in the squad. This means, he will no longer be in that squad. He will move with total freedom and will adopt the set of actions explained in the next point.
- Assign as leader: the wrestler will now be the leader. This means the whole squad is going to move and act relative to him.

3.4.2.2. Individual entity without squad actions

This set of actions is shown when the user clicks with the left button a wrestler without a squad.

- Join a squad: the wrestler is able to join a squad as long as that squad have a free position.
- Follow player: this works exactly as the *Follow player* action explained above.
- Move: the wrestler can move totally free without depending of a leader or a commander. He sets his destination in the designated point and he move towards it.
- Back to player: the wrestler will go to the player if him/her orders it.

As in the case of the explorers and the collectors, in order to simplify some code, a simple set of states have been created and each wrestler has one of them active in every frame. These are the following.

- *OnSquadObserving*: wrestler is inside a squad and the squad is not moving.
- *OnSquadAttacking:* the squad is facing another squad or individual wrestlers.
- *OnSquadCovering:* the squad is not moving and it is covering a body.
- *OnSquadCoveringMoving*: the squad is moving while covers a body.
- *OnSquadMoving:* the squad is moving.
- *AloneObserving*: a wrestler without squad is not moving or attacking.
- *AloneAttacking:* a wrestler without squad is attacking.
- *AloneFollowingPlayer:* a wrestler without squad is following player.
- *AloneMoving:* a wrestler without a squad moving.
- *BackToPlayer:* wrestler is going back to the player.

3.4.3. Squad battles system

At this point we will explain how the squad battles work. Some code will be showed and some demos too. First of all, it should be known that in order to two squads can face each other, they need to be from different teams. So, every wrestler has a parameter called *team.* We are going to divide this point into three parts: detection phase, attack phase and emergent changes.

3.4.3.1. Detection system

Each of the wrestlers has a system dedicated to detect if there is any wrestler from the other team in front of them. Now, it is going to be explained this detections system.

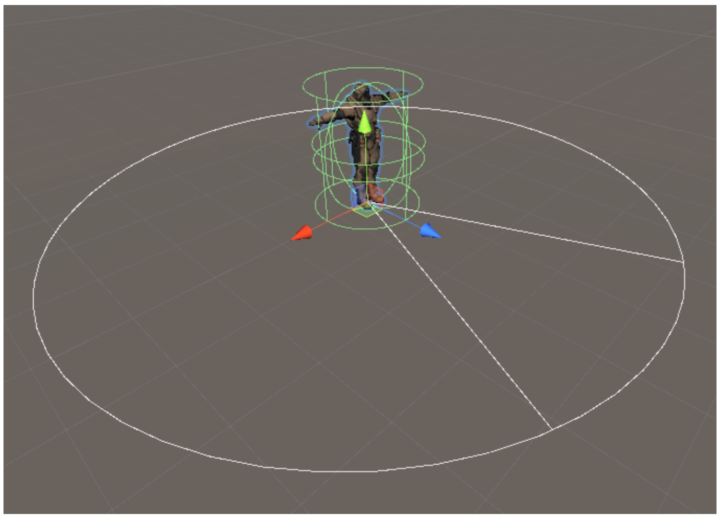
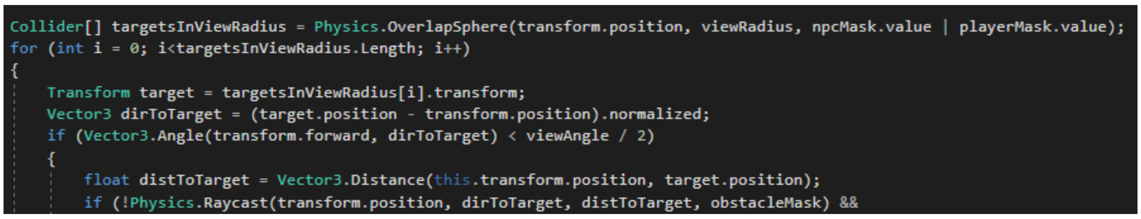


Figure 25: Detection system

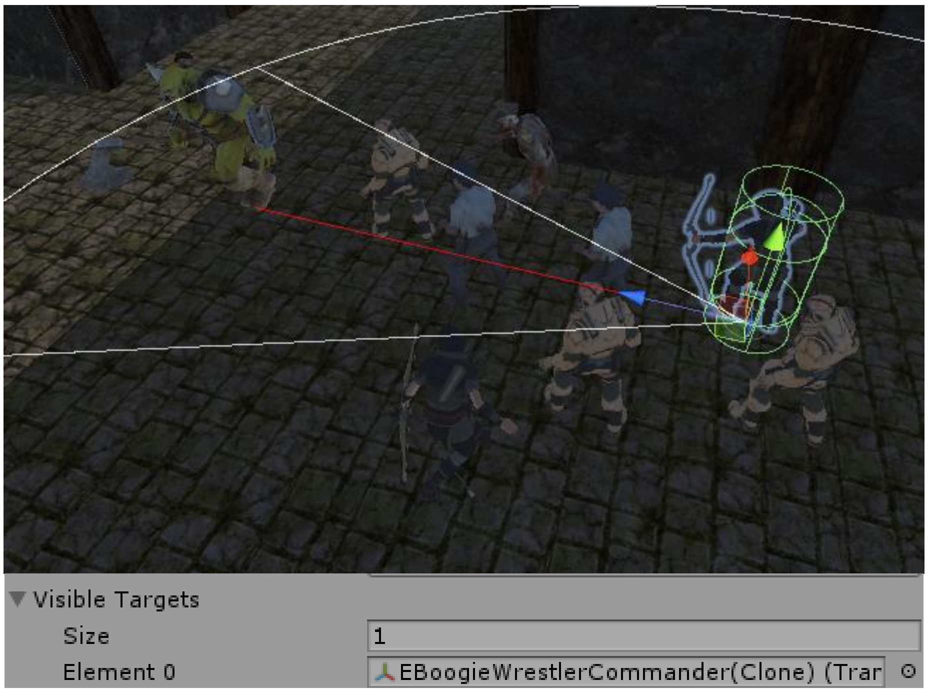
As we can see in Figure 25, they have a *viewRadius* which is the circle around him. This is the minimum area of detection. Besides, there is a view angle, which is the field of view called *viewAngle.* So, every wrestler from the other team that enters in that angle at a maximum distance of that circle's radius will be added to a list called *visibleTargets.*

Besides, there is one more thing that should be known: the masks. They will just detect those which are in the NPC mask (*npcMask)*. And if they are behind an object with the obstacle mask (*obstacleMask)* they will not be detected. For doing this, *Physics.OverlapSphere* has been used. It detects all the colliders with the specified mask inside a sphere of given radius. Once the colliders are taken, they are "raycasted" using *Physics.Raycast* in order to see if a ray between us and the target collides with any obstacle. If it does, the target is not added because this means there is an obstacle between the wrestler and the target. In Figure 26 we can see how that code works.



```
Collider[] targetsInViewRadius = Physics.OverlapSphere(transform.position, viewRadius, npcMask.value | playerMask.value);
for (int i = 0; i<targetsInViewRadius.Length; i++)
{
    Transform target = targetsInViewRadius[i].transform;
    Vector3 dirToTarget = (target.position - transform.position).normalized;
    if (Vector3.Angle(transform.forward, dirToTarget) < viewAngle / 2)
    {
        float distToTarget = Vector3.Distance(this.transform.position, target.position);
        if (!Physics.Raycast(transform.position, dirToTarget, distToTarget, obstacleMask) &&
```

Figure 26: Targets detection



Visible Targets
Size 1
Element 0 EBoogieWrestlerCommander(Clone) (Trar

Figure 27: Distance detecting Commander

The detection system is inspired in a detection system made by Sebastian Lague and modified and applied to the need of this project. Link of the video will be referenced at the end of the document. (Lague, 2015)

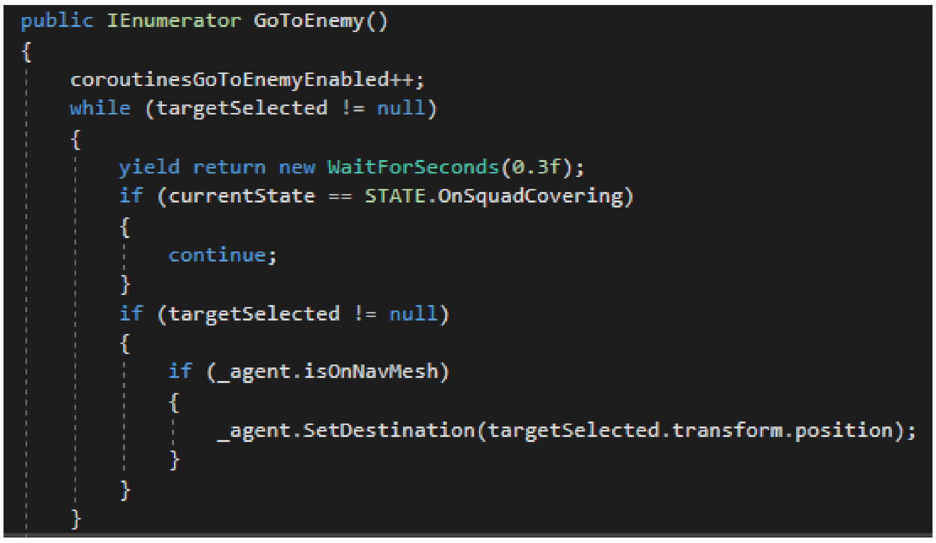
3.4.3.2. Attack system

Once *visibleTargets* has one or more elements, they select (this selection process will be explained in the next point) one of them as a *wrestlerSelected*. This selection is performed as follows: each role inside the squad has a list of preferences. This list is full with the roles. For example, the Distance role has the next preferences list: Commander, Giant, Distance and Close.

These preferences are sort by most preferred to less. This means that if this wrestler has inside his visible targets list a Commander, a Giant and a Close, he will choose the Commander because it is the most preferred by this role in this squad. It must be clear that these preferences for each role depend on the configuration of each squad. This will be explained in next points. The selection process is shown in Annex 1.

So, once they have a wrestler selected (a target to attack) some coroutines are started. These are the following:

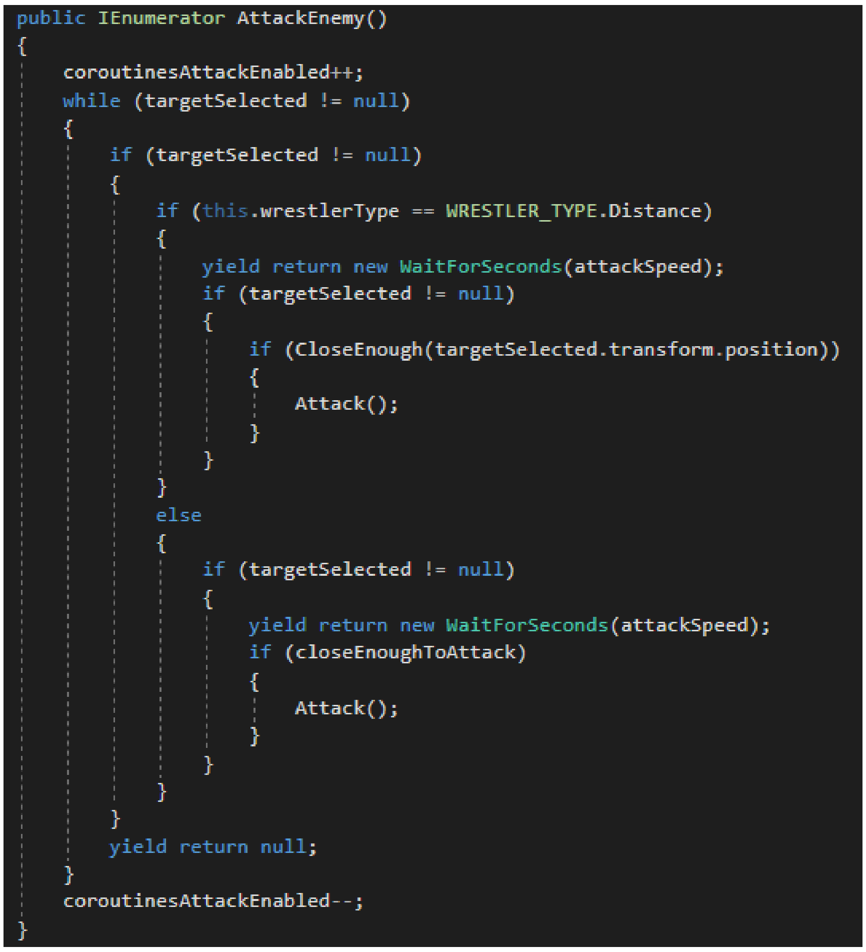
- <u>*GoToEnemy*</u>: set the destination of the wrestler where the target selected is located. It will be updating the destination every frame so he will be following every time while the selected is chosen as a target. It is shown in Figure 28.



```
public IEnumerator GoToEnemy()
{
    coroutinesGoToEnemyEnabled++;
    while (targetSelected != null)
    {
        yield return new WaitForSeconds(0.3f);
        if (currentState == STATE.OnSquadCovering)
        {
            continue;
        }
        if (targetSelected != null)
        {
            if (_agent.isOnNavMesh)
            {
                _agent.SetDestination(targetSelected.transform.position);
            }
        }
    }
```

Figure 28: GoToEnemy()

- *AttackPlayer*: it checks every frame if the wrestler is close enough to the selected target. If it is close enough then he attacks. This parameter to detect if it is close enough to attack or not depends on each wrestler. For example, it will not be the same for those who attack on a close-combat mode and for the distances, who attacks with a long range. This can be seen in Figure 29.
- *ChooseEnemyObjective:* this is a coroutine that brings variety to the battle. It depends of some parameters like *timeToFindTargets, probChangeObjective*, etc. So, if these parameters are well-adjusted the enemy will not be following and attacking the same enemy the whole time. He/she will vary and change the target. What it simply does is to select another player from the visible targets or select a target by emergency. This will be explained in the following points.



```
public IEnumerator AttackEnemy()
{
    coroutinesAttackEnabled++;
    while (targetSelected != null)
    {
        if (targetSelected != null)
        {
            if (this.wrestlerType == WRESTLER_TYPE.Distance)
            {
                yield return new WaitForSeconds(attackSpeed);
                if (targetSelected != null)
                {
                    if (CloseEnough(targetSelected.transform.position))
                    {
                        Attack();
                    }
                }
            }
            else
            {
                if (targetSelected != null)
                {
                    yield return new WaitForSeconds(attackSpeed);
                    if (closeEnoughToAttack)
                    {
                        Attack();
                    }
                }
            }
        }
        yield return null;
    }
    coroutinesAttackEnabled--;
}
```

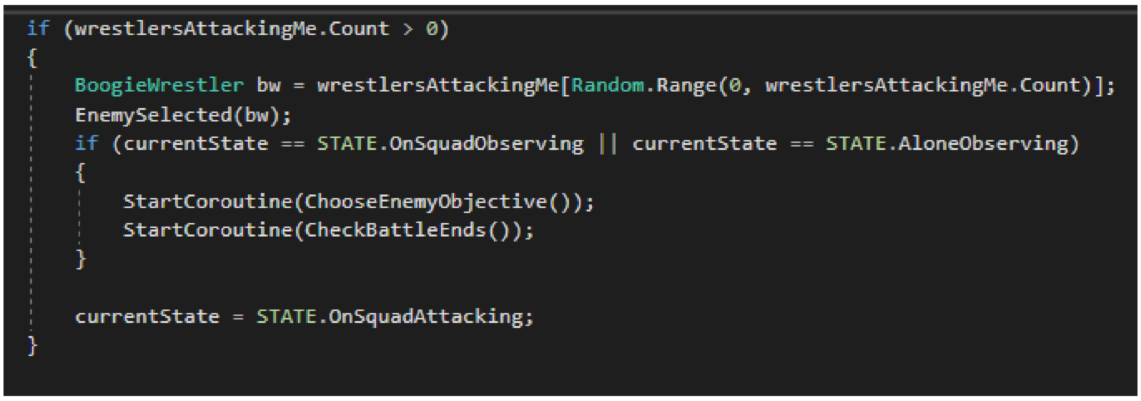
Figure 29: AttackEnemy()

Each wrestler has a *PlayerHealth.cs* component which has a health value. Each attack subtracts an amount to that health. When that health is 0 or under 0 the wrestler dies. If is the commander who has died, the whole squad is now without a commander, so they are not a squad anymore and they will change their behavior according to it. Besides, as soon as a battle between squads starts, a coroutine called *CheckBattleEnds()* is started. When the battle has ended, this coroutine will finish and will regroup those who remain alive of the same team.

3.4.3.3. Emergency behaviors in a battle

During a battle there are some things that can happen that vary the behavior of those involved in it. These changes are summarized in changing the current target selected. We are going to see in which situations this will occur. I consider this is an important part so I will show some code captures of these situations.

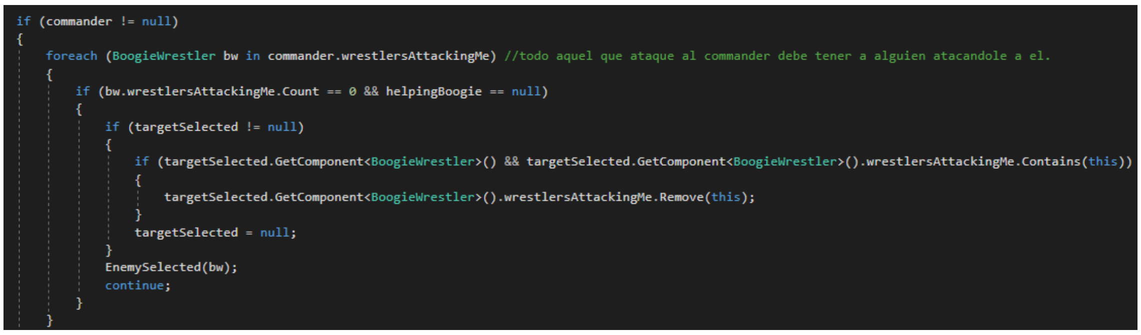
- A wrestler from team A has not any target selected and anyone from team B attacks him from behind. Automatically wrestler A will set wrestler B as the selected target. (Figure 30)



```
if (wrestlersAttackingMe.Count > 0)
{
    BoogieWrestler bw = wrestlersAttackingMe[Random.Range(0, wrestlersAttackingMe.Count)];
    EnemySelected(bw);
    if (currentState == STATE.OnSquadObserving || currentState == STATE.AloneObserving)
    {
        StartCoroutine(ChooseEnemyObjective());
        StartCoroutine(CheckBattleEnds());
    }

    currentState = STATE.OnSquadAttacking;
}
```

Figure 30: Emergency behavior 1

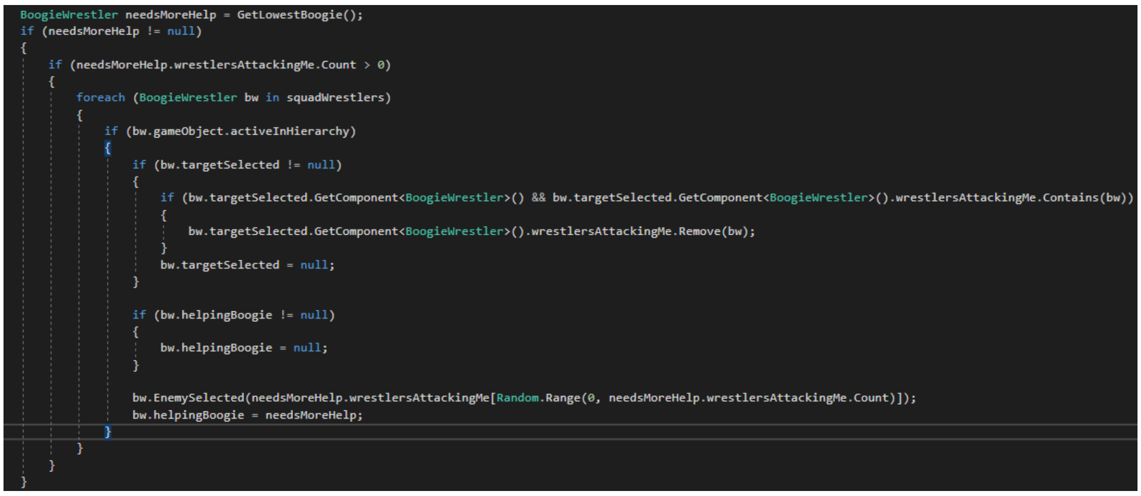
- If the commander from team A is being attacked by one or more wrestlers from team B, all the wrestlers of B attacking the commander of A must have any wrestler of A attacking him/her. This is a way of ensure that if anyone is attacking the commander, the commander will receive help immediately. (Figure 31)



```
if (commander != null)
{
    foreach (BoogieWrestler bw in commander.wrestlersAttackingMe) //todo aquel que ataque al commander debe tener a alguien atacandole a el.
    {
        if (bw.wrestlersAttackingMe.Count == 0 && helpingBoogie == null)
        {
            if (targetSelected != null)
            {
                if (targetSelected.GetComponent<BoogieWrestler>() && targetSelected.GetComponent<BoogieWrestler>().wrestlersAttackingMe.Contains(this))
                {
                    targetSelected.GetComponent<BoogieWrestler>().wrestlersAttackingMe.Remove(this);
                }
                targetSelected = null;
            }
            EnemySelected(bw);
            continue;
        }
    }
```

Figure 31: Emergency behavior 2

- If a wrestler from team A has less health points than a certain minimum, all the wrestlers from A will go protecting him attacking all those who attack the weak wrestler. (Figure 32)



```
BoogieWrestler needsMoreHelp = GetLowestBoogie();
if (needsMoreHelp != null)
{
    if (needsMoreHelp.wrestlersAttackingMe.Count > 0)
    {
        foreach (BoogieWrestler bw in squadWrestlers)
        {
            if (bw.gameObject.activeInHierarchy)
            {
                if (bw.targetSelected != null)
                {
                    if (bw.targetSelected.GetComponent<BoogieWrestler>() && bw.targetSelected.GetComponent<BoogieWrestler>().wrestlersAttackingMe.Contains(bw))
                    {
                        bw.targetSelected.GetComponent<BoogieWrestler>().wrestlersAttackingMe.Remove(bw);
                    }
                    bw.targetSelected = null;
                }

                if (bw.helpingBoogie != null)
                {
                    bw.helpingBoogie = null;
                }

                bw.EnemySelected(needsMoreHelp.wrestlersAttackingMe[Random.Range(0, needsMoreHelp.wrestlersAttackingMe.Count)]);
                bw.helpingBoogie = needsMoreHelp;
            }
        }
    }
}
```

Figure 32: Emergency behavior 3

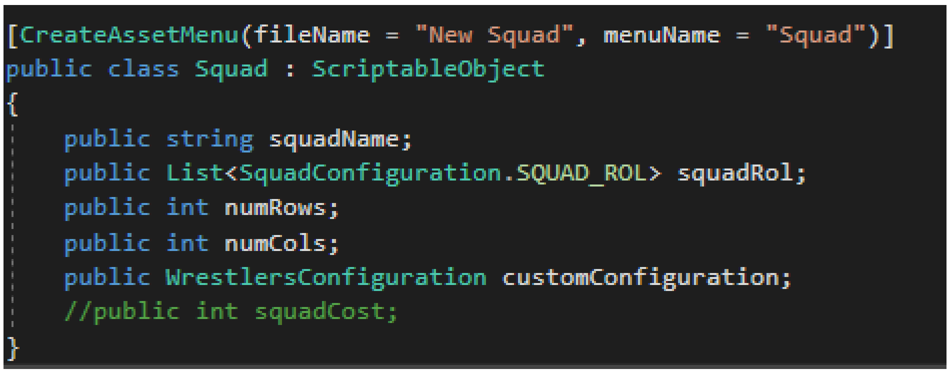
So, these are basically the situations that allow the emergency behaviors in the squad battles. Keep in mind that when a wrestler changes his objective, the wrestler who was attacking him will keep following trying to attack him and so over and over again, so all of the wrestlers involved in the battle will experiment a change.

3.4.4. In-game squad creator

One of the objectives of the project was to interact with the squads and for doing that some testing was needed. Each time I needed to test the interaction between different squads I had to create each of them, save them and load them every time in the game, this is, exiting the game and entering again every time. This was really inefficient.

Then, I had an idea: what if I created the squads while I was in play mode and at that moment, I saved them as Scriptable Objects and so I could load them without having to leave play mode? Besides, it should be really powerful for a player or for a designer. That was a really good idea.

First time I did, I had was to think some information about how to save and load data from Scriptable Objects. A Scriptable Object is an instance of a class, so I had to create a class which would contain the whole content that a squad would need. It is shown in Figure 33.



```
[CreateAssetMenu(fileName = "New Squad", menuName = "Squad")]
public class Squad : ScriptableObject
{
    public string squadName;
    public List<SquadConfiguration.SQUAD_ROL> squadRol;
    public int numRows;
    public int numCols;
    public WrestlersConfiguration customConfiguration;
    //public int squadCost;
}
```

Figure 33: Squad.cs

The *[CreateAssetMenu(filename = "New Squad", menuName = "Squad")]* (Unity Technologies, 2019). instruction is used to enable the creation of Squads from the editor of Unity. Looks like in the following Figure 34.

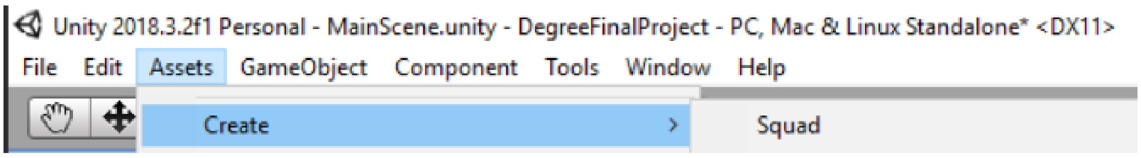

Unity 2018.3.2f1 Personal - MainScene.unity - DegreeFinalProject - PC, Mac & Linux Standalone* <DX11>
File
Edit
Assets
GameObject
Component
Tools
Window
Help
Create
Squad

Figure 34: Unity editor Create/Squad

Once this class was created, it was the moment to create a panel in which was possible to create the squad selecting a role for each position.

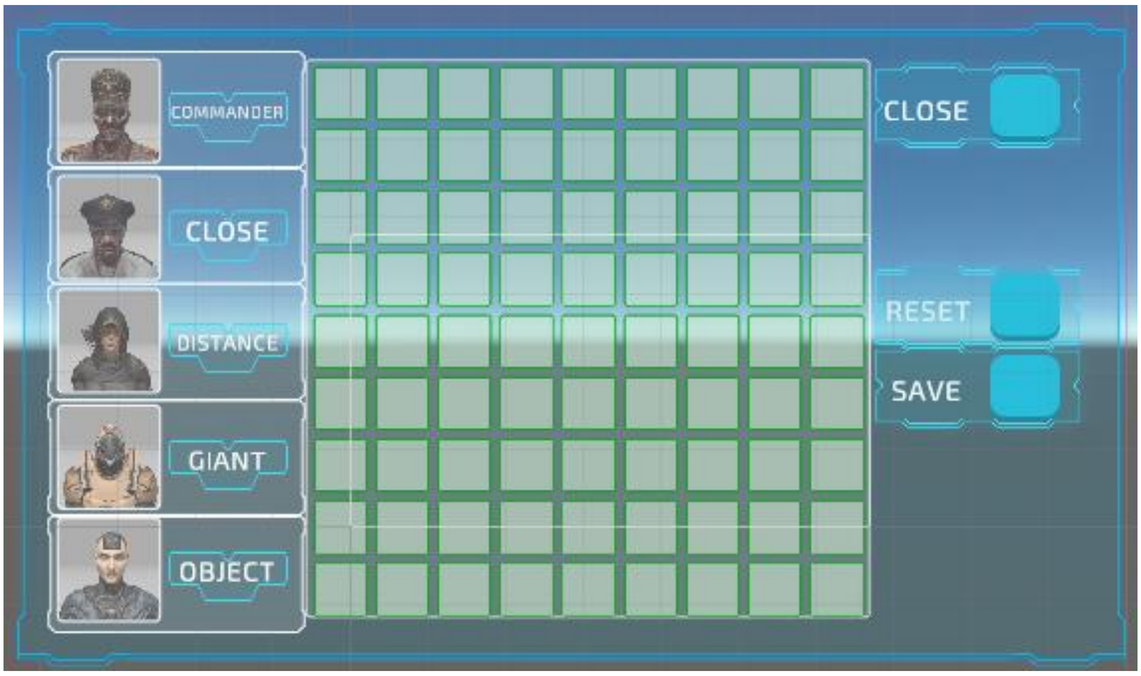


Figure 35: Squad in-game creator panel

As we can see in Figure 35, the operation was very simple. First click on the desired position and then select the desired role. When the squad was done, the button Save is pressed and it will execute the code shown in Figure 94.

What this method does is to create a list of roles. Remember that a role is just an identifier that indicates if it is a Commander, a Close, a Distance or a Giant. So, this list will be formed of all the selections of the player in the creator panel. Besides, it stores the number of rows and the number of columns of the squad.

Apart from executing this method, when the Save button was pressed a popup is shown requiring a name for the Squad. Internally it checks that name to be unique. Besides, this popup allows us to select a configuration for this Squad. This will be explained in the next point. Once the name is selected it is time to store these just-created Squad in the Unity Assets. Unity Assets is the place where all is stored. This is, it was the moment to store the Squad we have just finished creating as a configuration file. For doing this it was necessary to create an instance (Figure 45) of the Squad class (seen in Figure 33), assign in it the list of roles we just created in the previous step and more information as the number of rows, number of columns and the name of the squad. Finally, it was the moment to save it into Assets using *AssetDatabase.CreateAsset(asset, path)* and *AssetDatabase.SaveAssets()* (Unity Technologies, 2019). It can be seen in Figure 96.

Now it should be explained how to load the information stored in this Scriptable Objects in order to spawn a squad. But for doing this, it is needed to understand how the popups system works. Therefore, the Loading from assets will be explained in section 4.3.

Here is shown a demo of the full process:

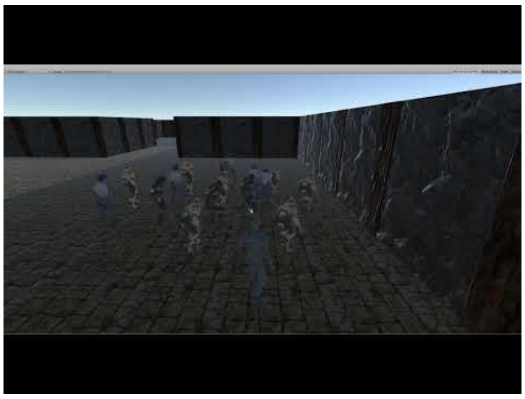


Figure 36: Squad in-game creator demo

[Video 5]

3.5. Emergency parameters

The emergency behaviors are based on a set of parameters. If one of these parameters vary, the emergent complex behavior obtained will change completely. So, at this point it is going to be shown how these parameters are organized into configuration files. Some of this configuration files captures will be referenced to Annex 8.

A configuration file is a file which contains all the parameters with certain value. This file is stored as a Scriptable Object and it is added to a group so the group will act basing their actions in these parameters and the behavior emerged will be close related to the configuration file. Remember that a Scriptable Object is an instance of a class. Remember that a configuration is just a set of parameters which variating them the emergent behavior of the multi-agent system will vary. So, there will not be the same set of parameters for the wrestlers and for the collectors, for example. So, every type of entity needs a different configuration, this is a different class. All of these classes are using the *[CreateAssetMenu(filename = "New … Configuration", menuName = "… Configuration")].* This is shown in Figure 37.

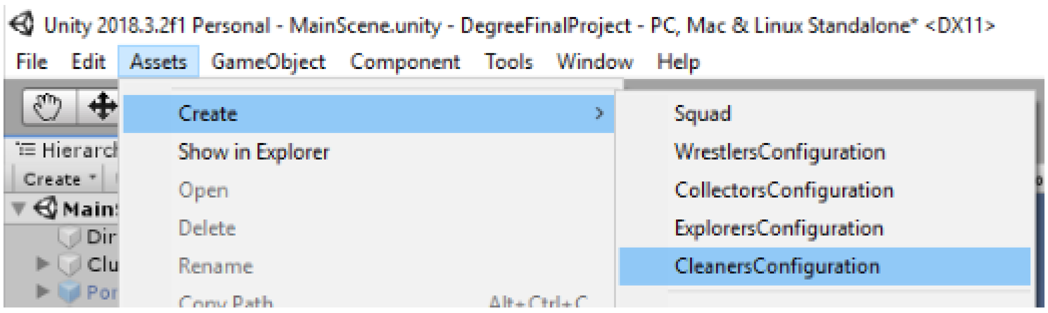


Figure 37: All the configurations

This point is going to be divided into the configuration parameters of the cleaners, the explorers, the collectors and the squad wrestlers. There will be code captures of the configuration files of each of them (this is, the Scriptable Objects). It is important to note that these following configurations are what I have considered as default, but these parameters can be varied and create infinite configurations.

3.5.1. Cleaners emergency parameters

Some of these parameters are common to the cleaners, the collectors and the explorers so these are going to be explained now and it will serve for the explorers and the collectors too.

- Min Speed: minimum speed of the agent.
- Max Speed: maximum speed of the agent.
- Probability Variate Speed: a value between 0 and 1. If it is 0 it will never vary the speed. If it is 1 it will vary the speed as soon as.
- Time Try Variate Speed: the time to try to variate the speed of the agent.
- Max Time To Find Objective: as soon as they are spawned, a coroutine is started. This coroutine checks the maximum time to come back to the player if they do not have an objective assigned.
- Time To Check If Work Finished: as soon as they have an objective assigned a coroutine is started. This coroutine checks if the work is done, this is: the obstacle is completed. So, this value is the time this check will be performed.

These are the common, now the proper parameters are going to be explained.

- Time To Charge Again: as soon as they place a debris, they are not able to charge again during this seconds.
- Time To Uncharge Again: as soon as they pick a debris, they are not able to uncharge again during this seconds.

3.5.2. Collectors emergency parameters

- Time To Collect: the time they are collecting in front of the rock.
- Time To Collect Again: as soon as they deposit elixir they are not able to collect again during this time.
- Time To Follow Marker Again: as soon as they deposit or collect elixir they are not able to follow markers (elixir paths) again during this time.
- Min Time Change Direction: minimum time to change direction.
- Max Time Change Direction: maximum time to change direction.

- Probability Change Direction: a value between 0 and 1 that handles the probability of changing the direction. If it is 0, they will never change the direction. If it is 1, they will change it always every X seconds.
- Time To Deposit: the time they are depositing the elixir.
- Time To Release Marker: they release a marker while they are carrying elixir. These markers will form the paths. They release one marker this time.

There is a coroutine started as soon as they are created that changes every X seconds the direction of the collectors. This is done because if they are following a path and it disappear, they will remain in that position without an objective so this will avoid it. This "X" is a random value between the minimum and maximum.

3.5.3. Explorers emergency parameters

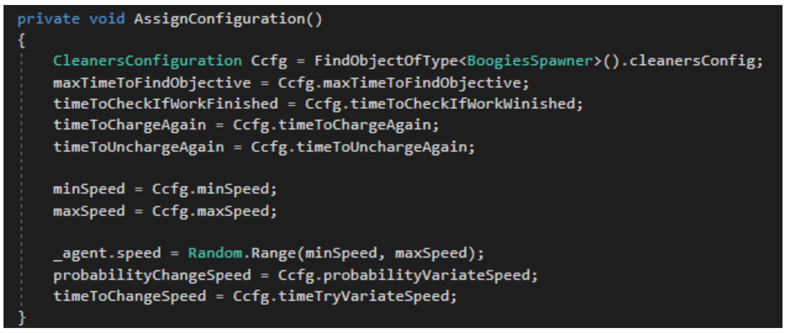
- Time To Carry Again: as soon as they deposit a clue, they are not able to carry another clue during this time.
- Probability Follow Clue: this is a value between 0 and 1. If it is 0, they will never follow the clues, so the clues will be useless. If it is 1, they will follow the clues always.

3.5.4. Squad emergency parameters

- Health Settings: initial health of each wrestler.
- Attack speed Settings: the wrestlers are able to attack every this time.
- Vision distance Settings: a value that indicates how far are the wrestlers able to see.
- Field of view Settings: a value that indicates how is the angle of vision (Field of View) of each wrestler.
- Attack range Settings: the minimum distance for the Distance Wrestler at which he can attack.
- Offset between them Settings: the distances between the wrestlers inside a Squad. This distance can variate between a maximum and a minimum every X time.
- Changing objective Settings: times and probabilities of changing the current wrestler target.
- Speed Settings: minimum and maximum values and probabilities of the speed of each kind of wrestler.

- Probability preferences Settings: probability of follow the preferences of each wrestler.
- Preferences Settings: list of the attack preferences of each wrestler.
- In Battle Settings: these parameters set the minimum percent of health points of each wrestler in order to be covered (remember the Emergency Behavior 3 of the Squad battles). Besides, there are some that indicate every few seconds they are able to cover.
- Callbacks when die Settings: a callback is a call to a function. So, this call will be specified in there and as soon as they die, they will call that function.

To conclude this point, it should be known that all the entities that use this configuration files have a function called *AssignConfiguration*() that gets all the values from the configuration file and assign the real components values with those (Figure 38).



```
private void AssignConfiguration()
{
    CleanersConfiguration Ccfg = FindObjectOfType<BoogiesSpawner>().cleanersConfig;
    maxTimeToFindObjective = Ccfg.maxTimeToFindObjective;
    timeToCheckIfWorkFinished = Ccfg.timeToCheckIfWorkWinished;
    timeToChargeAgain = Ccfg.timeToChargeAgain;
    timeToUnchargeAgain = Ccfg.timeToUnchargeAgain;

    minSpeed = Ccfg.minSpeed;
    maxSpeed = Ccfg.maxSpeed;

    _agent.speed = Random.Range(minSpeed, maxSpeed);
    probabilityChangeSpeed = Ccfg.probabilityVariateSpeed;
    timeToChangeSpeed = Ccfg.timeTryVariateSpeed;
}
```

Figure 38: AssignConfiguration()

4. Problems and results

4.1. Problems

During the development process there were some errors that had me stuck trying to solve them. In this point I am going to list them and explain how I solve them.

- Squad intern matrix was not exactly a problem, it was rather a constant headache. When I started to develop the Squad actions, some problems appeared. In particular, in those actions that required a change of position inside the matrix. This was really hard because it was needed to store the initial position, that is: the initial indexes inside the matrix, etc. And when some wrestler position was changed some things are needed to be changed. For example, when I changed the position of a wrestler with another wrestler, the current indexes inside the matrix were changed, the list of roles position was changed too, but the hardest part was to store all these changes because when the action "Reset formation" was performed, everything reseted as it was before doing any change.
- This one is close related with the explained above. Comparing custom classes is not as simple as it seems. The *SquadConfiguration* class (Annex 2) is a class that contains all needed to the creation of the Squads. It contains the following custom classes:
    - <u>*Squad*</u>: similar to the class that the Scriptable Objects instantiate to store and load the configuration of a squad. It contains some information about the squad (list of "roles", number of columns, number of rows, the name of the squad, the Configuration file selected, etc.).
    - <u>*SquadSlot*</u>: a class that contains the information needed for every slot of the squad. This is, the information needed on each position inside the squad (role, position, wrestler that is inside, etc.).
    - <u>*Index*</u>: a class that is just formed with two numbers (one indicating the row and another one indicating the column).

  So, the problem came when I tried to compare or assign values to these elements of different squads. When you equal two custom classes they both point to the same place in memory so changing the value of one will change the value of the other. This also means that two classes are equal if they are pointing to the same point of memory. These caused problems when I was trying to compare two lists of roles because despite of, they were exactly the same, they were pointing to different points in memory so it returned that they were different. And the opposite problem, when I equaled the indexes of one wrestler with the index of another wrestlers every time these indexes changed for one, they also changed for the other.

For solving the first problem I had to create a custom lists comparer (Figure 104) that internally compare element by element.

For solving the second problem, every time I wanted to change the Indexes of a wrestler with the values of the indexes of other wrestlers, I created them as new. So, each time that is created a class, it is created in another place of the memory so they will no longer be equal.

As it can be seen in Figure 105, first the values of the Index are stored and once it is done, the new Index can be created using the constructor. If we directly do: *leaderIndex = squadConfig.leaderPosition* we are saying that these two Index are pointing to the same position on memory so every time one of them changes, the other one will change too (Albahari, s.f.).

- Problems on the collectors when they were ordered to come back to the player: this problem happened when the collectors were carrying elixir and were ordered to return to the player. The problem here was that they subtracted the elixir from the stone but they had not placed it in the collecting machine because while they were looking for the collecting machine, the player ordered them to come back. So, this elixir disappeared forever and the Collecting Obstacle system would not work. The solution was to return that elixir to the stone source as soon as the action "Back to player" was executed.
- Problems with coroutines of wrestlers: I experimented a big problem when I was trying to face two squads. As soon as they started to fight everything was good, but few moments later all the squad members of one of the squads died inmediately. After some research of how this could be happening, I realized that they had active the same coroutine several times. It is important to remember that as soon as a wrestler was selected as a target, two coroutines were initialized. *GoToEnemy() and AttackEnemy().* These both coroutines were active since that moment to the end of the battle so every time they changed the objective and selected a new target these both coroutines were enabled again. The solution was to maintain these coroutines active while there was an enemy selected. In the moment the target died or it was changed the objective these coroutines stopped and started again as soon as a new target was selected.
- This is probably the biggest problem I experimented in the game. The 9 of April I decided to import a 3D pack which contained some models and animations. Specifically, those I was going to use for the enemy wrestlers. This pack was really big and included more than 20 models and I only needed 4. What I did was to delete them from Unity, nonetheless I had already committed it so when I tried to push it some errors appeared. These errors did not allow me to push during some time until I decided to repair it. The first option was to delete the repo and create another one but I would have lost all the commits already pushed, so what I do, after some research, cancel all the pendent commits and join all them in

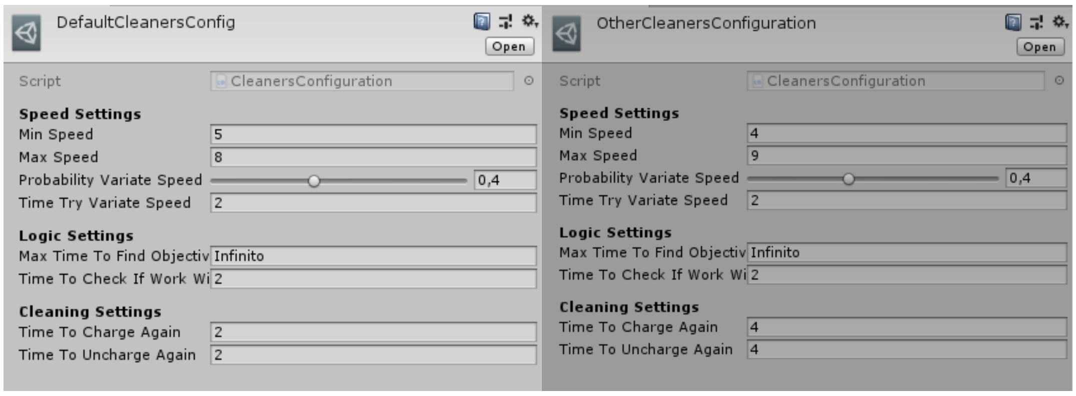
only one. After it, I tried to push it and the error disappeared. This commit contained the work of a month. The id of this commit is 8d272b5.

4.2. Results

Based on the objectives cited in point 1.5, I can ensure that all the main objectives have been completed. In this point and in the following (5.3) it is going to be shown how these objectives have been accomplished. The secondary objectives have been completed too and they will be explained in the Annexes.

In this point it is showed different complex behaviors that emerge from the actions of the entities and how they change depending on the parameters configuration of each of them. It is going to be also shown an example of the collectors, an example of the cleaners, an example of the explorers and an example of a wrestler's squad battle. Besides, this will serve to see demos of the explained in the section 3.

In the first place, we are going to talk about how the Cleaners will change depending on the emergency parameter they have. In the case of the Cleaners we are going to change the minimum time until they are able to charge again and the minimum time until they are able to deposit again. As we can see in Figure 39 these two parameters are the only changed and the demo (Figure 40) shows that the resultant complex behavior changes completely.


DefaultCleanersConfig — Open

Script	CleanersConfiguration
Speed Settings	
Min Speed	5
Max Speed	8
Probability Variate Speed	0,4
Time Try Variate Speed	2
Logic Settings	
Max Time To Find Objectiv	Infinito
Time To Check If Work Wi	2
Cleaning Settings	
Time To Charge Again	2
Time To Uncharge Again	2

OtherCleanersConfiguration — Open

Script	CleanersConfiguration
Speed Settings	
Min Speed	4
Max Speed	9
Probability Variate Speed	0,4
Time Try Variate Speed	2
Logic Settings	
Max Time To Find Objectiv	Infinito
Time To Check If Work Wi	2
Cleaning Settings	
Time To Charge Again	4
Time To Uncharge Again	4

Figure 39: Cleaners parameters

As it can be seen, the variation is from 2 seconds and 2 seconds to 4 seconds and 4 seconds, respectively. The first thing we can imagine is that they will take more time in order to finish the job because they will be more time wandering around the objective without being able to deposit or charge the debris. Nonetheless, the results are completely different. They were amazing to me too. So, as we know and we have been able to observe: the emergent behaviors can be unexpected. This is because every change made as small as it is triggers changes in all actions making it very difficult to predict the final result.

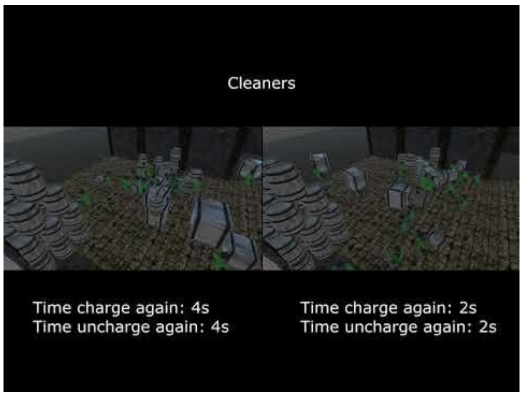
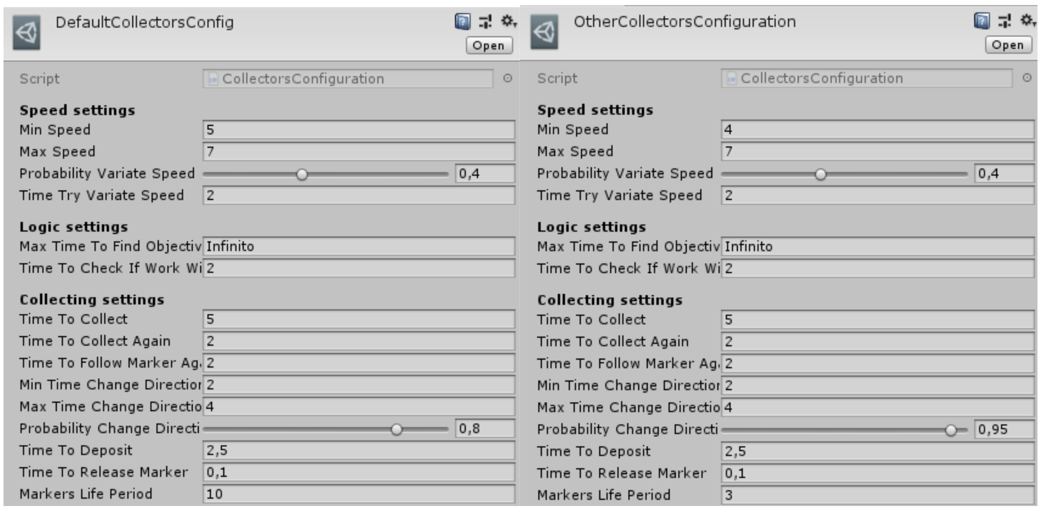


Figure 40: Cleaners variation demo

[Video 6]

In the second place, we can see how the Collectors will change depending on their emergency behaviors. In this case, it has been decided to change the path markers life period. As we can see in Figure 41 it is just the parameter called *MarkersLifePeriod.*



	DefaultCollectorsConfig	OtherCollectorsConfiguration
Script	CollectorsConfiguration	CollectorsConfiguration
Speed settings		
Min Speed	5	4
Max Speed	7	7
Probability Variate Speed	0,4	0,4
Time Try Variate Speed	2	2
Logic settings		
Max Time To Find Objectiv	Infinito	Infinito
Time To Check If Work Wi	2	2
Collecting settings		
Time To Collect	5	5
Time To Collect Again	2	2
Time To Follow Marker Ag	2	2
Min Time Change Directior	2	2
Max Time Change Directio	4	4
Probability Change Directi	0,8	0,95
Time To Deposit	2,5	2,5
Time To Release Marker	0,1	0,1
Markers Life Period	10	3

Figure 41: Collectors parameters

As we are about to see in the demo of Figure 42, due to the markers have a life period as short as 3s, the paths become not as useful as when they have 10s of life period. This affects the collectors in that they are no longer able to follow a path, so finding a stone or collecting stone is a matter of randomness. Therefore, in this case, they will take more time to complete the job.

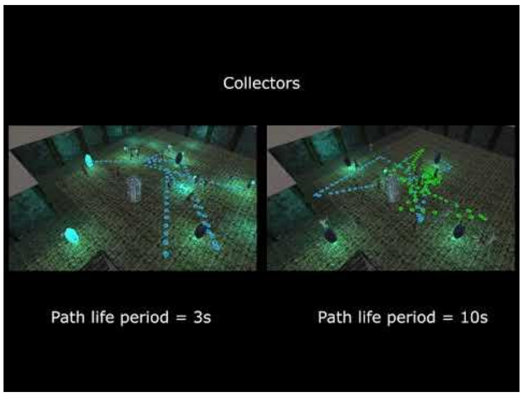
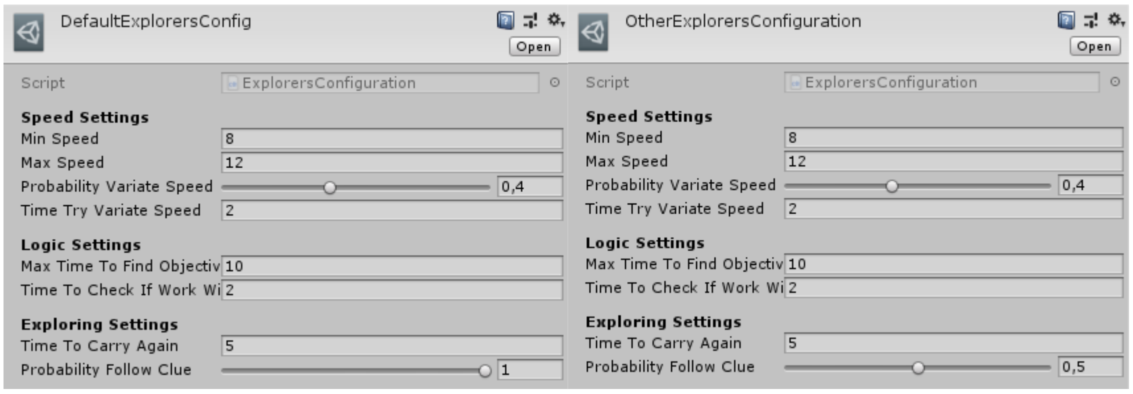


Figure 42: Collectors variation demo

[Video 7]

In the third place, we can see how the explorers change their behavior and the time to complete the multipath depending on the parameters. In this case it has been decided to vary the parameter *ProbabilityFollowClue*. That is, if there is a clue placed that indicates which path to take in each corridor, they have a probability of following it or select randomly a path. If this probability is 0, the clues are completely useless so it has been decided to set that probability to 50%. In Figure 43 we can see this variation.



	DefaultExplorersConfig	OtherExplorersConfiguration
Script	ExplorersConfiguration	ExplorersConfiguration
Speed Settings		
Min Speed	8	8
Max Speed	12	12
Probability Variate Speed	0,4	0,4
Time Try Variate Speed	2	2
Logic Settings		
Max Time To Find Objectiv	10	10
Time To Check If Work Wi	2	2
Exploring Settings		
Time To Carry Again	5	5
Probability Follow Clue	1	0,5

Figure 43: Explorers parameters

As it can be seen in the demo of Figure 44, in the case the probability is 50% they take a really long time in order to complete the level. This is because they select randomly the corridor at least the half of the times. So, as soon as this value is closer to 1 (100%), they will select the correct path in more occasions and it will be faster.

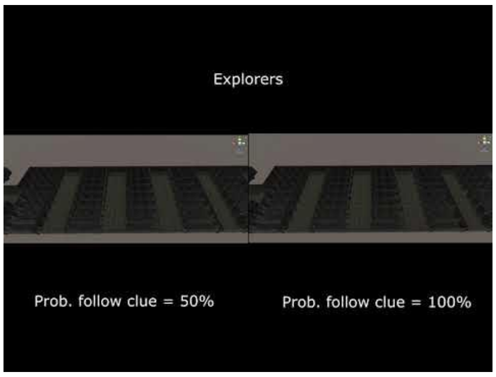


Figure 44: Explorers variation demo

[Video 8]

Finally, we are going to show how we can vary these parameters to the wrestlers in order to obtain different emergency complex behavior. These three examples shown before are quite simple compared with the number of parameters the squads have. The possibilities that offer the variation of all these parameters are infinite. We can create long battle squads, short battle squads, make them move fast, make them move slow, dynamic battles (continuously changing the objective) or more strategic battles (always attack to your first preference), etc. Besides, playing with the damage, range of attack, vision distance, etc. there can be created thousands of possible squads. This allows a lot of combinations and it is easy to a designer manipulate these parameters and see how the resulting battle changes in order to get the expected result.

It is important to remember that the in-game squad creator allows to select the configuration. What it does is to access the folder where all the Wrestlers Configurations are stored and show them. The user just has to select the one he wants.

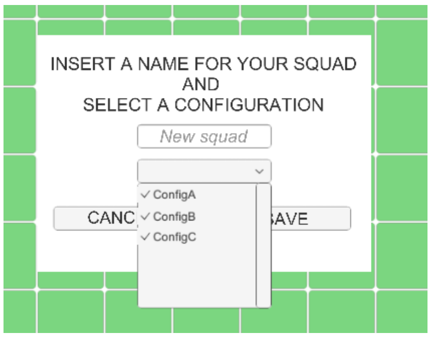


Figure 45: Select configuration

In order to simplify this, we are just going to vary the attack preferences of each wrestler and the probabilities of these. In Figure 107 and 108 we can see two different configuration files of the Enemy Squad. The Allie Squad has the same configuration in both of the battles (Figure 106). The only thing we are going to vary is the Enemy Squad parameters. Due to these images need to be big in order to allow the correct reading of them, these are in Annex 8.

As we can see in the demo of Figure 46, depending on the strategy attack adopted by the wrestlers, the result of the battle will be one or another. In the first case, when the allies win the enemy squad is attacking to giants and close as the first option. Besides, the Distance is in all the wrestlers the third option so they are allowing the Distances to be attacking from far with nobody attacking them. Furthermore, the probabilities are varying so probably, they are not attacking their first option in many occasions.

In the second case, when the enemies win, they change Distance from third place to second place in all the wrestlers. Besides, Giant become the first option for everyone because the Giants are wrestlers who have great power attack and high health. So, it is interesting to kill them as soon as possible. Furthermore, the probabilities are always 100%. This means they will always attack their first option.

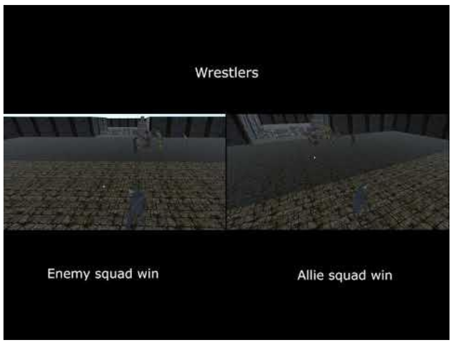


Figure 46: Wrestlers variation demo

[Video 9]

As it has been seen, varying a bit the parameters of the configuration files of each entity, the result changes. Here we have just seen four examples but the possibilities are unimaginable. Besides, this allows designers to create behaviors to their likely just varying some parameters.

4.3. Hours spent

In this subsection we show the number of hours spent in each task based on the 1.7. point schedule. Furthermore, these times are going to be compared with the expected.

Task	Hours expected	Hours spent
Read and learn theory of emergency behaviors	10 – 15	11
Entities behaviors design	10-15	18
Entities behaviors coding	50 – 60	65
Squad system develop	100 – 110	110

Organization of the parameters in configuration files	40 – 50	35
Project report and other documents	40 - 50	45
Video game demo	60 – 70	60
Total	310 - 370	344

As it can be seen, the expected hours are not fare from the hours used so it can be considered that the planification and design of the project was carried out fairly well.

4.4. Project access and download

In order to access to the project repository, it is necessary to follow this link. https://github.com/ricardosanz97/DegreeFinalProject

Here, in the following link there is the Trello of this project. In it, all the develop of the project can be followed: https://trello.com/b/DCYKIfDc/dfp

In the following link of Google Drive is located the build of the project in order to be tested: https://drive.google.com/drive/folders/18bQ1A-IJ9-87n2IJM-iftLtxjBBvdeCG?usp=sharing

In my YouTube channel there are some videos of the development of the project, including a video of the playable demo. Here is the link:

https://www.youtube.com/channel/UC75iDHTEptSxIGxSK0aJWJg?view_as=subscriber

5. Conclusions and future work

As soon as my DFP's supervisor talked to me about the emergency behaviors and multi-agent systems I started to get interested on it and I thought it would be really interesting to do this project around it. Besides, before talking with my supervisor I was already interested in Artificial Intelligence so this project theme fitted perfectly on me.

Once I have finished it, I can affirm that the emergency behaviors applied to multi-agent systems possibilities are immense and the workload that is saved using them correctly is enormous. I am pretty sure that the AI guidelines in the video games that are going to come will be based in the emergency behaviors. The main advantage of them is that they do not need a controller that manages everything. They act by emergency, this is, they act according to the interactions with other entities, with the player or with the scenario.

Besides, the tools created in this project can be used in other project and if more behaviors are needed the base system is already created so creating news would be really simple. The squad system allows multiple implementations in a video game. They can be used as an army, as environment elements, as the main elements of the game, etc. The squad creator can be used in MMO games or in simulations of battles in order to see how one formation of the squad would be better than another, etc.

Talking about the future work, there are some things that I will finish that have not been developed during the project because of the time limitation. Besides, more emergency behaviors will be included. My main purpose for the future of this project is to adapt it as a tool that can be exported to other projects and can be easily manipulated by someone. Besides, due to the quality of code that allows scalability, it is easily scalable and using the already created base system, creating and expanding the project with new behaviors, new actions, etc. would be really easy.

I have enjoyed developing this project and I have learnt a lot about coding and all the advantages of using this. Besides, as it is related with the Artificial Intelligence, I think I have learnt a lot doing this project. Furthermore, I feel proud of having worked on something innovative as it is the application of emergency behaviors on multi-agent systems in video games as a first-class element.

6. Bibliography

3D, J. (n.d.). *Medieval barrels and boxes*. Retrieved April 20, 2019, from Unity Asset Store: https://assetstore.unity.com/packages/3d/props/exterior/medieval-barrels-and-boxes-137474

3DHaupt. (n.d.). *Sci-Fi Emergency Backup Generator*. Retrieved May 18, 2019, from sharecg: https://www.sharecg.com/v/92921/browse/5/3D-Model/Sci-Fi-Emergency-Backup-Generator

Adobe. (2019). *Adobe Photoshop Free Trial Download*. Retrieved April 29, 2019, from Adobe Photoshop: https://www.adobe.com/es/products/photoshop/free-trial-download.html#

Adobe. (2019). *Mixamo*. Retrieved April 07, 2019, from Mixamo: https://www.mixamo.com/#/?page=1&type=Character

Albahari, J. (n.d.). *C# Concepts: Value vs Reference Types*. Retrieved May 11, 2019, from Albahari: http://www.albahari.com/valuevsreftypes.aspx

Atlassian. (2019). *Trello*. Retrieved May 17, 2019, from Trello: https://trello.com/

Audacity. (2019). *Audacity Download*. Retrieved April 22, 2019, from Audacity: https://www.audacityteam.org/download/

Biomolecular Blog. (n.d.). *Biología Molecular*. Retrieved May 3, 2019, from Biomolecular Blog: https://biomolecularblog.wordpress.com/2018/03/31/eusocialidad-que-es/

Fungus Games. (n.d.). *Fungus*. Retrieved May 10, 2019, from Unity Asset Store: https://assetstore.unity.com/packages/templates/systems/fungus-34184

GitHub. (2019). *GitHub*. Retrieved May 07, 2019, from GitHub: https://github.com/

Johansson, D. (2012, July 1). Complex Systems in Video Games - a literature survey.

Lague, S. (2015, December 26). *Field of view visualisation (E01).* Retrieved May 15, 2019, from YouTube: https://www.youtube.com/watch?v=rQG9aUWarwE

MAGIX Software GmbH. (2019). *Vegas Dowload*. Retrieved April 20, 2019, from Vegas Creative Software: https://www.vegascreativesoftware.com/index.php?id=351&L=52&AffiliateID=178&phash=LjmUipAVFVxrmmmH&ef_id=Cj0KCQjw2v7mBRC1ARIsAAiw348hJvdRWMEKtRg2gv1FfIQB9WjGIx9goQdOtI1nJeIMA2HPWAauaxsaAv8OEALw_wcB:G:s&gclid=Cj0KCQjw2v7mBRC1ARIsAAiw348hJvdRWMEKtRg2gv1FfIQ

Mathivet, V. (2018). Sistemas multi-agentes. In V. Mathivet, *Inteligencia Artificial para desarrolladores - Conceptos e implementación en C#* (pp. 359-422).

medwuf. (n.d.). *Teleportation - Portal*. Retrieved May 03, 2019, from CGTrader: https://www.cgtrader.com/free-3d-models/exterior/sci-fi/teleportation-portal

Mestras, J. P. (n.d.). Metodologías de desarrollo de Sistemas Multi-Agente. Madrid, Madrid, Spain.

Microsoft. (2019). *Download Visual Studio*. Retrieved May 05, 2019, from Visual Studio: https://visualstudio.microsoft.com/es/downloads/

Protofactor. (n.d.). *Heroic Fantasy Creatures Full pack Volume 1*. Retrieved April 01, 2019, from Unity Asset Store: https://assetstore.unity.com/packages/3d/characters/creatures/heroic-fantasy-creatures-full-pack-volume-1-5730

saladin0511. (n.d.). *Old Gate model 3d*. Retrieved May 10, 2019, from Free3d: https://free3d.com/3d-model/old-gate-36315.html

Shiffman, D. (n.d.). *Cellular Automata*. Retrieved May 10, 2019, from The Nature of Code.

TL, & DR. (2018). *What does NavMesh.AllAreas specify in Unity?* Retrieved April 03, 2019, from GameDev: https://gamedev.stackexchange.com/questions/147915/what-does-navmesh-allareas-specify-in-unity

TutorialsPoint. (n.d.). *Design Patter - Singleton Pattern*. Retrieved May 1, 2019, from Tutorialspoint: https://www.tutorialspoint.com/design_pattern/singleton_pattern.htm

TutorialsPoint. (n.d.). *Design Patterns - Observer Pattern*. Retrieved February 20, 2019, from Tutorialspoint: https://www.tutorialspoint.com/design_pattern/observer_pattern.htm

Unity Technologies. (2019). *AssetDatabase.CreateAsset*. Retrieved April 19, 2019, from Unity 3D Documentation: https://docs.unity3d.com/ScriptReference/AssetDatabase.CreateAsset.html

Unity Technologies. (2019). *AssetDatabase.LoadAssetAtPath*. Retrieved April 10, 2019, from Unity 3D Documentation: https://docs.unity3d.com/ScriptReference/AssetDatabase.LoadAssetAtPath.html

Unity Technologies. (2019). *Button.onClick*. Retrieved May 10, 2019, from Unity 3D Documentation: https://docs.unity3d.com/ScriptReference/UI.Button-onClick.html

Unity Technologies. (2019). *CreateAssetMenuAttribute*. Retrieved May 2, 2019, from Unity 3D Documentation: https://docs.unity3d.com/ScriptReference/CreateAssetMenuAttribute.html

Unity Technologies. (2019). *NavMeshPath*. Retrieved May 05, 2019, from Unity 3D Documentation: https://docs.unity3d.com/ScriptReference/AI.NavMeshPath.html

Unity Technologies. (2019). *Unity Download Archive*. Retrieved May 17, 2019, from Unity 3D: https://unity3d.com/es/get-unity/download/archive?_ga=2.72937107.1687820219.1558112145-1103797267.1532608754

Wilensky, U. (2016). *Net Logo*. Retrieved May 18, 2019, from Northwestern: https://ccl.northwestern.edu/netlogo/

Zygomatic. (n.d.). *Diagram Editor*. Retrieved May 18, 2019, from Diagram Editor: https://www.diagrameditor.com/

Music used for the Menu: https://www.youtube.com/watch?v=5oF5lS0sajs

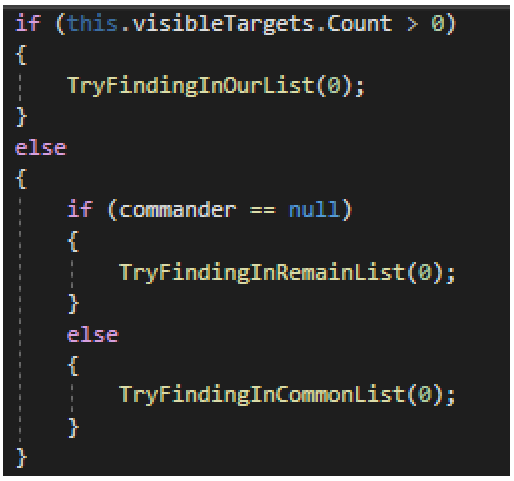
Music used for the Demo: https://www.youtube.com/watch?v=a0Av2XNPd_g

7. Annexes

7.1. Annex 1 (Selection Process)

Remember that to select the attacking target of the wrestlers they do a selection process. In this annex it is going to show this selection process. Keep in mind that each wrestler has a list of preferences to attack and a parameter which indicates the probability of follow these preferences. It this parameter is close to 0, the selection will be almost random and if it is close to 1 it will follow the preferences list.

Each wrestler has a list in which are stored all the targets that are visible to the wrestler. Besides, there is another common list for all the wrestlers of the squads. This is the visible targets list of the commander. So, each time a wrestler adds a target to its list, it also adds the target to the common commander visible targets list. So, in Figure 47 it can be shown how they perform the target selection searching in these three different lists.

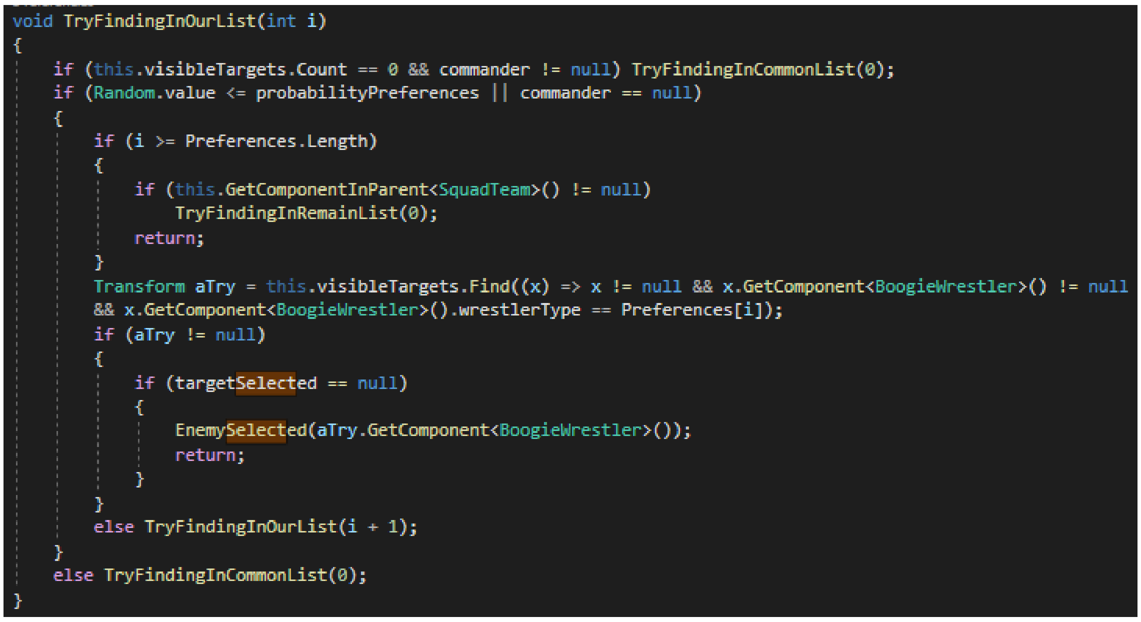


```
if (this.visibleTargets.Count > 0)
{
    TryFindingInOurList(0);
}
else
{
    if (commander == null)
    {
        TryFindingInRemainList(0);
    }
    else
    {
        TryFindingInCommonList(0);
    }
}
```

Figure 47: Selection process

Before explaining it, the remain list is a list contained in the parent of the Squad that contains all the enemies. This list is also accessed when the commander is dead. If the wrestler visible targets list has more than one element, they select find an enemy in this list. Otherwise, depending if the commander is alive *(!= null*) or not *(== null*) they will find on the common or in the remain list, respectively.

Once it is established where the wrestler is going to find a target, these methods use recursion depending on the probabilities and preferences. Depending on the probability of follow the preferences or not they will keep finding in this list or will jump to find on one of the other lists. In Figure 48 it is shown how the *TryFindingInOurList()* is implemented. The *TryFindingInRemainList()* and *TryFindingInCommonList()* work in the same way.



```
void TryFindingInOurList(int i)
{
    if (this.visibleTargets.Count == 0 && commander != null) TryFindingInCommonList(0);
    if (Random.value <= probabilityPreferences || commander == null)
    {
        if (i >= Preferences.Length)
        {
            if (this.GetComponentInParent<SquadTeam>() != null)
                TryFindingInRemainList(0);
            return;
        }
        Transform aTry = this.visibleTargets.Find((x) => x != null && x.GetComponent<BoogieWrestler>() != null
        && x.GetComponent<BoogieWrestler>().wrestlerType == Preferences[i]);
        if (aTry != null)
        {
            if (targetSelected == null)
            {
                EnemySelected(aTry.GetComponent<BoogieWrestler>());
                return;
            }
        }
        else TryFindingInOurList(i + 1);
    }
    else TryFindingInCommonList(0);
}
```

Figure 48: TryFindingInOurList()

As it can be seen it finds a target in the list which role is equal to the current preference. If it exists (*!=null*), that target is selected. If it does not exist, the method *TryFindingInOurList()* is called again, but now incrementing the index passed via parameter by 1.

7.2. Annex 2 (Squad Class)

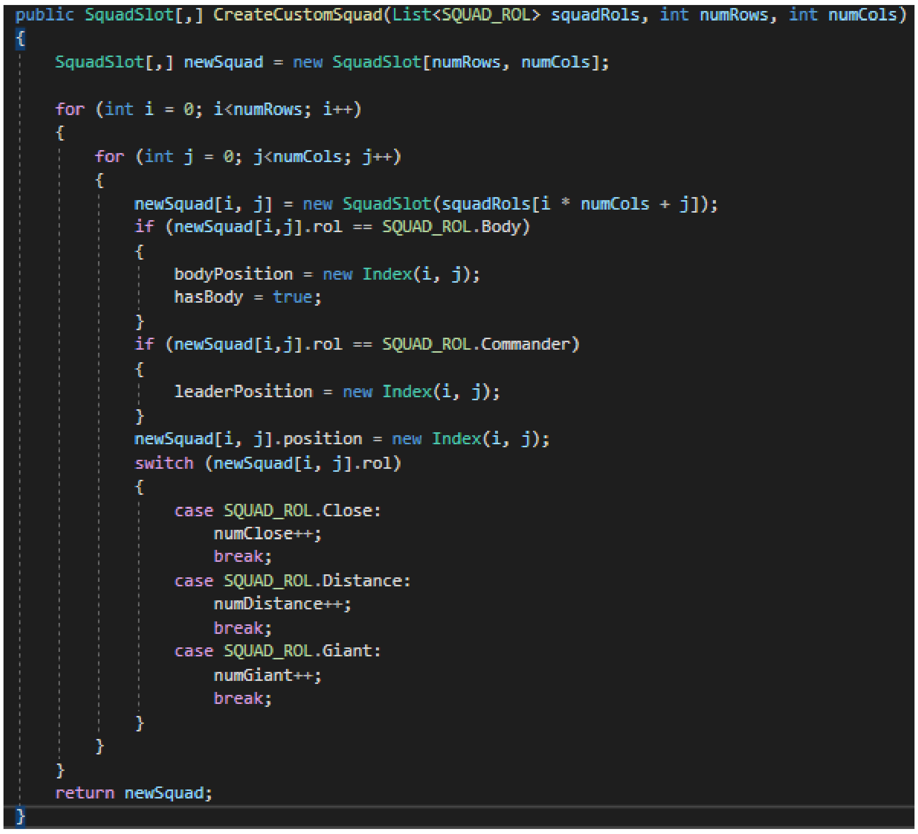
In this annex it is going to be shown how the *SquadConfiguration.cs* script is. This class contains all the information that allows creating, modifying and interacting with the squads. It is formed by the *SquadConfiguration* class, the Squad class, the *SquadSlot* class and the *Index* class.

The *SquadConfiguration* class contains information about the different roles and some methods to compare different squads, like the one shown in Figure 33.

The *Squad* class contains the information needed to define a squad. Apart from the information already seen like name, list of roles, indexes of body, indexes of commander, number of columns, etc. it has a constructor that returns a matrix of *SquadSlot.* It can be seen in Figure 49.

The *SquadSlot* class contains the information needed for each position inside a squad matrix because a squad is a list of *SquadSlot* objects. This contains the role, the indexes position and the wrestler that is occupying this slot.

For ending this annex, the Index class have already been explained and it just contains two int values. One for indicating the row and other for indicating the column.



```
public SquadSlot[,] CreateCustomSquad(List<SQUAD_ROL> squadRols, int numRows, int numCols)
{
    SquadSlot[,] newSquad = new SquadSlot[numRows, numCols];

    for (int i = 0; i<numRows; i++)
    {
        for (int j = 0; j<numCols; j++)
        {
            newSquad[i, j] = new SquadSlot(squadRols[i * numCols + j]);
            if (newSquad[i,j].rol == SQUAD_ROL.Body)
            {
                bodyPosition = new Index(i, j);
                hasBody = true;
            }
            if (newSquad[i,j].rol == SQUAD_ROL.Commander)
            {
                leaderPosition = new Index(i, j);
            }
            newSquad[i, j].position = new Index(i, j);
            switch (newSquad[i, j].rol)
            {
                case SQUAD_ROL.Close:
                    numClose++;
                    break;
                case SQUAD_ROL.Distance:
                    numDistance++;
                    break;
                case SQUAD_ROL.Giant:
                    numGiant++;
                    break;
            }
        }
    }
    return newSquad;
}
```

Figure 49: CreateCustomSquad(...)

7.3. Annex 3 (NavMesh Unity)

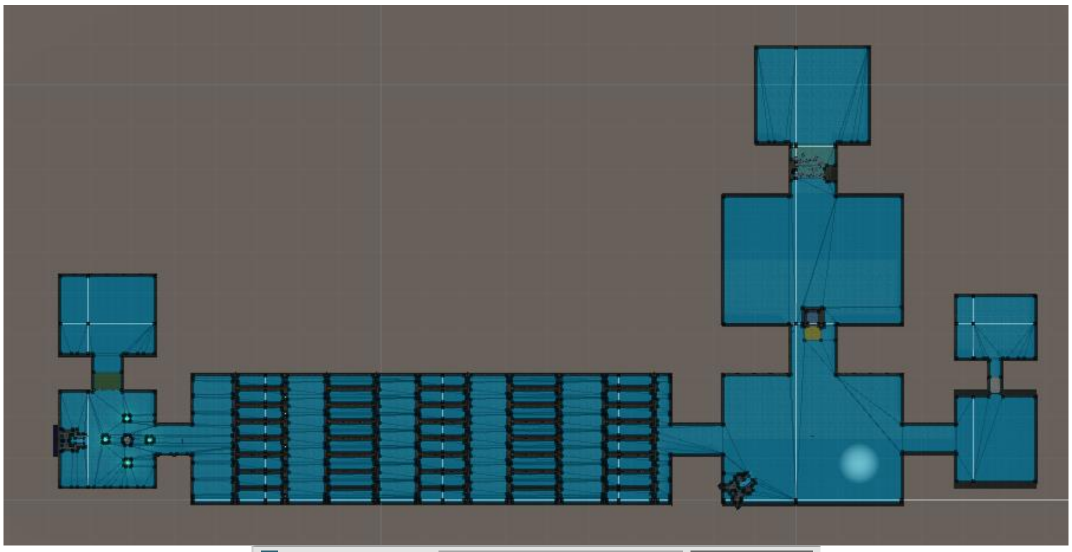
In this point it is not going to be explained how the AI NavMesh system works. Instead of it, I am going to show what I have learnt of it because this was one of the secondary objectives of the project.

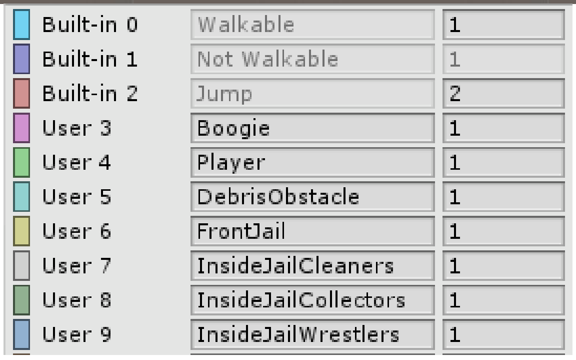
7.3.1. Area masks

Each environment object has a NavMesh area mask and an agent has a set of masks. The agent will be able to walk just in the areas that are inside that set. So, for example, if a cleaner has the masks: A, B and C. He will be able to walk on those objects with area A, B or C. This has been used for locking and unlocking areas. For example, in the demo, until the cleaner's door has not been opened, the cleaners are not able to exit the room in which they are and the player is not able to enter in that room, because:

- The area mask of the room where the collectors are is not in the set of masks of the player.
- The area mask outside the room where the collectors are is not in the set of masks of the cleaners.

As seen in Figure 50, in the environment of the demo there are some masks in different color.





🟦	Built-in 0	Walkable	1
🟪	Built-in 1	Not Walkable	1
🟥	Built-in 2	Jump	2
🟪	User 3	Boogie	1
🟩	User 4	Player	1
🟦	User 5	DebrisObstacle	1
🟨	User 6	FrontJail	1
⬜	User 7	InsideJailCleaners	1
🟩	User 8	InsideJailCollectors	1
🟦	User 9	InsideJailWrestlers	1

Figure 50: Navigation and masks

The set of masks for each agent can be modified at runtime. Nonetheless it is not trivial. For doing it using the attribute areaMask we have to add the following:

+= 1 << NavMesh.GetAreaFromName("name of the area mask to add").

This was for adding a new area mask. If we want to delete one of them:

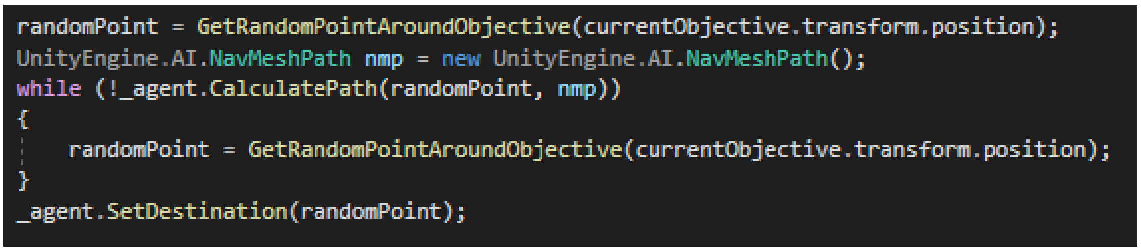
+= 0 << NavMesh.GetAreaFromName("name of the area mask to delete").

This information was extracted from an external source. (TL & DR, 2018)

7.3.2. Determine if a destination is valid

As it has been shown, all the entities on the project are agents. That is, they move using the *SetDestination(Vector3 where)* method from the *NavMeshAgent* component. It is important to determine if a destination is reachable or not by the agent before ordering him to go there.

The way for doing it is to use the he *NavMeshPath*. (Unity Technologies, 2019). This declare a path and using *CalculatePath* it returns if it is possible to create a path from the current position to the selected destination or not. It can be seen in Figure 51.



```
randomPoint = GetRandomPointAroundObjective(currentObjective.transform.position);
UnityEngine.AI.NavMeshPath nmp = new UnityEngine.AI.NavMeshPath();
while (!_agent.CalculatePath(randomPoint, nmp))
{
    randomPoint = GetRandomPointAroundObjective(currentObjective.transform.position);
}
_agent.SetDestination(randomPoint);
```

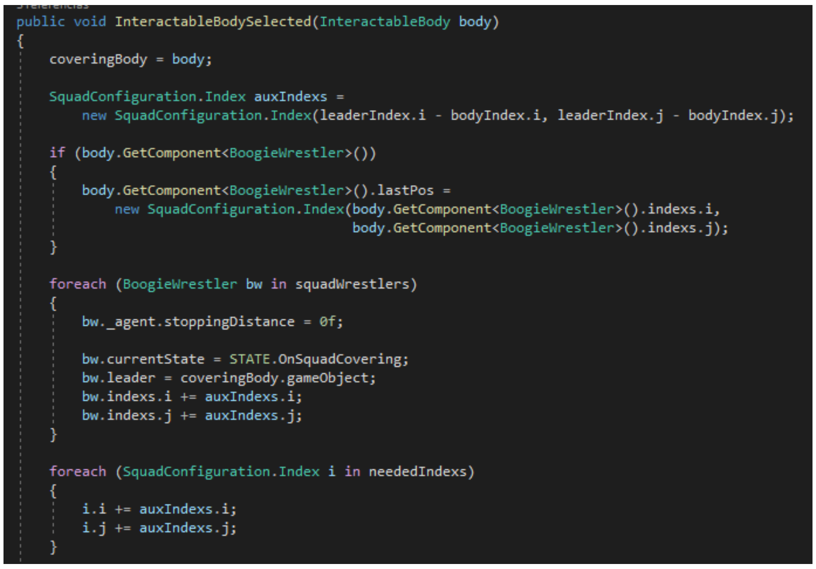
Figure 51: Check destination valid

7.4. Annex 4 (Squad actions)

In the following annex it is going to be explained some actions performed by the wrestler I consider interesting to explain in order to fully understand how the internal system of the squads works.

7.4.1. Squad cover

The whole squad covers a selected body. For doing this, for each member of the squad, their indexes in the internal matrix are modified adding the difference between the current leader indexes and the body selected to cover indexes. This difference is going to be called *auxIndexes.* Keep in mind that there is a list called *neededIndexs* which contains the indexes corresponding to the positions that those wrestlers who have left the squad. If any wrestler want to join this squad this list has to has at least one element, and the wrestler who wants to join will take this indexes. It is important to modify these stored indexes with the *auxIndexes* too. This process can be seen in Figure 52.

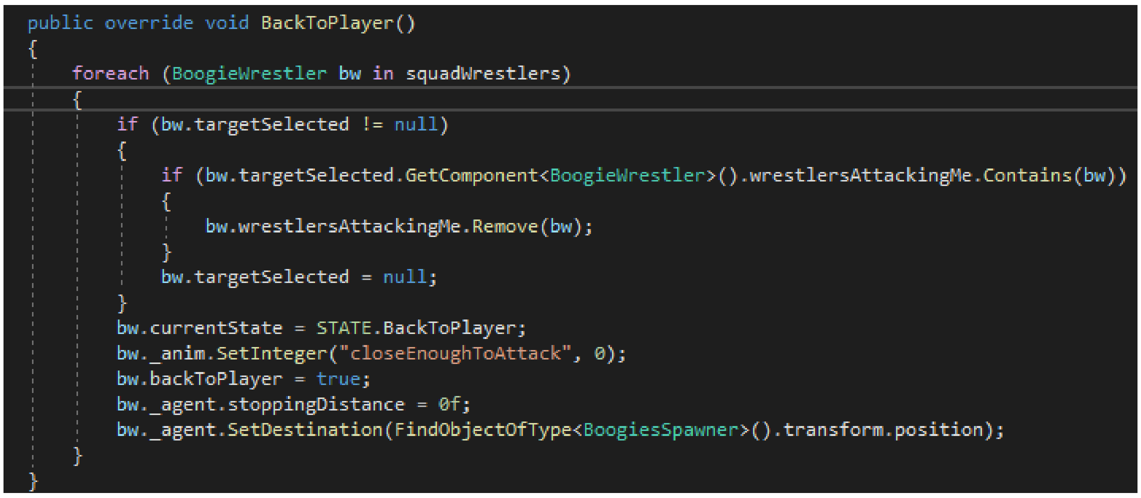


```
public void InteractableBodySelected(InteractableBody body)
{
    coveringBody = body;

    SquadConfiguration.Index auxIndexs =
        new SquadConfiguration.Index(leaderIndex.i - bodyIndex.i, leaderIndex.j - bodyIndex.j);

    if (body.GetComponent<BoogieWrestler>())
    {
        body.GetComponent<BoogieWrestler>().lastPos =
            new SquadConfiguration.Index(body.GetComponent<BoogieWrestler>().indexs.i,
                                         body.GetComponent<BoogieWrestler>().indexs.j);
    }

    foreach (BoogieWrestler bw in squadWrestlers)
    {
        bw._agent.stoppingDistance = 0f;

        bw.currentState = STATE.OnSquadCovering;
        bw.leader = coveringBody.gameObject;
        bw.indexs.i += auxIndexs.i;
        bw.indexs.j += auxIndexs.j;
    }

    foreach (SquadConfiguration.Index i in neededIndexs)
    {
        i.i += auxIndexs.i;
        i.j += auxIndexs.j;
    }
```

Figure 52: Squad selects body to cover

7.4.2. Squad returns to the player

All the members of the squad set their destination to the player. If they were attacking a target this target will be set to null and their state will change to *STATE.BackToPlayer.* This process can be seen in Figure 53.

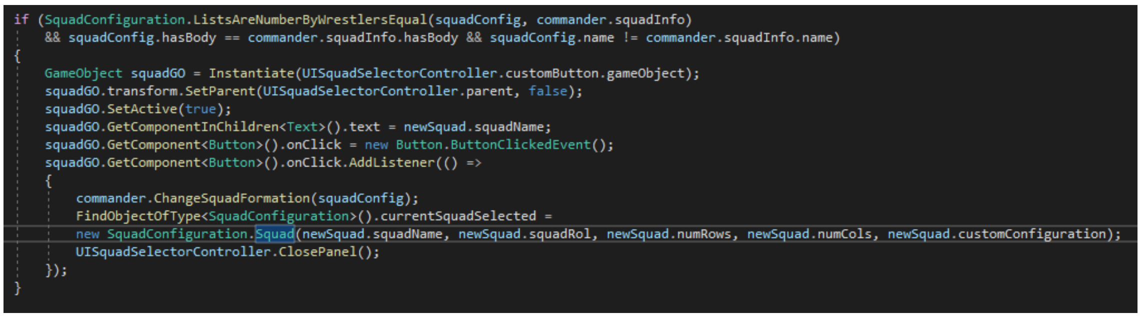


```
public override void BackToPlayer()
{
    foreach (BoogieWrestler bw in squadWrestlers)
    {
        if (bw.targetSelected != null)
        {
            if (bw.targetSelected.GetComponent<BoogieWrestler>().wrestlersAttackingMe.Contains(bw))
            {
                bw.wrestlersAttackingMe.Remove(bw);
            }
            bw.targetSelected = null;
        }
        bw.currentState = STATE.BackToPlayer;
        bw._anim.SetInteger("closeEnoughToAttack", 0);
        bw.backToPlayer = true;
        bw._agent.stoppingDistance = 0f;
        bw._agent.SetDestination(FindObjectOfType<BoogiesSpawner>().transform.position);
    }
}
```

Figure 53: Squad back to player

7.4.3. Squad changes formation

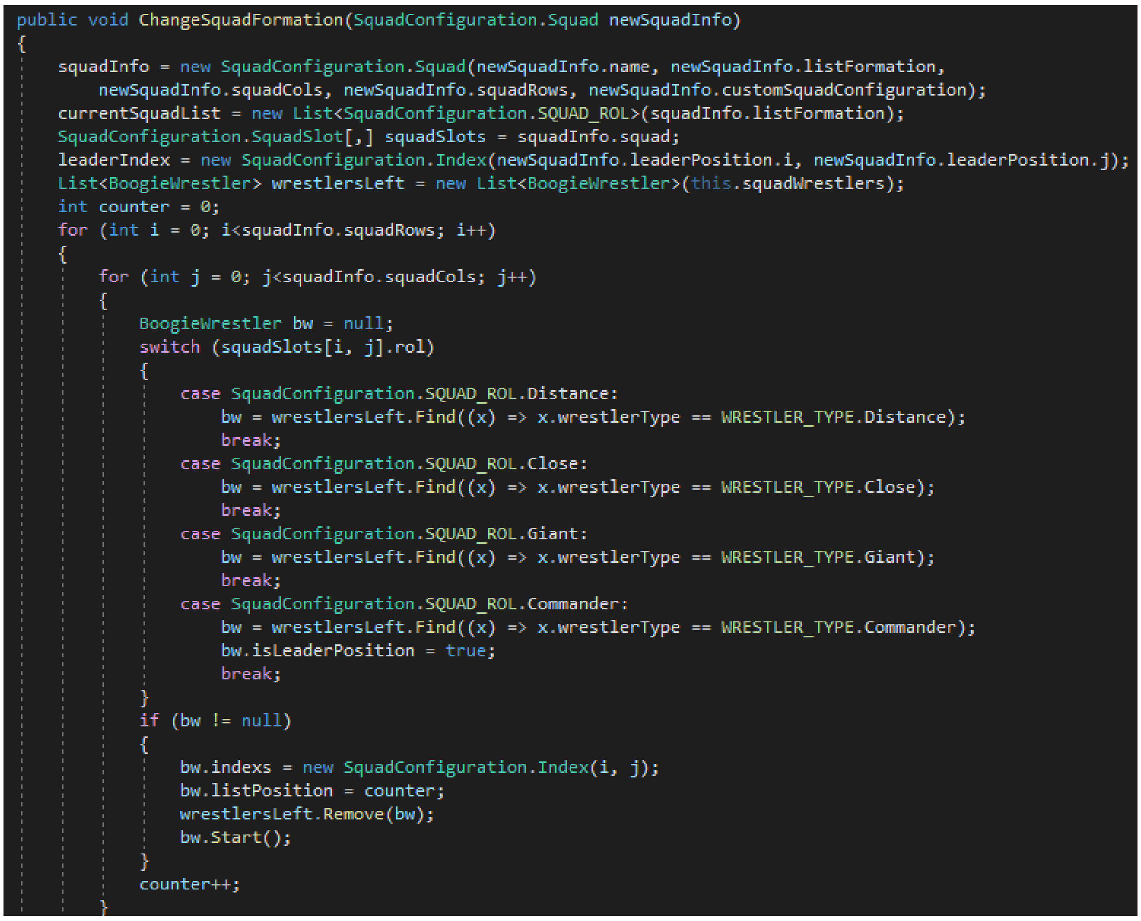
The squad changes the formation to the selected new one. These formations are shown in a popup. In order to be able to change to a new formation, this new formation has to have the same number of each type of wrestlers than the current one. Besides, they both have to allow (or not) the squad cover body. So, when the action Change Formation is selected, the popup loads all the formations that meet these conditions. This load is performed using the *LoadAssetAtPath* explained in the previous points: it loads all the possible Squads formations from the directory where all the created squads' formations are stored.



```
if (SquadConfiguration.ListsAreNumberByWrestlersEqual(squadConfig, commander.squadInfo)
    && squadConfig.hasBody == commander.squadInfo.hasBody && squadConfig.name != commander.squadInfo.name)
{
    GameObject squadGO = Instantiate(UISquadSelectorController.customButton.gameObject);
    squadGO.transform.SetParent(UISquadSelectorController.parent, false);
    squadGO.SetActive(true);
    squadGO.GetComponentInChildren<Text>().text = newSquad.squadName;
    squadGO.GetComponent<Button>().onClick = new Button.ButtonClickedEvent();
    squadGO.GetComponent<Button>().onClick.AddListener(() =>
    {
        commander.ChangeSquadFormation(squadConfig);
        FindObjectOfType<SquadConfiguration>().currentSquadSelected =
        new SquadConfiguration.Squad(newSquad.squadName, newSquad.squadRol, newSquad.numRows, newSquad.numCols, newSquad.customConfiguration);
        UISquadSelectorController.ClosePanel();
    });
}
```

Figure 54: Call to change formation

As it can be seen in Figure 54, for each squad stored in the directory it is checked if it meets the condition of equal number of wrestlers and both of them allow the squad cover action. This is, all this was to show the available squads to change with the formation. In Figure 55 we can see how this change is performed.



```
public void ChangeSquadFormation(SquadConfiguration.Squad newSquadInfo)
{
    squadInfo = new SquadConfiguration.Squad(newSquadInfo.name, newSquadInfo.listFormation,
        newSquadInfo.squadCols, newSquadInfo.squadRows, newSquadInfo.customSquadConfiguration);
    currentSquadList = new List<SquadConfiguration.SQUAD_ROL>(squadInfo.listFormation);
    SquadConfiguration.SquadSlot[,] squadSlots = squadInfo.squad;
    leaderIndex = new SquadConfiguration.Index(newSquadInfo.leaderPosition.i, newSquadInfo.leaderPosition.j);
    List<BoogieWrestler> wrestlersLeft = new List<BoogieWrestler>(this.squadWrestlers);
    int counter = 0;
    for (int i = 0; i<squadInfo.squadRows; i++)
    {
        for (int j = 0; j<squadInfo.squadCols; j++)
        {
            BoogieWrestler bw = null;
            switch (squadSlots[i, j].rol)
            {
                case SquadConfiguration.SQUAD_ROL.Distance:
                    bw = wrestlersLeft.Find((x) => x.wrestlerType == WRESTLER_TYPE.Distance);
                    break;
                case SquadConfiguration.SQUAD_ROL.Close:
                    bw = wrestlersLeft.Find((x) => x.wrestlerType == WRESTLER_TYPE.Close);
                    break;
                case SquadConfiguration.SQUAD_ROL.Giant:
                    bw = wrestlersLeft.Find((x) => x.wrestlerType == WRESTLER_TYPE.Giant);
                    break;
                case SquadConfiguration.SQUAD_ROL.Commander:
                    bw = wrestlersLeft.Find((x) => x.wrestlerType == WRESTLER_TYPE.Commander);
                    bw.isLeaderPosition = true;
                    break;
            }
            if (bw != null)
            {
                bw.indexs = new SquadConfiguration.Index(i, j);
                bw.listPosition = counter;
                wrestlersLeft.Remove(bw);
                bw.Start();
            }
            counter++;
        }
```

Figure 55: Change formation

This method works as following: the whole internal matrix of the objective formation is traversed. For each element, we find in our current wrestlers one that has the same role. When we find it, we assign it the new indexes and it is deleted from the auxiliary list (*wrestlersLeft*) used for store our wrestlers that have not been changed yet. Besides, this is assigned the *listPosition* index. This is, the index in the list of roles. Remember that this list is unique for each squad and it contains the representation of roles of each squad.

7.4.4. Wrestler follows player

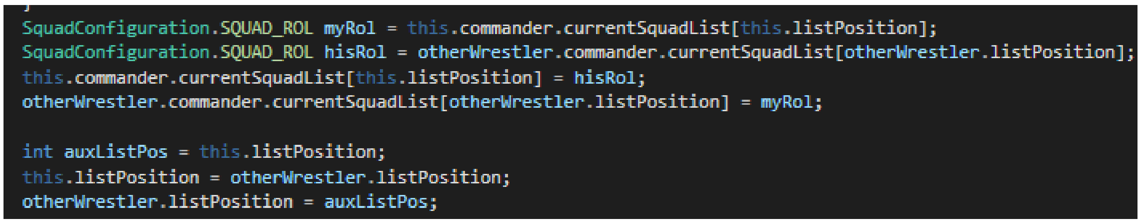
The follow player action consists in set the destination of the selected wrestler in a random point inside a circle with center on the player's position. Each time this position is reached, a new position is assigned. The hardest part here is that the wrestler breaks the formation so his/her indexes inside the matrix have to be added to the needed indexes list so he/she will be able to join the squad in the future. There are two cases to keep in mind.

- The selected wrestler to follow player was the current leader: in this case the commander is assigned as leader. All the wrestlers in the squad change their position according to it.

- The selected wrestler was being attacked by other wrestler or was attacking him: in this case the attack target is set to null.

7.4.5. Wrestler changes formation position

In order to change the position of a wrestler, another wrestler needs to be selected so they both will interchange positions. For doing this, they change their indexes, their index in the list of roles and the role inside the list (Figure 56).



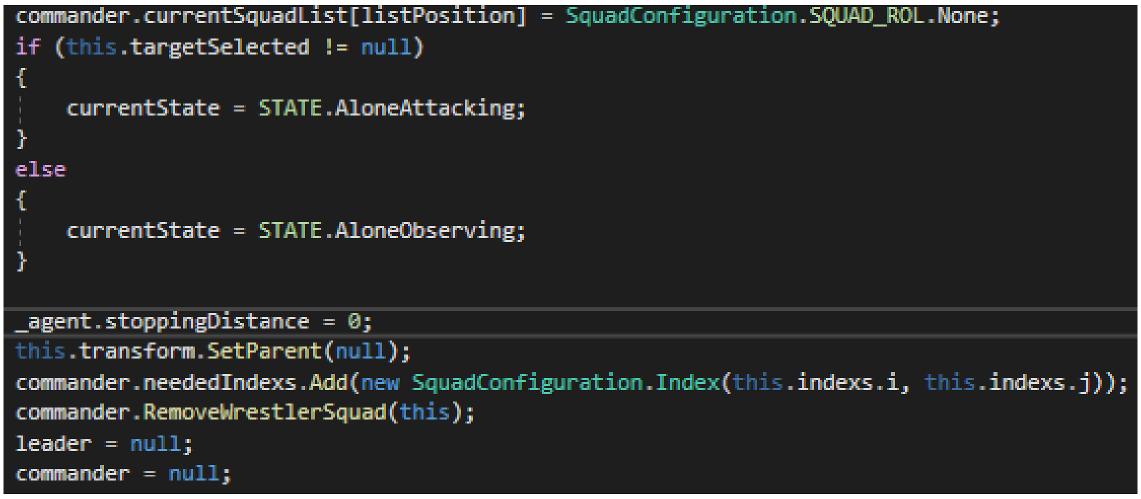
```
SquadConfiguration.SQUAD_ROL myRol = this.commander.currentSquadList[this.listPosition];
SquadConfiguration.SQUAD_ROL hisRol = otherWrestler.commander.currentSquadList[otherWrestler.listPosition];
this.commander.currentSquadList[this.listPosition] = hisRol;
otherWrestler.commander.currentSquadList[otherWrestler.listPosition] = myRol;

int auxListPos = this.listPosition;
this.listPosition = otherWrestler.listPosition;
otherWrestler.listPosition = auxListPos;
```

Figure 56: Assignments

As in the previous action, it must be considered if one of both wrestlers is the leader. In that case, the one who is changing with the leader now will be the leader and will perform the action *AssignAsLeader().* This action will be explained later.

7.4.6. Wrestler breaks formation

This action is similar to the follow player one. What it does is to store the selected wrestler's indexes in the needed indexes list, set to null the commander and current leader references and change the current state to *STATE.AloneObserving.* Besides, the list of roles has to be modified changing the entry of this wrestler to *Rol.None.*

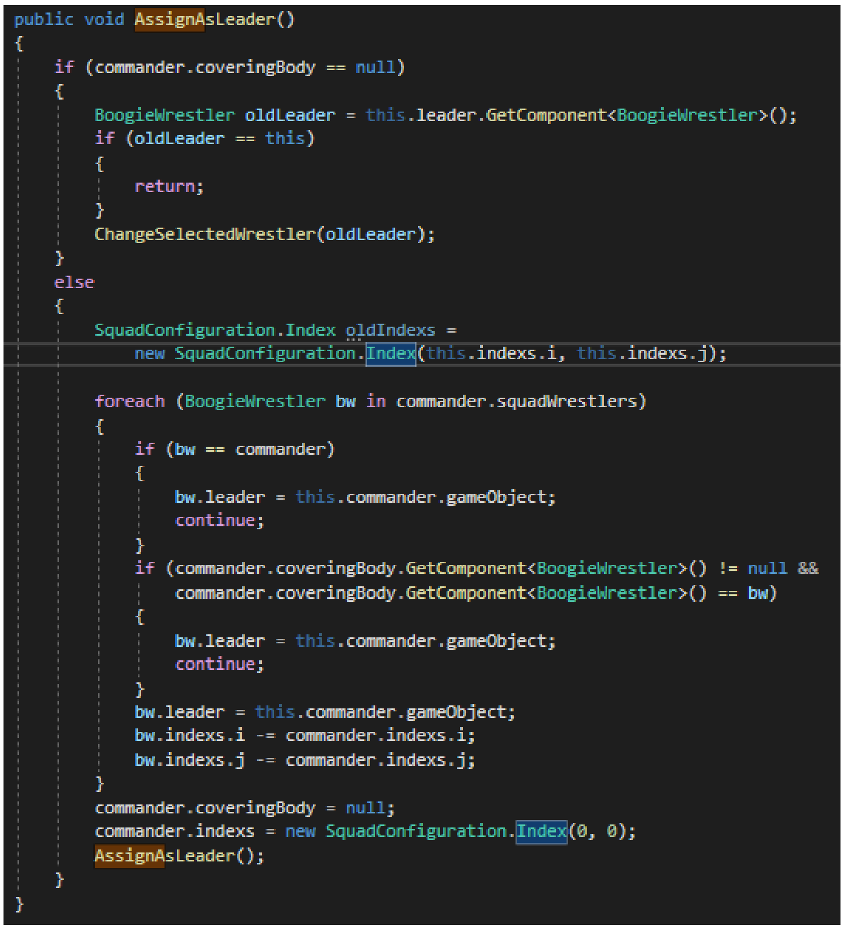


```
commander.currentSquadList[listPosition] = SquadConfiguration.SQUAD_ROL.None;
if (this.targetSelected != null)
{
    currentState = STATE.AloneAttacking;
}
else
{
    currentState = STATE.AloneObserving;
}

_agent.stoppingDistance = 0;
this.transform.SetParent(null);
commander.neededIndexs.Add(new SquadConfiguration.Index(this.indexs.i, this.indexs.j));
commander.RemoveWrestlerSquad(this);
leader = null;
commander = null;
```

Figure 57: Break formation action

7.4.7. Wrestler assigned as leader

In this action a wrestler is assigned as a leader. That is, the whole squad will move and act relative to him.
For doing this, all the indexes have to be modified according to it. In Figure 58 we can see the process.



```
public void AssignAsLeader()
{
    if (commander.coveringBody == null)
    {
        BoogieWrestler oldLeader = this.leader.GetComponent<BoogieWrestler>();
        if (oldLeader == this)
        {
            return;
        }
        ChangeSelectedWrestler(oldLeader);
    }
    else
    {
        SquadConfiguration.Index oldIndexs =
            new SquadConfiguration.Index(this.indexs.i, this.indexs.j);

        foreach (BoogieWrestler bw in commander.squadWrestlers)
        {
            if (bw == commander)
            {
                bw.leader = this.commander.gameObject;
                continue;
            }
            if (commander.coveringBody.GetComponent<BoogieWrestler>() != null &&
                commander.coveringBody.GetComponent<BoogieWrestler>() == bw)
            {
                bw.leader = this.commander.gameObject;
                continue;
            }
            bw.leader = this.commander.gameObject;
            bw.indexs.i -= commander.indexs.i;
            bw.indexs.j -= commander.indexs.j;
        }
        commander.coveringBody = null;
        commander.indexs = new SquadConfiguration.Index(0, 0);
        AssignAsLeader();
    }
}
```

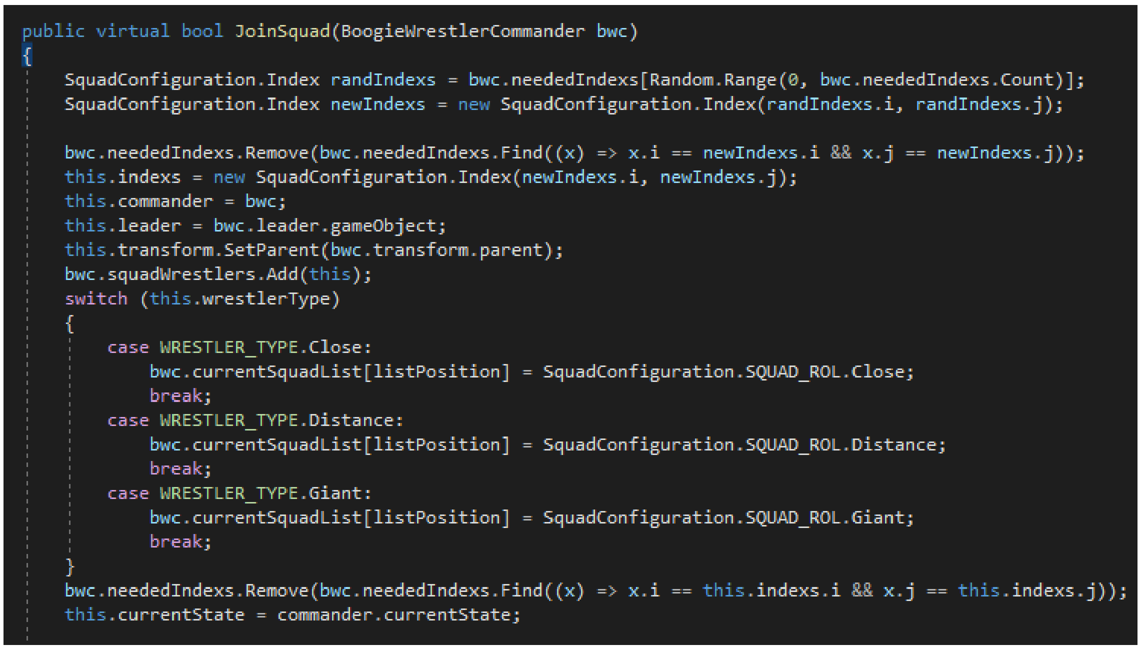
Figure 58: AssignAsLeader()

As it can be seen, it depends on if the squad was covering a body or not. If they were not doing it, it is as simple as call the method *ChangeSelectedWrestler()* shown in Figure 56 passing via parameter the old leader. This is, changing the position with the old leader.

In case the squad was covering a body, the first assign as leader the commander and modify the whole indexes in function of the commander indexes. After doing this they set the commander indexes to (0,0) and set *coveringBody* to *null* and call the method *AssignAsLeader()* again. The difference is that now there is no body being covered so it will proceed as explained in the paragraph above.

7.4.8. Wrestler joins a squad

A wrestler can join a squad if the needed indexes list has at least one element. If it has more than one element it selects one randomly. It sets the current leader of the squad, the commander, it is added to the list of wrestlers that conform the squad and it is modified the list of roles in function of the role of the wrestler and is changed his state to the commander current state. At the end, the index now busy by this wrestler is deleted from the list.



```
public virtual bool JoinSquad(BoogieWrestlerCommander bwc)
{
    SquadConfiguration.Index randIndexs = bwc.neededIndexs[Random.Range(0, bwc.neededIndexs.Count)];
    SquadConfiguration.Index newIndexs = new SquadConfiguration.Index(randIndexs.i, randIndexs.j);

    bwc.neededIndexs.Remove(bwc.neededIndexs.Find((x) => x.i == newIndexs.i && x.j == newIndexs.j));
    this.indexs = new SquadConfiguration.Index(newIndexs.i, newIndexs.j);
    this.commander = bwc;
    this.leader = bwc.leader.gameObject;
    this.transform.SetParent(bwc.transform.parent);
    bwc.squadWrestlers.Add(this);
    switch (this.wrestlerType)
    {
        case WRESTLER_TYPE.Close:
            bwc.currentSquadList[listPosition] = SquadConfiguration.SQUAD_ROL.Close;
            break;
        case WRESTLER_TYPE.Distance:
            bwc.currentSquadList[listPosition] = SquadConfiguration.SQUAD_ROL.Distance;
            break;
        case WRESTLER_TYPE.Giant:
            bwc.currentSquadList[listPosition] = SquadConfiguration.SQUAD_ROL.Giant;
            break;
    }
    bwc.neededIndexs.Remove(bwc.neededIndexs.Find((x) => x.i == this.indexs.i && x.j == this.indexs.j));
    this.currentState = commander.currentState;
```

Figure 59: JoinSquad()

There are more actions performed by the wrestlers but once knew how these functions work the whole squad system is understandable so showing them and explaining its code does not give us more useful information. For ending this annex, we are going to see a demo of the squads in which we are able to see all these actions together (Figure 60) and another one paying more attention to how the squads are able to change their formation (Figure 61).

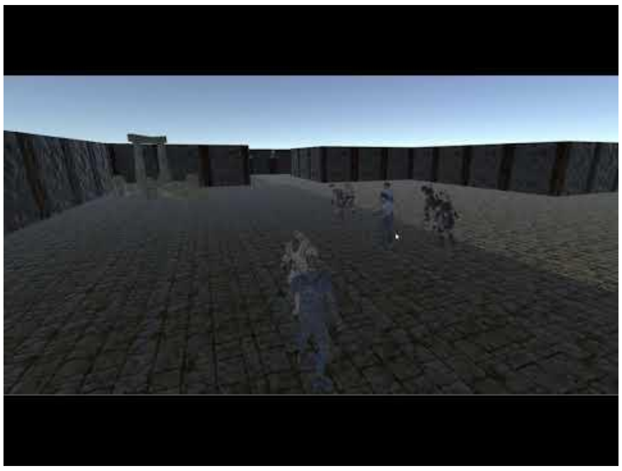


Figure 60: Squad actions demo

[Video 10]

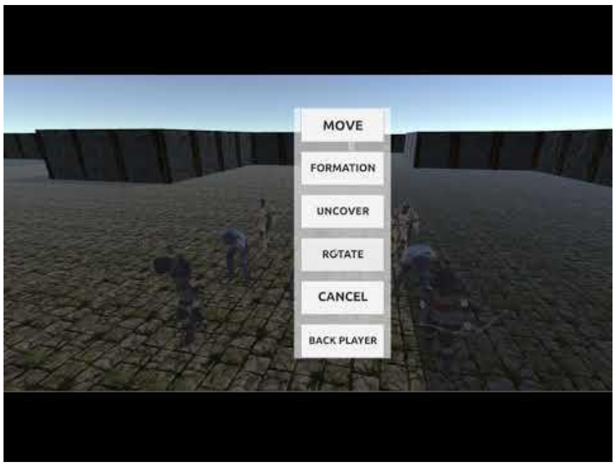


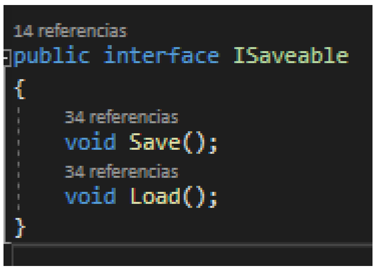
Figure 61: Squad formation changes demo

[Video 11]

7.5. Annex 5 (Saving system)

The functionality of the saving system is the following: an interface called *ISaveable* that has two methods. *Load()* and *Save().* All the objects or entities in the game that can be affected when saving and loading implements this interface. For example, all the NPC's have it, also the obstacles and the player.

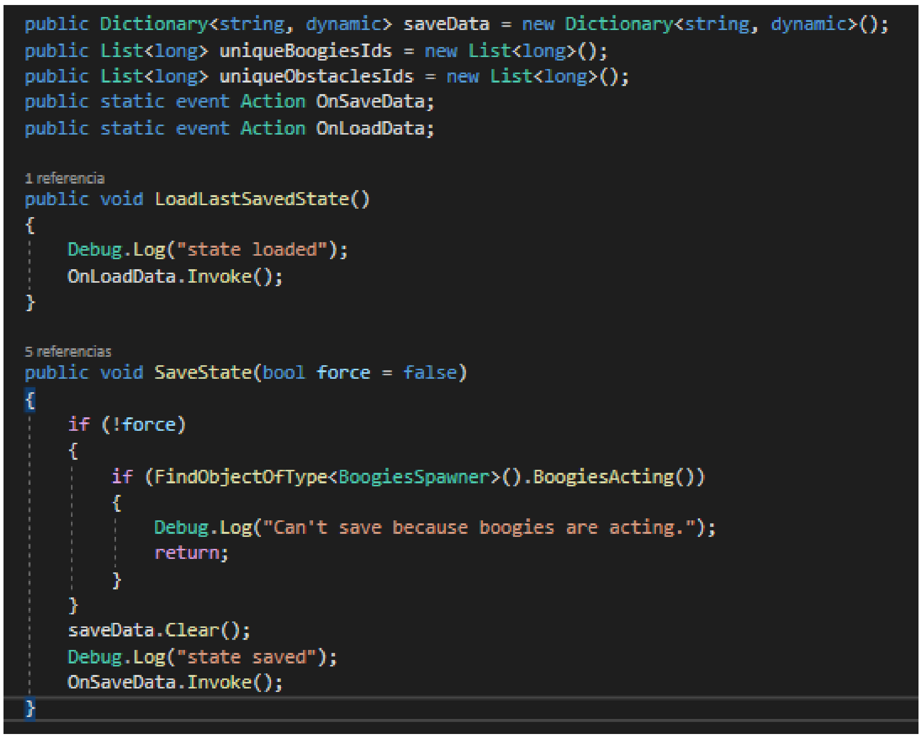
In Figure 106 it is shown how the interface *ISaveable* is.



```
14 referencias
public interface ISaveable
{
    34 referencias
    void Save();
    34 referencias
    void Load();
}
```

Figure 62: ISaveable

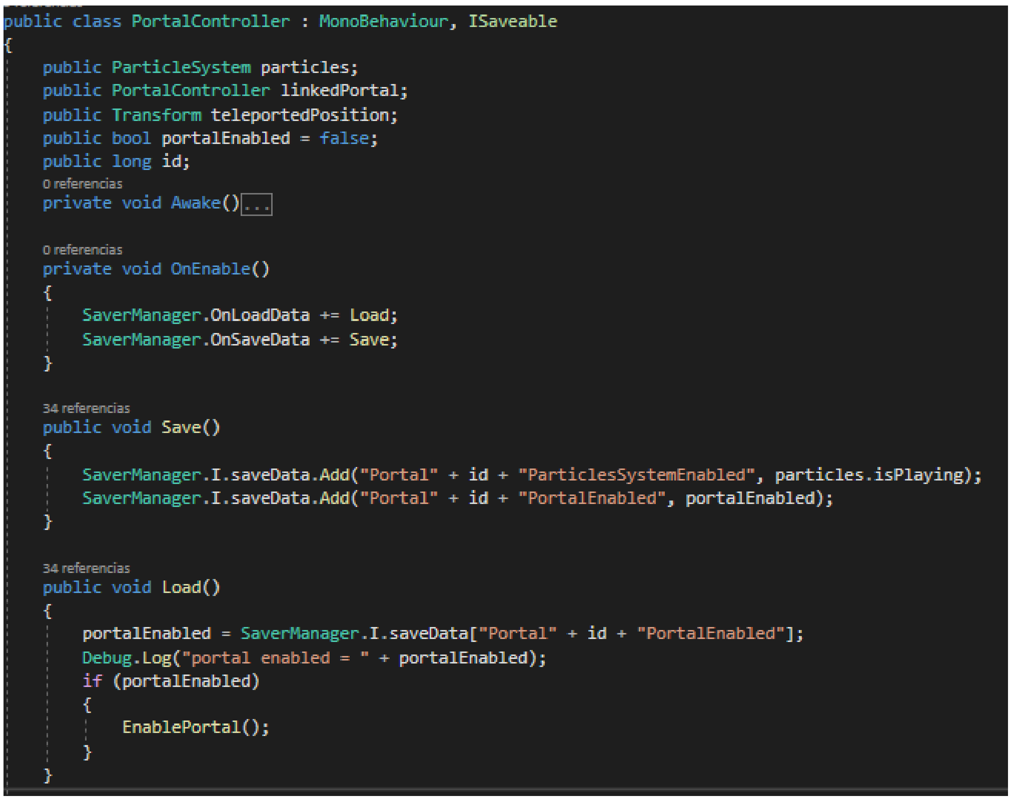
There is a controller called *SaverManager* that has two events, one (*OnLoadData*) that is invoked when the data stored has to be loaded and another one (*OnSaveData*) that is invoked when the data has to be stored. All the data is stored using a dictionary. The key of a dictionary in C# has to be unique so every entity uses a single id declared as soon as they are created. Besides, it is important to see that the dictionary is of type <string, dynamic>. This means that the value can be of different types (Vector3, string, int, long, etc.). In Figure 63 it can be seen how this controller works.



```
public Dictionary<string, dynamic> saveData = new Dictionary<string, dynamic>();
public List<long> uniqueBoogiesIds = new List<long>();
public List<long> uniqueObstaclesIds = new List<long>();
public static event Action OnSaveData;
public static event Action OnLoadData;

1 referencia
public void LoadLastSavedState()
{
    Debug.Log("state loaded");
    OnLoadData.Invoke();
}

5 referencias
public void SaveState(bool force = false)
{
    if (!force)
    {
        if (FindObjectOfType<BoogiesSpawner>().BoogiesActing())
        {
            Debug.Log("Can't save because boogies are acting.");
            return;
        }
    }
    saveData.Clear();
    Debug.Log("state saved");
    OnSaveData.Invoke();
}
```

Figure 63: SaverManager

As it can be seen, it is not allowed to save while the Boogies are acting. For ending this point, it is going to be shown how an entity (the Portal, for example) implements this interface and which data is suitable to be stored in the dictionary.



```
public class PortalController : MonoBehaviour, ISaveable
{
    public ParticleSystem particles;
    public PortalController linkedPortal;
    public Transform teleportedPosition;
    public bool portalEnabled = false;
    public long id;
    0 referencias
    private void Awake()...

    0 referencias
    private void OnEnable()
    {
        SaverManager.OnLoadData += Load;
        SaverManager.OnSaveData += Save;
    }

    34 referencias
    public void Save()
    {
        SaverManager.I.saveData.Add("Portal" + id + "ParticlesSystemEnabled", particles.isPlaying);
        SaverManager.I.saveData.Add("Portal" + id + "PortalEnabled", portalEnabled);
    }

    34 referencias
    public void Load()
    {
        portalEnabled = SaverManager.I.saveData["Portal" + id + "PortalEnabled"];
        Debug.Log("portal enabled = " + portalEnabled);
        if (portalEnabled)
        {
            EnablePortal();
        }
    }
```

Figure 64: PortalController

As it can be seen in Figure 64, as soon as the object is enabled (*OnEnable()* method), the *Save()* and *Load()* methods are subscribed to the events. The *Save()* method stores information as the state of the particle system of the portal and a Boolean that indicates if it is enabled or not.

7.6. Annex 6: Demo Game Design Document (GDD)

In order to show all the features explained in the previous points, a playable demo has been created. In this point it is going to be explained everything about this demo. Besides, doing this demo we are achieving the objective of incorporate these emergency complex behaviors in a video game and using them as an active element of the game.

It is important to remark that the main purpose of the project was to develop a back-end system which is based in the develop of the emergency behaviors applied to multi-agent systems and apply how to apply them into active elements in a videogame such as characters that help the player or squad systems that can battle between them.

The game is called “The Cursed Dungeon” and it is created with the objective of showing how these behaviors work. Using the system created more games and mechanics could be easily created.

7.6.1. Narrative design

We take control of Jack. A soldier of the division A-85 who has the mission of rescuing the people who are left alive in a Dungeon where an unknown infection is mutating people and converting them in monsters. Besides, there is another division, the A-82, that already tried this mission, but they failed.

Jack is going to know an ancient who will say him to rescue her daughter Aidil that is in the dungeon. For doing it, the ancient says Jack that he was scientist and he did some advances in how to genetically modify the mutants in order to obey his orders. He will give Jack some creatures that will help him to find his daughter.

Besides, Jack will know three different types of entities that will help him to overcome the dungeon’s obstacles: cleaners, collectors and explorers. During it, Jack finds some soldiers from the A-82 and they join Jack's army. When Jack finally arrives where Aidil is, she tells him the truth. Her real name is not Aidil as the ancient said, it is Roxie; and she is not the daughter of the ancient. Besides, she tells Jack that all these modified mutants have been modified by her and she tells him that the ancient is called Lodwig. Lodwig is a liar and the only thing he wants is to torture Roxie to obtain all the advances she did with the mutants.

Once Jack know this, he goes back to face Lodwig but he finds that there are a whole enemy squad waiting for him. Jack, using the mutants that Lodwig gave him, with the soldiers of the A-82 and with some help sent by Roxie, starts an epic battle. When Jack and his squad defeats all the enemies, Lodwig dies.

7.6.2. Characters

- Jack: Jack is a soldier from the division A-85. His mission is to rescue people trapped in that dungeon. (Figure 65)

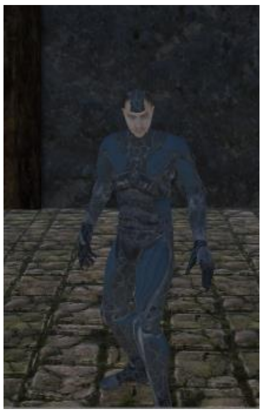


Figure 65: Jack

- Ancient/Lodwig: Lodwig is a liar that is trying to get the advances of Roxie in the genetic modifications of the mutants. He wants to create an army of mutants that obey his orders. (Figure 66)

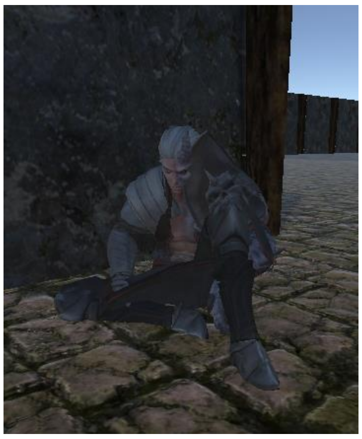


Figure 66: Ancient/Lodwig

- Aidil/Roxie: Roxie is a member of the A-82 that was trapped here and she started to investigate about how to genetically modify the mutants of the dungeon in order to survive there. She says that her purpose is to help people.

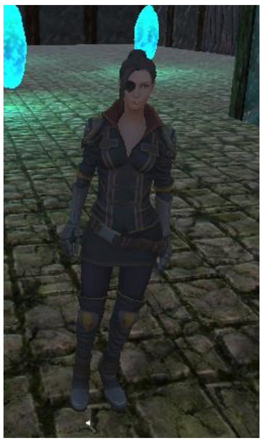


Figure 67: Aidil/Roxie

- Boogies: the modified creatures of the dungeon. This is, the cleaners, the collectors, the explorers and the wrestlers who fight for helping Jack.

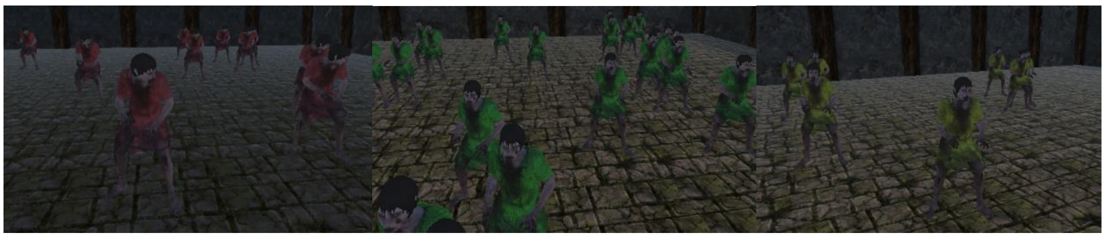


Figure 68: Boogies collectors, cleaners, explorers, respectively

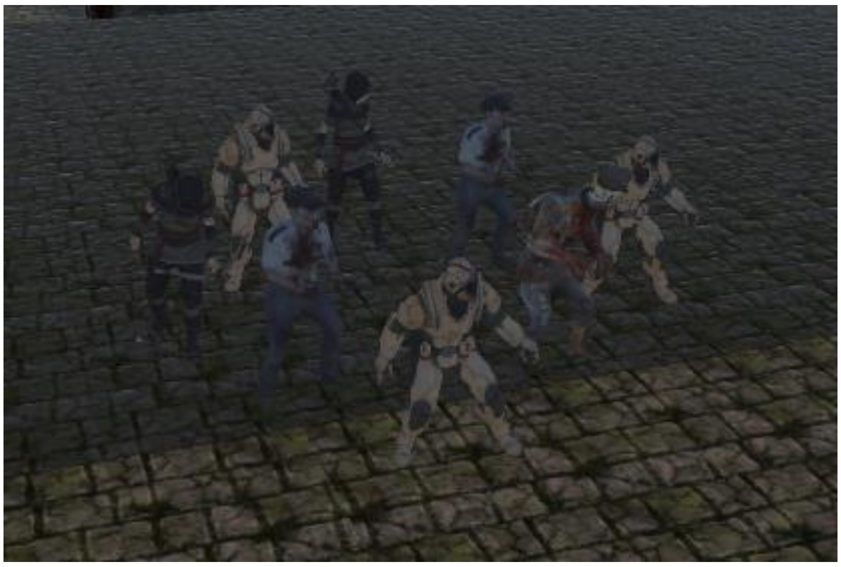


Figure 69: Boogie wrestlers squad

- Mutants: the mutants modified by the ancient. They obey the ancient Lodwig.

Figure 70: Enemy mutants

7.6.3. Environment

The environment mechanics are subject to the obstacles mentioned in section 3.2. The first obstacle we find is the Cleaning Obstacle. There are a lot of boxes and barrels that are blocking the path, so we need the Cleaners to clean it. The second obstacle is the multipath, once we have the Explorers it is easy to complete it and then we arrive to a zone in which it is necessary to activate a portal. This portal can only be enabled if a generator (collecting machine) is full of energy (elixir) from the energy sources (elixir stones). This is the Collecting Obstacle.

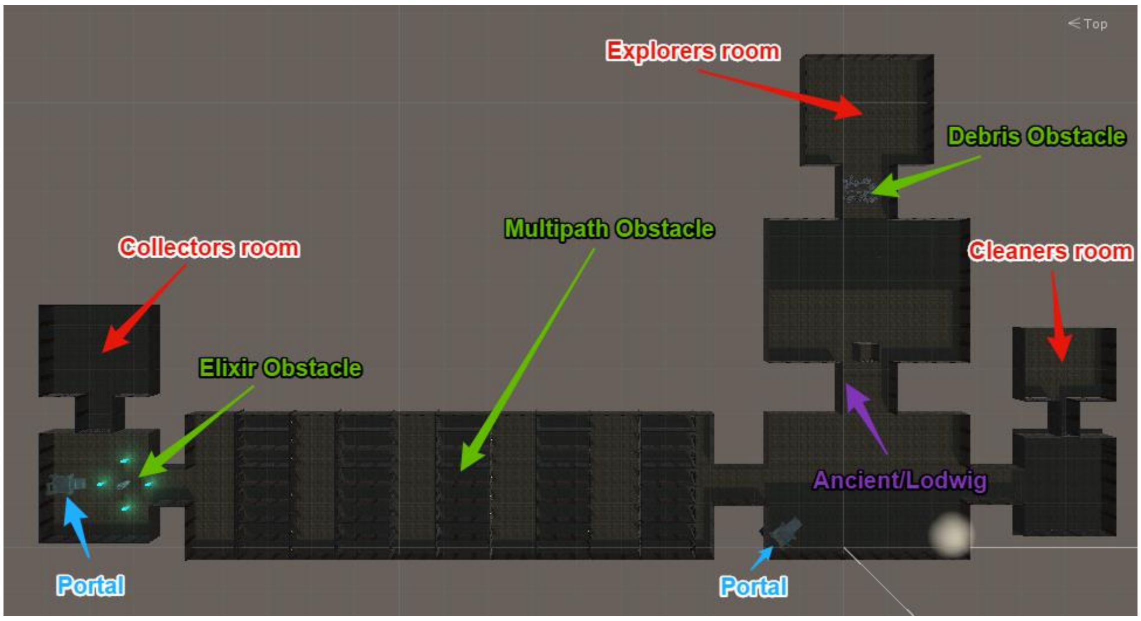


Figure 71: Game Scenario

All these obstacles require to have the proper Boogies that are able to solve it. Initially, the collectors, the cleaners and the explorers are spared in different rooms so in order to be able to take them it is necessary to unlock these rooms.

As it can be seen in Figure 71, there are two portals that are communicated between them. In Figure 72 there is an image of them.

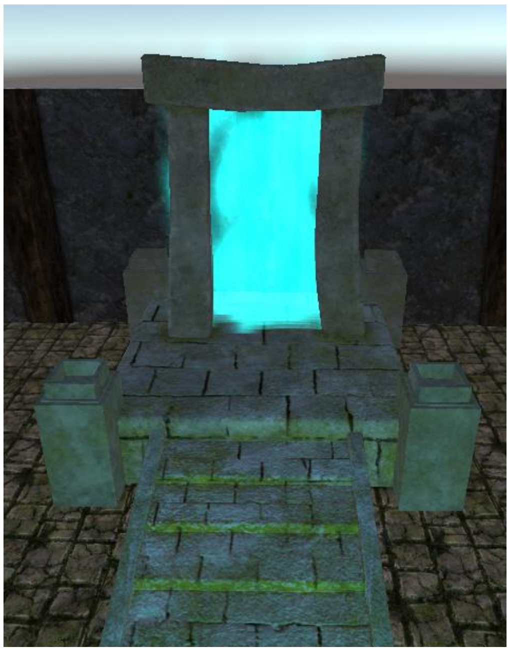


Figure 72: Portal

7.6.4. Playability

The playability is limited to the Boogies actions. Here are all the mechanics of the player. We divide it into active and passive mechanics.

7.6.4.1. Active mechanics

- Move and watch around: the controller of the player has been done by me and it is an interesting part of the game because it helps a lot in order to see the AI behaviors and see how they change when the player is moving.

- Spawn boogies: the player has a component *BoogiesSpawner* that spawns the boogies in a position selected by the player. This is not a boogies controller because it just serves to spawn the boogies. This class has a counter of all the entities the player has selected. The player selects it using a UI popup and then selects a position to spawn those, he has selected.

- Interact with squads: the player can choose the squad where to go, which formation to adopt, etc.

7.6.4.2. Passive mechanics

- Be defended by allies: the player can be defended by the squads if the player set it on the squad. Besides, the behavior of the allies will be differently according of the life of the player.

- Helped by the boogies: the practical approach of the AI behaviors is to help the player to overcome obstacles. For example, to spawn the Boogies Cleaners to clean the path of debris in order to be able to continue the path.

- Killed by enemies: enemies can kill the player, so the player will have to be careful in order to handle the boogies in that situation.

7.6.5. Art

All the art excepting the Particle System used in the portal and the Multipath clues have been taken from external sources. These sources are listed here.

- Floors, walls, doors and roofs of the dungeon: these have been taken from a project of the subject Artificial Intelligence.
- 3D Models for Player, Aidil, Ancient, Collectors, Cleaners and ally wrestlers: These have been taken from a webpage called Mixamo (Adobe, 2019). These models also include animations. Nonetheless, the animations controllers of each model have been made by me.
- 3D Models for enemy wrestlers: These models have been taken from a Unity Asset Store pack that I bought. (Protofactor, s.f.).
- Portal: this have been taken from a webpage called CGTrader (medwuf, s.f.).
- Debris Obstacle parts: all the barrels and boxes have been taken from Unity Asset Store. (3D, s.f.)
- Elixir Obstacle generator: this model has been taken from CGTrader. (3DHaupt, s.f.)
- Multipath Doors: these doors have been taken from Free3D. (saladin0511, s.f.)
- Elixir stones: these stones have been taken from a project of the subject Artificial Intelligence.

7.6.6. HUD and Controls

The User Interface in this demo is really simple. It consists of a menu of the Collectors, Explorers and Cleaners available and some graphic elements that serve as a marker to know where to spawn or where to move the entities. This menu at the beginning of the game is empty. As we go finding the Boogies the corresponding part of the Menu will be shown and now, we are available to spawn them and control them.

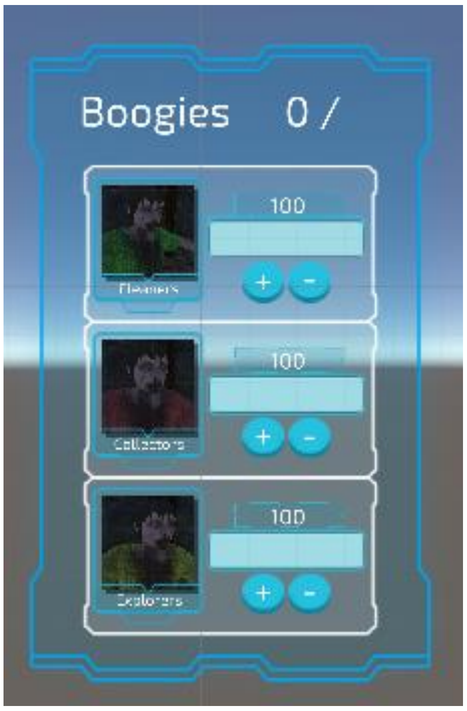


Figure 73: Boogies menu selector

The controls are quite simple. To move the player, it is used the W-A-S-D keys and to look around is the mouse. To select actions, options, positions, Boogies or other elements it is done with the left click of the mouse. With the right click of the mouse there are some actions that the wrestlers can do.

In order to spawn the collectors, the cleaners or the explorers it is done with the entity we want to spawn assigning a number superior than 0 in the menu explained above and pressing the F key. In order to see the ally's squads formations, it is done pressing the G key. To see the enemy squads' formations, it is done pressing the H key. Nonetheless, this is just available in the Sandbox scene, not in the demo of the game.

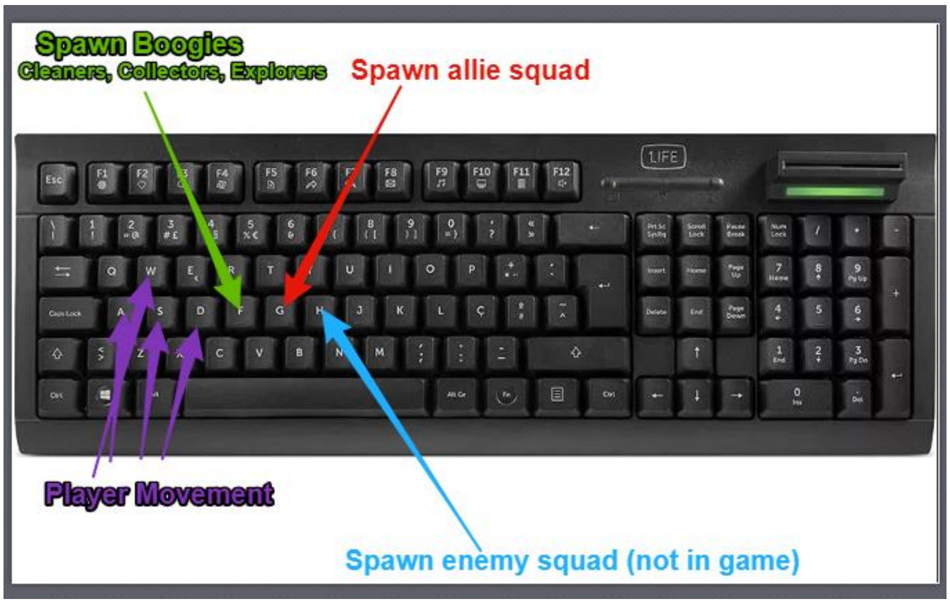

Spawn Boogies
Cleaners, Collectors, Explorers
Spawn allie squad
Player Movement
Spawn enemy squad (not in game)

Figure 74: Keyboard controls

Other element used in the demo are the dialogs. These are used in the conversation that Jack has with the ancient Lodwig, with Roxie and with the Giants. The dialogs are part of an imported Plugin called Fungus. (Fungus Games, s.f.)

Before finishing, it is import to talk about a feature created for the game. That is, a saving system created in order to be able to play again when the player dies. This system is just created for the in-game mode. In the future I will adapt it to save it in the local files of the PC. For this demo it is more than enough to store the in-game data and be able to load it when desired. It is explained in Annex 5.

Here is the video of the Demo of "The Cursed Dungeon", the demo created to show the project mechanics.

[Video 12]

7.7. Annex 7: Code architecture

In this section it is going to be explained the code architecture of some elements of the project. This is a very important point because one of the objectives of the project is to allow the future scalability, so in order to carry this out a good code architecture is needed.

This section is going to be divided in the subsections:

- Architecture of the entities (collectors, explorers, cleaners and wrestlers).
- Architecture of the obstacles (debris, elixir, multipath).
- Input and User Interface (UI) architecture.

7.7.1. Entities architecture

All the entities (collectors, cleaners, explorers and wrestlers) share some things so it is efficient to create a class that contains all these common behaviors. In this project, these entities are known as "Boogies".

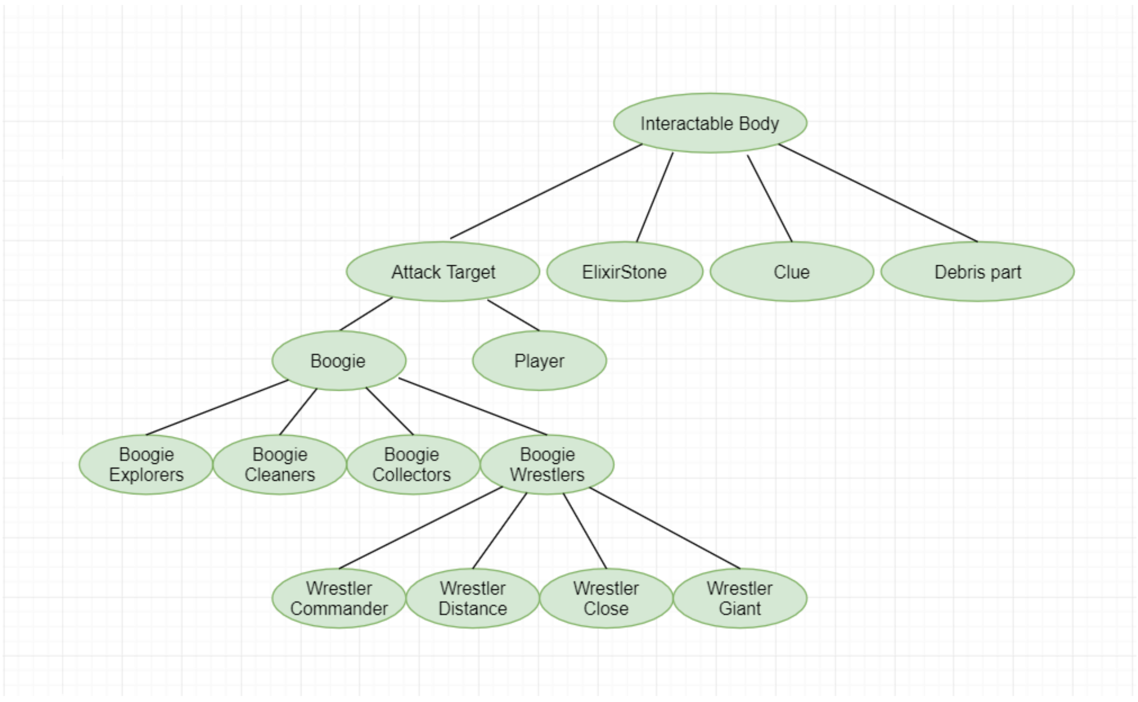


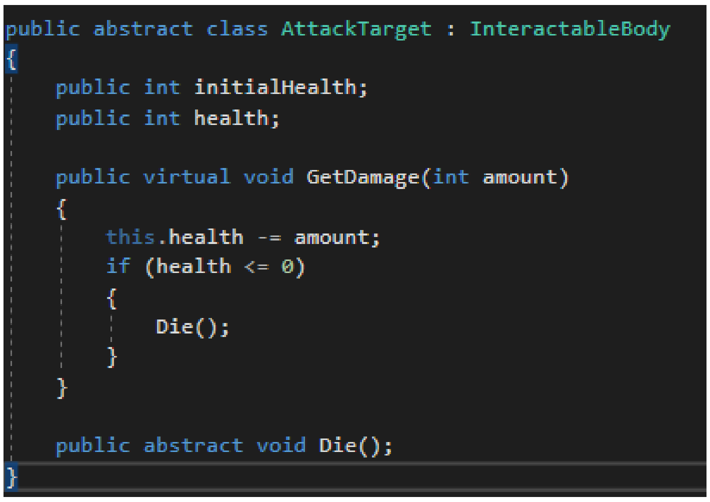
Figure 75: Class diagram (Zygomatic, s.f.)

The Interactable Body is the parent class. Those who can be covered by a wrestler are Interactable Body: The Elixir stone, the Clue and the Debris part too. The other child of Interactable body is *AttackTarget*. These are those who can be attacked by the wrestlers. Those who inherit from *AttackTarget* are those who inherit from *Boogie* and the *Player.* This means that the player can be attacked.

The *Boogie* class contains all the data that the entities that help the player have in common. That is, those who have emergent behaviors with the objective of helping the player. They have been explained in the previous points and they are the explorers, the cleaners, the collectors and the wrestlers. Besides, as there are several types of wrestlers, they all have common behaviors so they inherit from *BoogieWrestler*.

It is important to remark that if we have a class A which inherits from a class B, and this class B is inheriting from class C, A is inheriting from class C too. So, in the case of this project, a Boogie Wrestler Distance has all the behaviors from *BoogieWrestler*, from Boogie, from *AttackTarget* and from *InteractableBody*.

Some of the information that each of these parent classes contains it is going to be explained. First of all, it should be known that *InteractableBody* is an empty class so it is just for identifying that the object that has it attached to is interactable, that is, it can be covered by the wrestlers (remember the Squad actions).



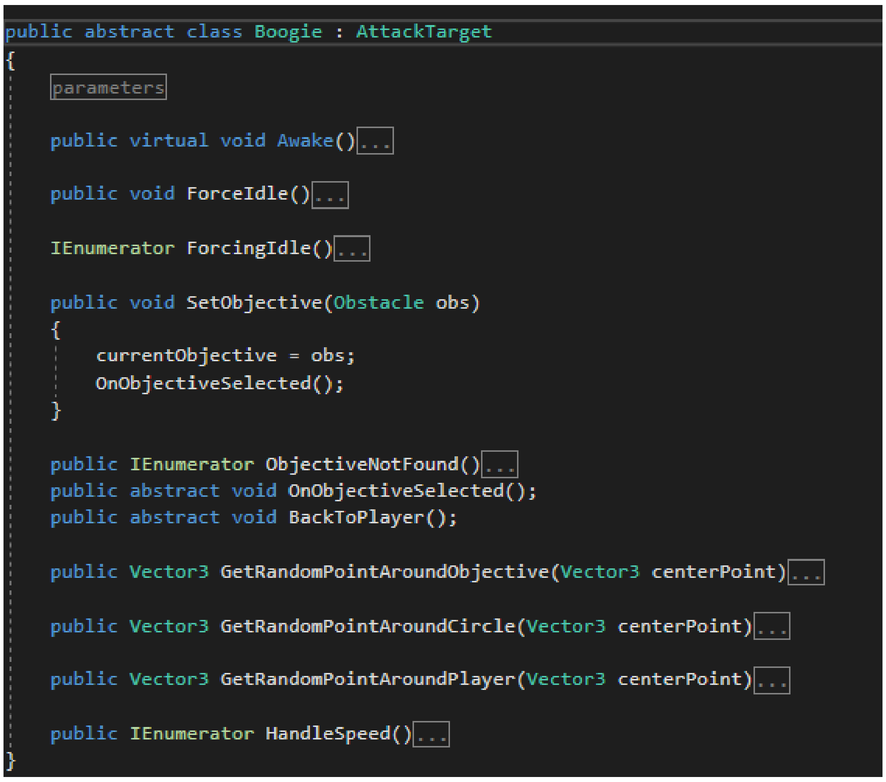
```
public abstract class AttackTarget : InteractableBody
{
    public int initialHealth;
    public int health;

    public virtual void GetDamage(int amount)
    {
        this.health -= amount;
        if (health <= 0)
        {
            Die();
        }
    }

    public abstract void Die();
}
```

Figure 76: AttackTarget.cs

AttackTarget is an abstract class. As we can see in Figure 76, those who can be attacked by the target has a method called *GetDamage(int amount)* and a *health* parameter. Besides, they have an abstract method called *Die().*

Before continuing, it is important to explain what is an abstract class. It is an "incompleted class" and those classes which inherit from this abstract class must complete it. This is why the *Die()* method has no information inside. It is expected to those classes which inherit from this class complete it, so, it is required that these classes implement this method. So every class that inherits from *AttackTarget* will have a *Die()* method with his own information.

The Boogie class is going to be explained because it is what matters most to us and has more relation with the main purpose of the project.



```
public abstract class Boogie : AttackTarget
{
    parameters

    public virtual void Awake()...

    public void ForceIdle()...

    IEnumerator ForcingIdle()...

    public void SetObjective(Obstacle obs)
    {
        currentObjective = obs;
        OnObjectiveSelected();
    }

    public IEnumerator ObjectiveNotFound()...
    public abstract void OnObjectiveSelected();
    public abstract void BackToPlayer();

    public Vector3 GetRandomPointAroundObjective(Vector3 centerPoint)...

    public Vector3 GetRandomPointAroundCircle(Vector3 centerPoint)...

    public Vector3 GetRandomPointAroundPlayer(Vector3 centerPoint)...

    public IEnumerator HandleSpeed()...
}
```

Figure 77: Boogie.cs

As we can see, this class contains information that all the different types of entites/Boogies are going to use. On the one hand it contains methods they use like:

- *SetObjective(Obstacle obs):* to assign the objective.
- *GetRandomPointAroundObjective*
- *GetRandomPointAroundCircle*
- *GetRandomPointAroundPlayer*
- *HandleSpeed:* handle the variations of speed.
- *ObjectiveNotFound*: what to do in case the maximum time to find an objective is elapsed.

There are some methods that has a base information and can be extended by those who inherit. These methods are those that have the world virtual in the declaration:

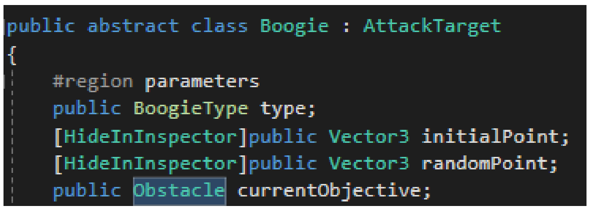
- *Awake*: first method that executes as soon as the object is created.

And finally, it contains some methods that are just declared (the abstracts) and those who inherit from this class are obligated to implement it. Remember that if the direct "son" of this class does not implement it, the "grandchild" can do it. The inheritance allows multiple generations. In this case:

- *OnObjectiveNotFound:* all those who inherits from this class have to do something when they do not find the objective.
- *ObjectiveSelected:* all those who inherits from this class have to do something when they assign an objective.

7.7.2. Obstacles architecture

The obstacles of the environment also have a simple common code so the inheritance is a good option here again. Before continuing, it is important to show that all those who inherit from *Boogie.cs* have a *currentObjective* parameter, which is of the class Obstacle.cs.

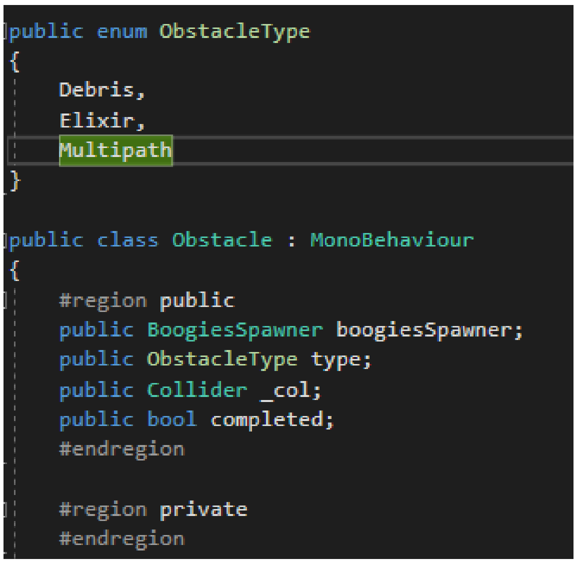


```
public abstract class Boogie : AttackTarget
{
    #region parameters
    public BoogieType type;
    [HideInInspector]public Vector3 initialPoint;
    [HideInInspector]public Vector3 randomPoint;
    public Obstacle currentObjective;
```

Figure 78: Boogie.cs parameters

So, as we can see in Figure 79, the Obstacle class contains common information for all the different kinds of obstacles such as:

- *Type*: this is the type of Obstacle.
- *_Col*: this is the collider of this obstacle. This is used to know the area covered by the obstacle.
- *Completed*: this is a Boolean that indicates if the obstacle is completed or it is not.



```
public enum ObstacleType
{
    Debris,
    Elixir,
    Multipath
}

public class Obstacle : MonoBehaviour
{
    #region public
    public BoogiesSpawner boogiesSpawner;
    public ObstacleType type;
    public Collider _col;
    public bool completed;
    #endregion

    #region private
    #endregion
```

Figure 79: Obstacle.cs

These are the classes that inherit from *Obstacle* class. In order to see them, at the end of the document is linked the repository where all the project is uploaded:

- *DebrisObstacle:* the class corresponding to the Debris Obstacle.
- *ElixirObstacle:* the class corresponding to the Elixir Obstacle.
- *MultipathObstacle:* the class corresponding to the Multipath Obstacle.

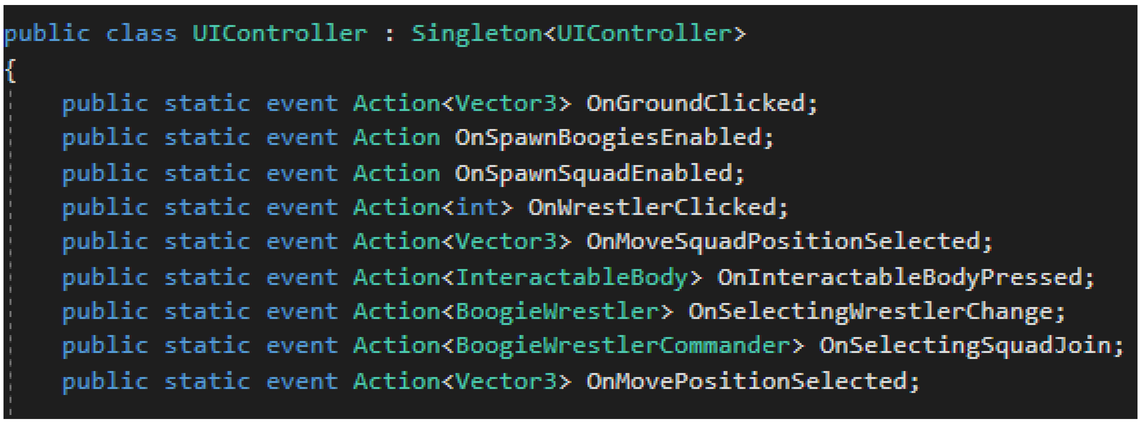
7.7.3. UI and Input architecture

The UI and the Input system plays a very important role in this project and they both are close related. So, this is why a good architecture should guarantee absence of bugs and possibility of scaling the project.

First of all, it is important to know that there is a UI controller class that handles all the logic input. This class uses the Singleton design pattern (TutorialsPoint, Singleton, s.f.) so from every class we can access the members of this UIController class with no references.

In order to able the player to select which actions the entities have to develop; some popups have been created as soon as the player clicks an entity. This click-checking is inside the UI controller. There are some actions that require to select a position (for example, the Move action), to select a body to cover (the Cover action), etc. In order to do it, the Observer design pattern (TutorialsPoint, Observer, s.f.) has been used. This is, the UI controller has declared some events and the operation is the one that follows:

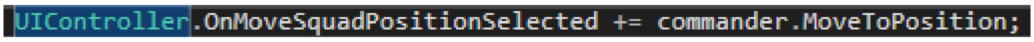
1. As soon as an action is selected, the troop that is going to perform this action is subscribed to the event. This is, to assign a function that will execute as soon as the event is invoked. We are going to use as an example the action Move. The user selects this option from the Popup and the wrestler that is going to move is subscribed to the event. That is, this wrestler assigns a method that do the following: *ChangeDestination(Where).*
2. As soon as the player selects with a click this position, the event is invoked and the method of the wrestler subscribed to this event is executed. This is, the *ChangeDestination(Where)* method will set the destination of the wrestler in the click position.



```
public class UIController : Singleton<UIController>
{
    public static event Action<Vector3> OnGroundClicked;
    public static event Action OnSpawnBoogiesEnabled;
    public static event Action OnSpawnSquadEnabled;
    public static event Action<int> OnWrestlerClicked;
    public static event Action<Vector3> OnMoveSquadPositionSelected;
    public static event Action<InteractableBody> OnInteractableBodyPressed;
    public static event Action<BoogieWrestler> OnSelectingWrestlerChange;
    public static event Action<BoogieWrestlerCommander> OnSelectingSquadJoin;
    public static event Action<Vector3> OnMovePositionSelected;
```

Figure 80: UIController events

As we can see in Figure 81, the OnMoveSquadPositionSelected is the event that will be invoked as soon as a destination position be clicked by the user. So, the wrestler that is going to move needs to subscribe some action to do when this event is invoked.

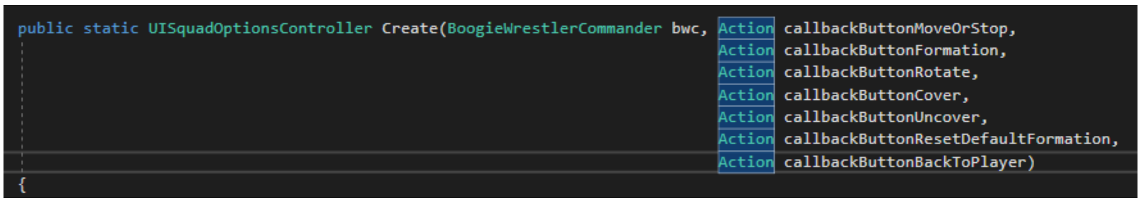


```
UIController.OnMoveSquadPositionSelected += commander.MoveToPosition;
```

Figure 81: Subscribe MoveToPosition

That function will execute the code shown in Figure 101 as soon as the event *OnMoveSquadPositionSelected* is invoked. For ending with the events, we just have to see in Figure 102 when this event is invoked.

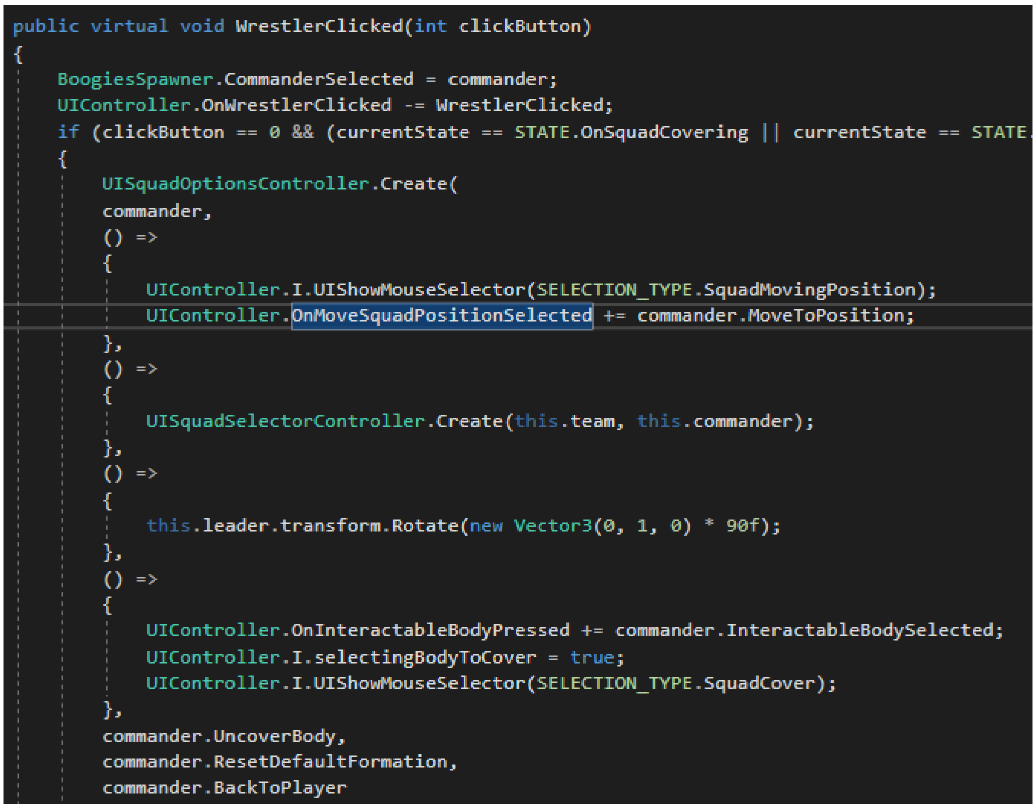
The Observer design pattern has been explained. Now, we are going to see how the popups are created. As seen before, depending on how we click inside a wrestler a different popup will appear. So, we have three different types of popups. The way of creating them is really efficient because they create and destroy themselves. Besides, as they are static classes, we do not need a reference of them.



```
public static UISquadOptionsController Create(BoogieWrestlerCommander bwc, Action callbackButtonMoveOrStop,
                                                                            Action callbackButtonFormation,
                                                                            Action callbackButtonRotate,
                                                                            Action callbackButtonCover,
                                                                            Action callbackButtonUncover,
                                                                            Action callbackButtonResetDefaultFormation,
                                                                            Action callbackButtonBackToPlayer)
{
```

Figure 82: Popup constructor

This way of creating popups is really efficient because we establish the actions to do in each button passing them via parameters. This is very comfortable and it is as simple as calling to the function as shown as in Figure 83.

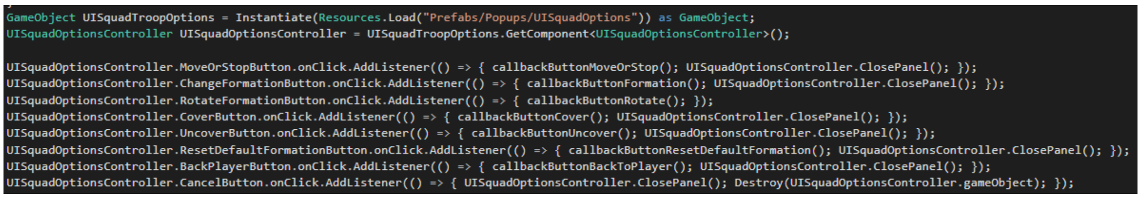


```
public virtual void WrestlerClicked(int clickButton)
{
    BoogiesSpawner.CommanderSelected = commander;
    UIController.OnWrestlerClicked -= WrestlerClicked;
    if (clickButton == 0 && (currentState == STATE.OnSquadCovering || currentState == STATE.
    {
        UISquadOptionsController.Create(
        commander,
        () =>
        {
            UIController.I.UIShowMouseSelector(SELECTION_TYPE.SquadMovingPosition);
            UIController.OnMoveSquadPositionSelected += commander.MoveToPosition;
        },
        () =>
        {
            UISquadSelectorController.Create(this.team, this.commander);
        },
        () =>
        {
            this.leader.transform.Rotate(new Vector3(0, 1, 0) * 90f);
        },
        () =>
        {
            UIController.OnInteractableBodyPressed += commander.InteractableBodySelected;
            UIController.I.selectingBodyToCover = true;
            UIController.I.UIShowMouseSelector(SELECTION_TYPE.SquadCover);
        },
        commander.UncoverBody,
        commander.ResetDefaultFormation,
        commander.BackToPlayer
```

Figure 83: Constructing popup

The method *WrestlerClicked* is executed when the event of clicking a wrestler is invoked. The first thing it does is to create a popup showing all the options available for that wrestler and for that type of click. Create that popup is as simple as passing via parameter all the actions that have to be performed depending on the action selected. The buttons in Unity have an internal event called *OnClick* (Unity Technologies, 2019). So, for each button we assign the *OnClick* action passed via parameter in the popup constructor.

In Figure 84 it can be seen how these callbacks passed by parameter are assigned as the actions to execute when the corresponding button is clicked using the *AddListener* function. This code is the continuation of the code shown in Figure 82.



```
GameObject UISquadTroopOptions = Instantiate(Resources.Load("Prefabs/Popups/UISquadOptions")) as GameObject;
UISquadOptionsController UISquadOptionsController = UISquadTroopOptions.GetComponent<UISquadOptionsController>();

UISquadOptionsController.MoveOrStopButton.onClick.AddListener(() => { callbackButtonMoveOrStop(); UISquadOptionsController.ClosePanel(); });
UISquadOptionsController.ChangeFormationButton.onClick.AddListener(() => { callbackButtonFormation(); UISquadOptionsController.ClosePanel(); });
UISquadOptionsController.RotateFormationButton.onClick.AddListener(() => { callbackButtonRotate(); });
UISquadOptionsController.CoverButton.onClick.AddListener(() => { callbackButtonCover(); UISquadOptionsController.ClosePanel(); });
UISquadOptionsController.UncoverButton.onClick.AddListener(() => { callbackButtonUncover(); UISquadOptionsController.ClosePanel(); });
UISquadOptionsController.ResetDefaultFormationButton.onClick.AddListener(() => { callbackButtonResetDefaultFormation(); UISquadOptionsController.ClosePanel(); });
UISquadOptionsController.BackPlayerButton.onClick.AddListener(() => { callbackButtonBackToPlayer(); UISquadOptionsController.ClosePanel(); });
UISquadOptionsController.CancelButton.onClick.AddListener(() => { UISquadOptionsController.ClosePanel(); Destroy(UISquadOptionsController.gameObject); });
```

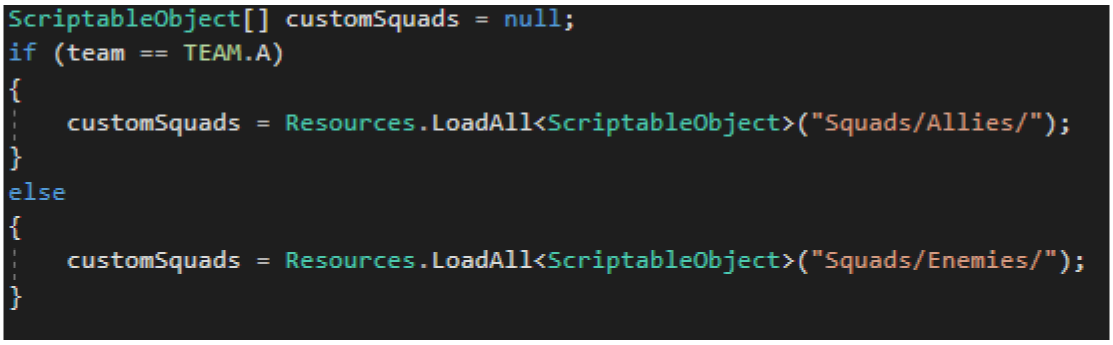
Figure 84: onClick.AddListener

This way of programming popups via constructors with the callbacks sent as parameters is also known as procedural popups.

For ending this point, it is going to be explained how to load Scriptable Objects information from the Assets. This is a pending explanation of the point 3.4.4 that required understanding the popups system. For understanding this load, we are going to see the following example:

When the player wants to spawn a squad of wrestlers, before doing it, he/she has to select which squad he wants to do from all the squads available. All these squads appear on a popup. What this popup does is to call this method:
UISquadSelectorController.Create(team:TEAM.A);

The constructor of this popup loads all the assets inside a squad and store them in a list. (Figure 85).



```
ScriptableObject[] customSquads = null;
if (team == TEAM.A)
{
    customSquads = Resources.LoadAll<ScriptableObject>("Squads/Allies/");
}
else
{
    customSquads = Resources.LoadAll<ScriptableObject>("Squads/Enemies/");
}
```

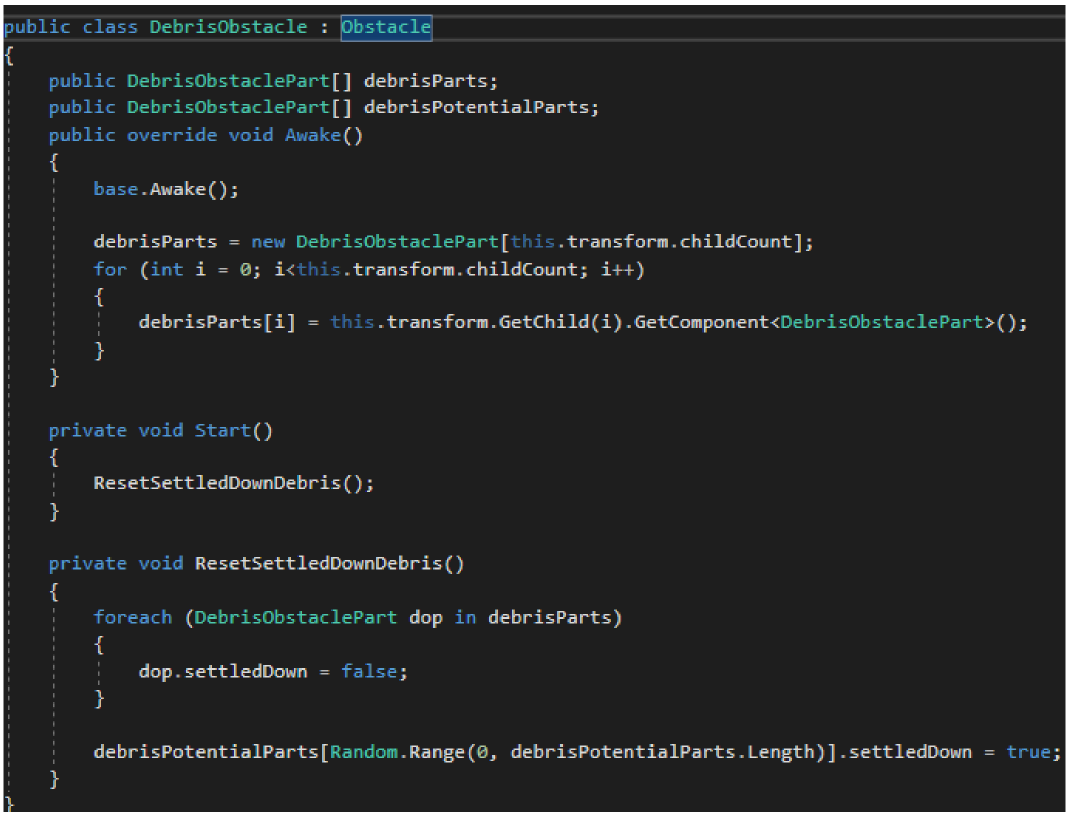
Figure 85: Loading assets

In order to be able to use the *Resources* library it is necessary to import the *UnityEditor* library.

Once the list is full, we will go through it and for each Scriptable Object in this list, we will load an asset doing a cast of the class that this Scriptable Object is instantiating. For doing this load it is used the *AssetDatabase.LoadAssetAtPath* (Unity Technologies, 2019). It is important to see Figure 103 in order to understand this.

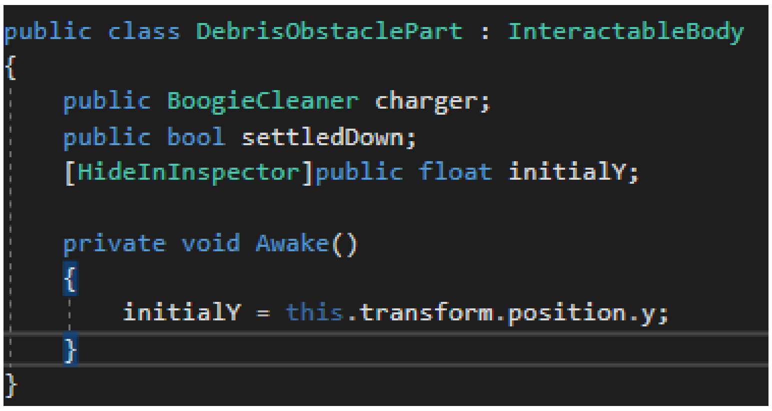
Once we have the instance of the class Squad, we can access the whole information of the file and spawn the squad.

7.8. Annex 8: Code captures



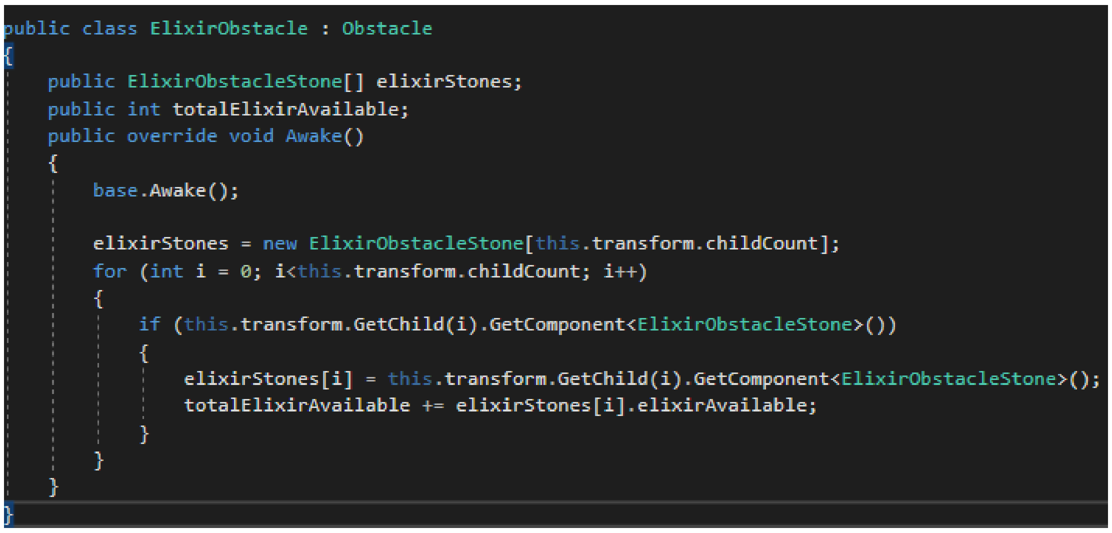
```
public class DebrisObstacle : Obstacle
{
    public DebrisObstaclePart[] debrisParts;
    public DebrisObstaclePart[] debrisPotentialParts;
    public override void Awake()
    {
        base.Awake();

        debrisParts = new DebrisObstaclePart[this.transform.childCount];
        for (int i = 0; i<this.transform.childCount; i++)
        {
            debrisParts[i] = this.transform.GetChild(i).GetComponent<DebrisObstaclePart>();
        }
    }

    private void Start()
    {
        ResetSettledDownDebris();
    }

    private void ResetSettledDownDebris()
    {
        foreach (DebrisObstaclePart dop in debrisParts)
        {
            dop.settledDown = false;
        }

        debrisPotentialParts[Random.Range(0, debrisPotentialParts.Length)].settledDown = true;
    }
}
```

Figure 86: DebrisObstacle.cs



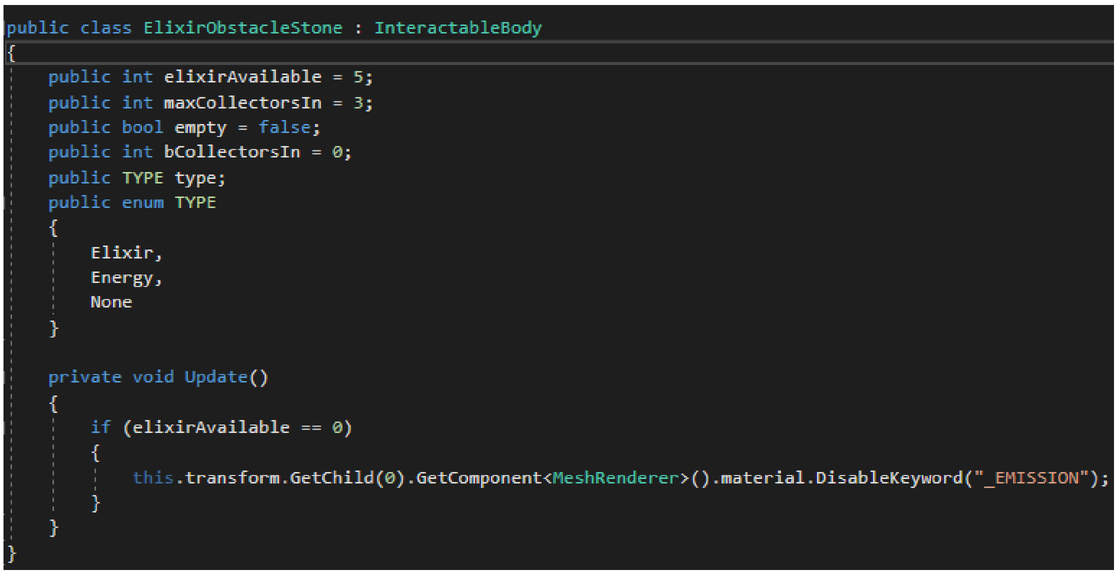
```
public class DebrisObstaclePart : InteractableBody
{
    public BoogieCleaner charger;
    public bool settledDown;
    [HideInInspector]public float initialY;

    private void Awake()
    {
        initialY = this.transform.position.y;
    }
}
```

Figure 87: DebrisObstaclePart.cs



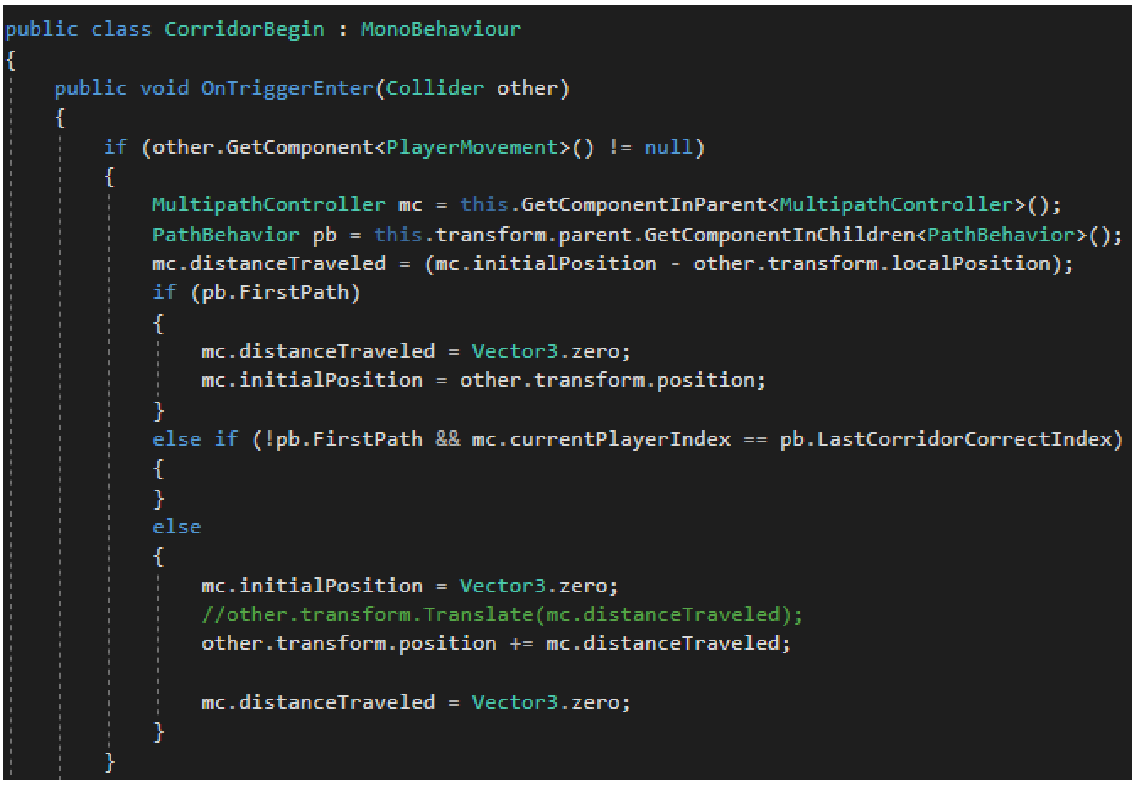
```
public class ElixirObstacle : Obstacle
{
    public ElixirObstacleStone[] elixirStones;
    public int totalElixirAvailable;
    public override void Awake()
    {
        base.Awake();

        elixirStones = new ElixirObstacleStone[this.transform.childCount];
        for (int i = 0; i<this.transform.childCount; i++)
        {
            if (this.transform.GetChild(i).GetComponent<ElixirObstacleStone>())
            {
                elixirStones[i] = this.transform.GetChild(i).GetComponent<ElixirObstacleStone>();
                totalElixirAvailable += elixirStones[i].elixirAvailable;
            }
        }
    }
}
```

Figure 88: ElixirObstacle.cs



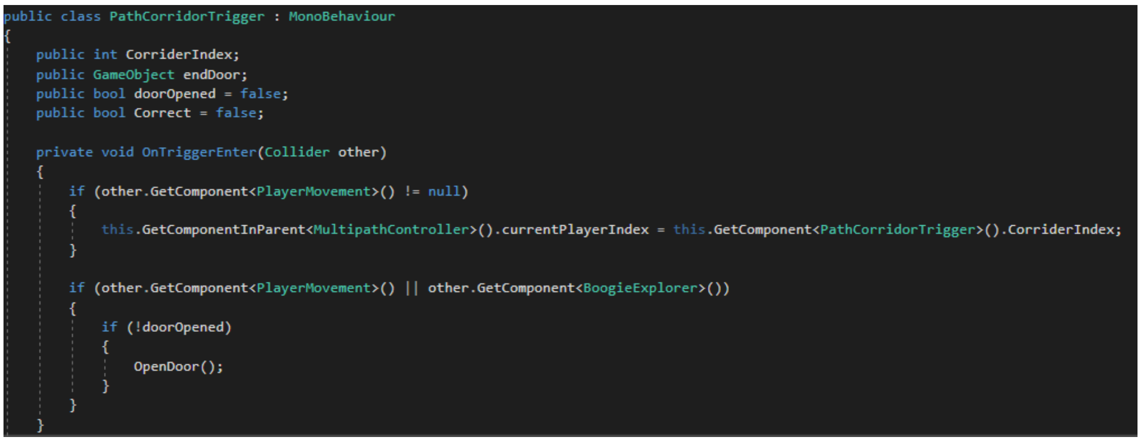
```
public class ElixirObstacleStone : InteractableBody
{
    public int elixirAvailable = 5;
    public int maxCollectorsIn = 3;
    public bool empty = false;
    public int bCollectorsIn = 0;
    public TYPE type;
    public enum TYPE
    {
        Elixir,
        Energy,
        None
    }

    private void Update()
    {
        if (elixirAvailable == 0)
        {
            this.transform.GetChild(0).GetComponent<MeshRenderer>().material.DisableKeyword("_EMISSION");
        }
    }
}
```

Figure 89: ElixirObstacleStone.cs



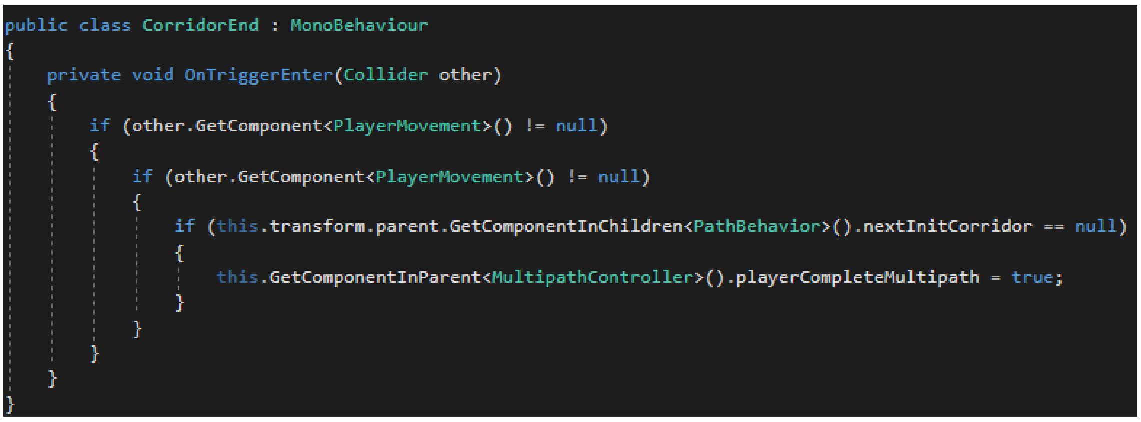
```
public class CorridorBegin : MonoBehaviour
{
    public void OnTriggerEnter(Collider other)
    {
        if (other.GetComponent<PlayerMovement>() != null)
        {
            MultipathController mc = this.GetComponentInParent<MultipathController>();
            PathBehavior pb = this.transform.parent.GetComponentInChildren<PathBehavior>();
            mc.distanceTraveled = (mc.initialPosition - other.transform.localPosition);
            if (pb.FirstPath)
            {
                mc.distanceTraveled = Vector3.zero;
                mc.initialPosition = other.transform.position;
            }
            else if (!pb.FirstPath && mc.currentPlayerIndex == pb.LastCorridorCorrectIndex)
            {
            }
            else
            {
                mc.initialPosition = Vector3.zero;
                //other.transform.Translate(mc.distanceTraveled);
                other.transform.position += mc.distanceTraveled;

                mc.distanceTraveled = Vector3.zero;
            }
        }
```

Figure 90: CorridorBegin.cs



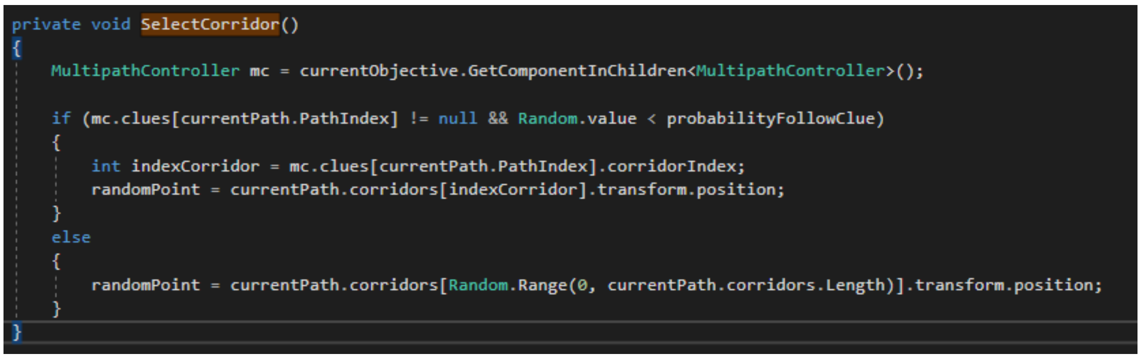
```
public class PathCorridorTrigger : MonoBehaviour
{
    public int CorriderIndex;
    public GameObject endDoor;
    public bool doorOpened = false;
    public bool Correct = false;

    private void OnTriggerEnter(Collider other)
    {
        if (other.GetComponent<PlayerMovement>() != null)
        {
            this.GetComponentInParent<MultipathController>().currentPlayerIndex = this.GetComponent<PathCorridorTrigger>().CorriderIndex;
        }

        if (other.GetComponent<PlayerMovement>() || other.GetComponent<BoogieExplorer>())
        {
            if (!doorOpened)
            {
                OpenDoor();
            }
        }
    }
```

Figure 91: PathCorridorTrigger.cs (Corridor trigger)



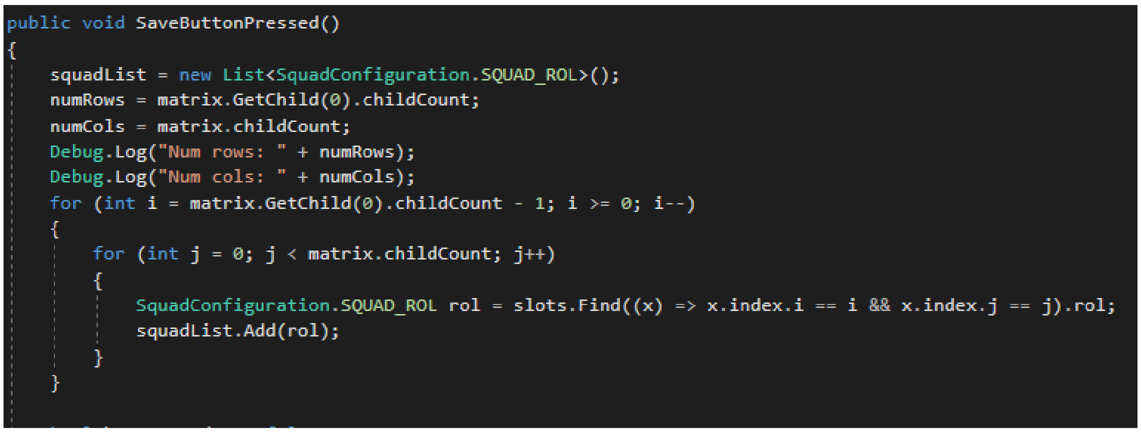
```
public class CorridorEnd : MonoBehaviour
{
    private void OnTriggerEnter(Collider other)
    {
        if (other.GetComponent<PlayerMovement>() != null)
        {
            if (other.GetComponent<PlayerMovement>() != null)
            {
                if (this.transform.parent.GetComponentInChildren<PathBehavior>().nextInitCorridor == null)
                {
                    this.GetComponentInParent<MultipathController>().playerCompleteMultipath = true;
                }
            }
        }
    }
}
```

Figure 92: CorridorEnd.cs



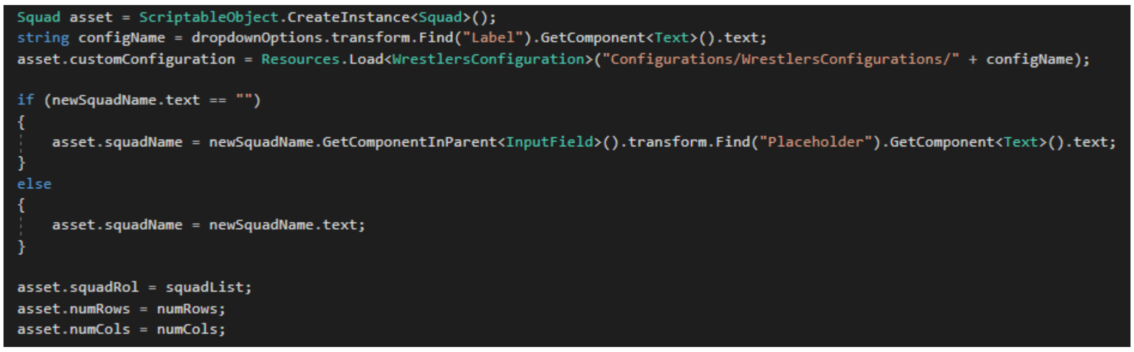
```
private void SelectCorridor()
{
    MultipathController mc = currentObjective.GetComponentInChildren<MultipathController>();

    if (mc.clues[currentPath.PathIndex] != null && Random.value < probabilityFollowClue)
    {
        int indexCorridor = mc.clues[currentPath.PathIndex].corridorIndex;
        randomPoint = currentPath.corridors[indexCorridor].transform.position;
    }
    else
    {
        randomPoint = currentPath.corridors[Random.Range(0, currentPath.corridors.Length)].transform.position;
    }
}
```

Figure 93: SelectCorridor()



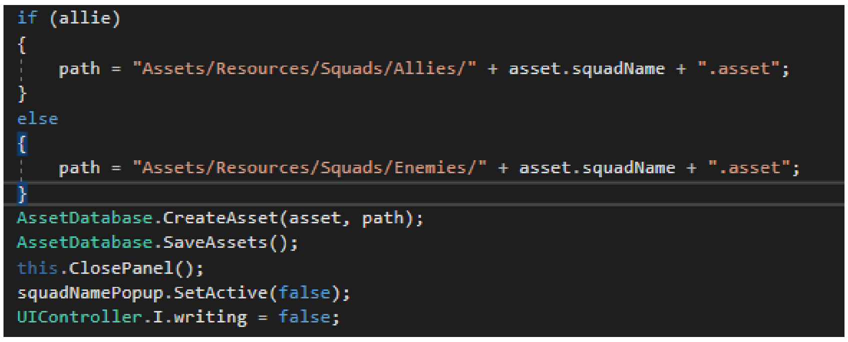
```
public void SaveButtonPressed()
{
    squadList = new List<SquadConfiguration.SQUAD_ROL>();
    numRows = matrix.GetChild(0).childCount;
    numCols = matrix.childCount;
    Debug.Log("Num rows: " + numRows);
    Debug.Log("Num cols: " + numCols);
    for (int i = matrix.GetChild(0).childCount - 1; i >= 0; i--)
    {
        for (int j = 0; j < matrix.childCount; j++)
        {
            SquadConfiguration.SQUAD_ROL rol = slots.Find((x) => x.index.i == i && x.index.j == j).rol;
            squadList.Add(rol);
        }
    }
```

Figure 94: Save button pressed



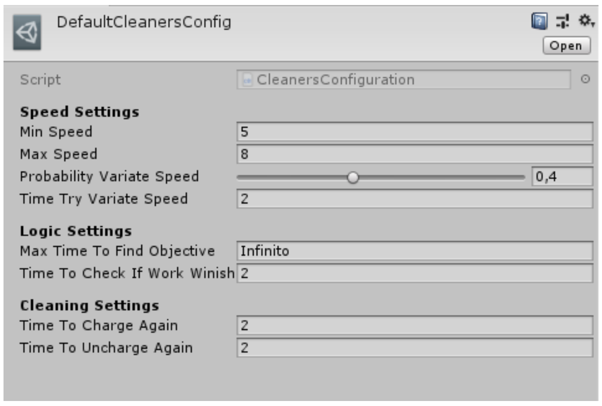
```
Squad asset = ScriptableObject.CreateInstance<Squad>();
string configName = dropdownOptions.transform.Find("Label").GetComponent<Text>().text;
asset.customConfiguration = Resources.Load<WrestlersConfiguration>("Configurations/WrestlersConfigurations/" + configName);

if (newSquadName.text == "")
{
    asset.squadName = newSquadName.GetComponentInParent<InputField>().transform.Find("Placeholder").GetComponent<Text>().text;
}
else
{
    asset.squadName = newSquadName.text;
}

asset.squadRol = squadList;
asset.numRows = numRows;
asset.numCols = numCols;
```

Figure 95: Creating instance of Squad.cs



```
if (allie)
{
    path = "Assets/Resources/Squads/Allies/" + asset.squadName + ".asset";
}
else
{
    path = "Assets/Resources/Squads/Enemies/" + asset.squadName + ".asset";
}
AssetDatabase.CreateAsset(asset, path);
AssetDatabase.SaveAssets();
this.ClosePanel();
squadNamePopup.SetActive(false);
UIController.I.writing = false;
```

Figure 96: Saving into assets

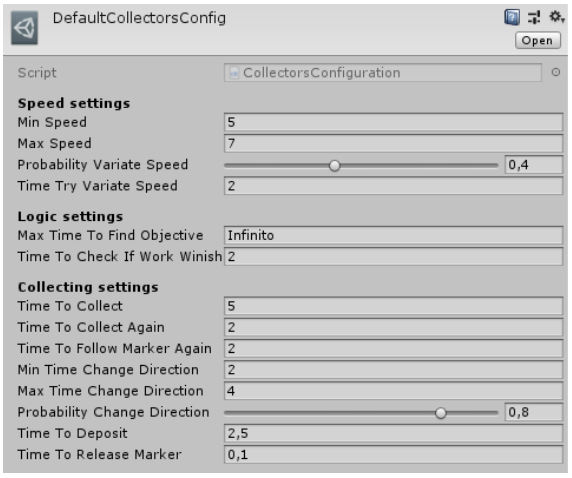


DefaultCleanersConfig

Open

Script	CleanersConfiguration
Speed Settings	
Min Speed	5
Max Speed	8
Probability Variate Speed	0,4
Time Try Variate Speed	2
Logic Settings	
Max Time To Find Objective	Infinito
Time To Check If Work Winish	2
Cleaning Settings	
Time To Charge Again	2
Time To Uncharge Again	2

Figure 97: Cleaners configuration

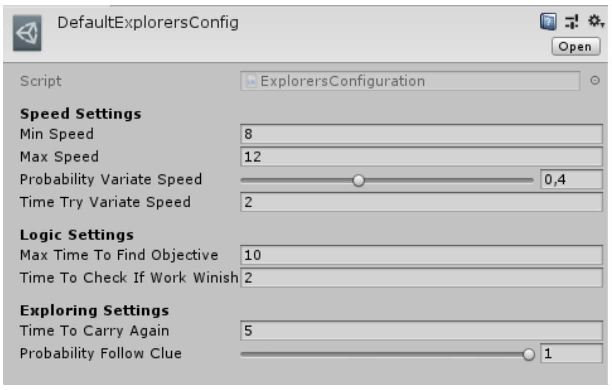


DefaultCollectorsConfig

Open

Script	CollectorsConfiguration
Speed settings	
Min Speed	5
Max Speed	7
Probability Variate Speed	0,4
Time Try Variate Speed	2
Logic settings	
Max Time To Find Objective	Infinito
Time To Check If Work Winish	2
Collecting settings	
Time To Collect	5
Time To Collect Again	2
Time To Follow Marker Again	2
Min Time Change Direction	2
Max Time Change Direction	4
Probability Change Direction	0,8
Time To Deposit	2,5
Time To Release Marker	0,1

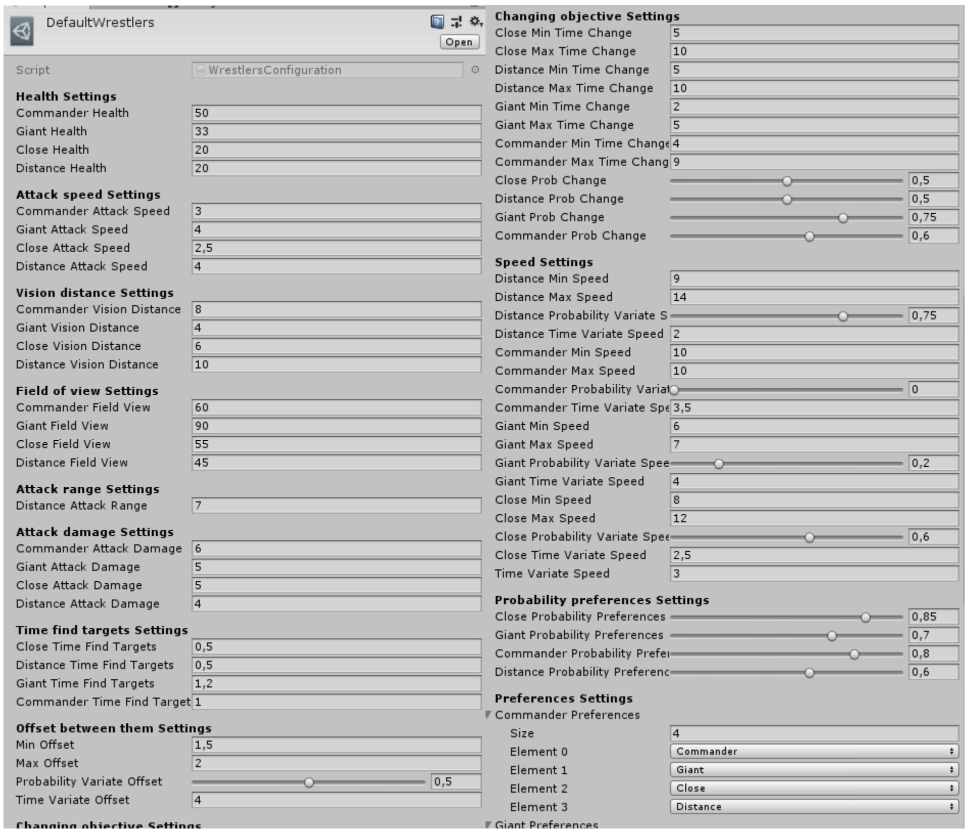
Figure 98: Collectors configuration



DefaultExplorersConfig

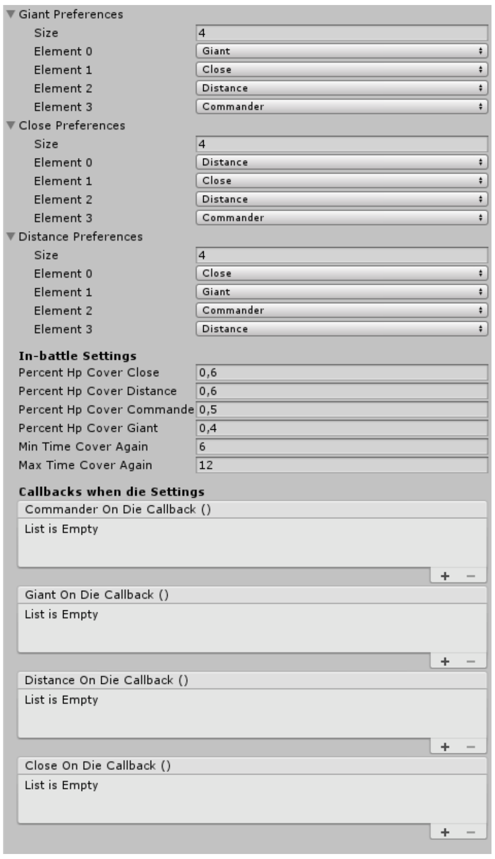
Script	ExplorersConfiguration
Speed Settings	
Min Speed	8
Max Speed	12
Probability Variate Speed	0,4
Time Try Variate Speed	2
Logic Settings	
Max Time To Find Objective	10
Time To Check If Work Winish	2
Exploring Settings	
Time To Carry Again	5
Probability Follow Clue	1

Figure 99: Explorers configuration



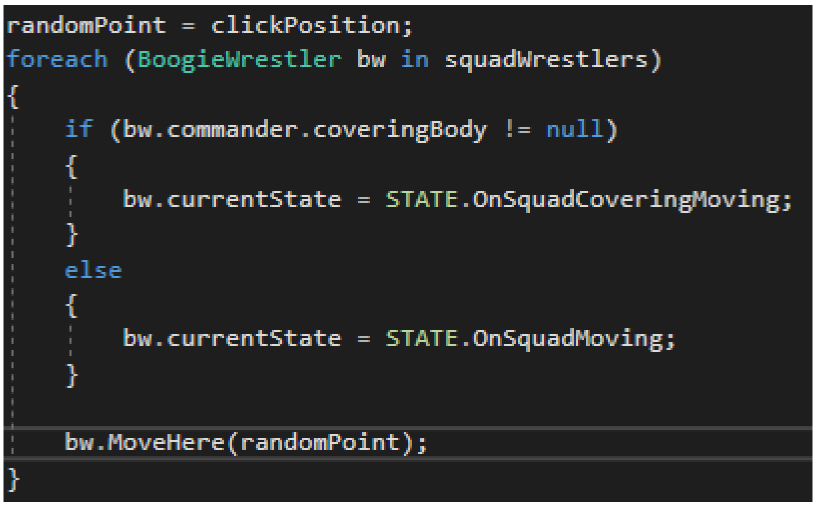
DefaultWrestlers

Script	WrestlersConfiguration
Health Settings	
Commander Health	50
Giant Health	33
Close Health	20
Distance Health	20
Attack speed Settings	
Commander Attack Speed	3
Giant Attack Speed	4
Close Attack Speed	2,5
Distance Attack Speed	4
Vision distance Settings	
Commander Vision Distance	8
Giant Vision Distance	4
Close Vision Distance	6
Distance Vision Distance	10
Field of view Settings	
Commander Field View	60
Giant Field View	90
Close Field View	55
Distance Field View	45
Attack range Settings	
Distance Attack Range	7
Attack damage Settings	
Commander Attack Damage	6
Giant Attack Damage	5
Close Attack Damage	5
Distance Attack Damage	4
Time find targets Settings	
Close Time Find Targets	0,5
Distance Time Find Targets	0,5
Giant Time Find Targets	1,2
Commander Time Find Target	1
Offset between them Settings	
Min Offset	1,5
Max Offset	2
Probability Variate Offset	0,5
Time Variate Offset	4
Changing objective Settings	
Close Min Time Change	5
Close Max Time Change	10
Distance Min Time Change	5
Distance Max Time Change	10
Giant Min Time Change	2
Giant Max Time Change	5
Commander Min Time Change	4
Commander Max Time Chang	9
Close Prob Change	0,5
Distance Prob Change	0,5
Giant Prob Change	0,75
Commander Prob Change	0,6
Speed Settings	
Distance Min Speed	9
Distance Max Speed	14
Distance Probability Variate S	0,75
Distance Time Variate Speed	2
Commander Min Speed	10
Commander Max Speed	10
Commander Probability Variat	0
Commander Time Variate Spe	3,5
Giant Min Speed	6
Giant Max Speed	7
Giant Probability Variate Spee	0,2
Giant Time Variate Speed	4
Close Min Speed	8
Close Max Speed	12
Close Probability Variate Spee	0,6
Close Time Variate Speed	2,5
Time Variate Speed	3
Probability preferences Settings	
Close Probability Preferences	0,85
Giant Probability Preferences	0,7
Commander Probability Prefer	0,8
Distance Probability Preferenc	0,6
Preferences Settings	
Commander Preferences	
Size	4
Element 0	Commander
Element 1	Giant
Element 2	Close
Element 3	Distance
Giant Preferences	



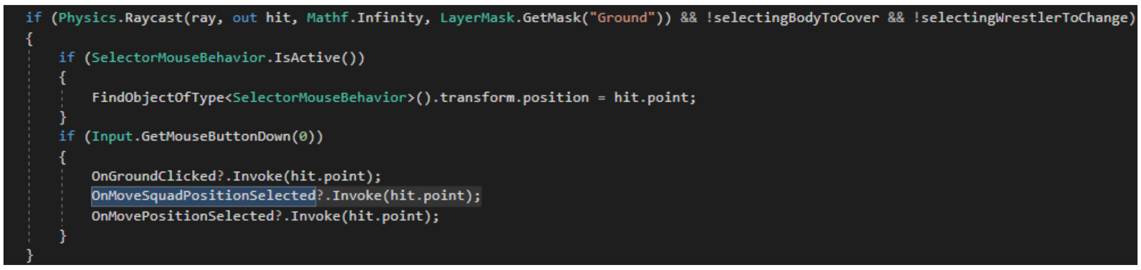
▼ Giant Preferences	
Size	4
Element 0	Giant
Element 1	Close
Element 2	Distance
Element 3	Commander
▼ Close Preferences	
Size	4
Element 0	Distance
Element 1	Close
Element 2	Distance
Element 3	Commander
▼ Distance Preferences	
Size	4
Element 0	Close
Element 1	Giant
Element 2	Commander
Element 3	Distance
In-battle Settings	
Percent Hp Cover Close	0,6
Percent Hp Cover Distance	0,6
Percent Hp Cover Commande	0,5
Percent Hp Cover Giant	0,4
Min Time Cover Again	6
Max Time Cover Again	12
Callbacks when die Settings	
Commander On Die Callback ()	List is Empty
Giant On Die Callback ()	List is Empty
Distance On Die Callback ()	List is Empty
Close On Die Callback ()	List is Empty

Figure 100: Wrestlers configuration



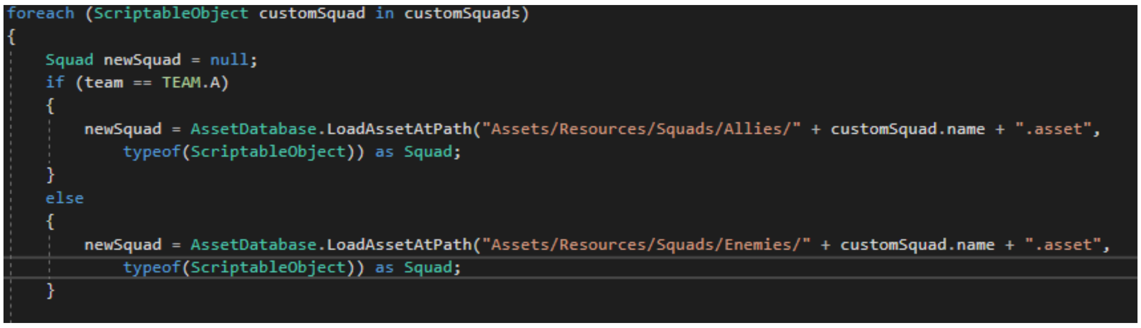
```
randomPoint = clickPosition;
foreach (BoogieWrestler bw in squadWrestlers)
{
    if (bw.commander.coveringBody != null)
    {
        bw.currentState = STATE.OnSquadCoveringMoving;
    }
    else
    {
        bw.currentState = STATE.OnSquadMoving;
    }

    bw.MoveHere(randomPoint);
}
```

Figure 101: MoveToPosition



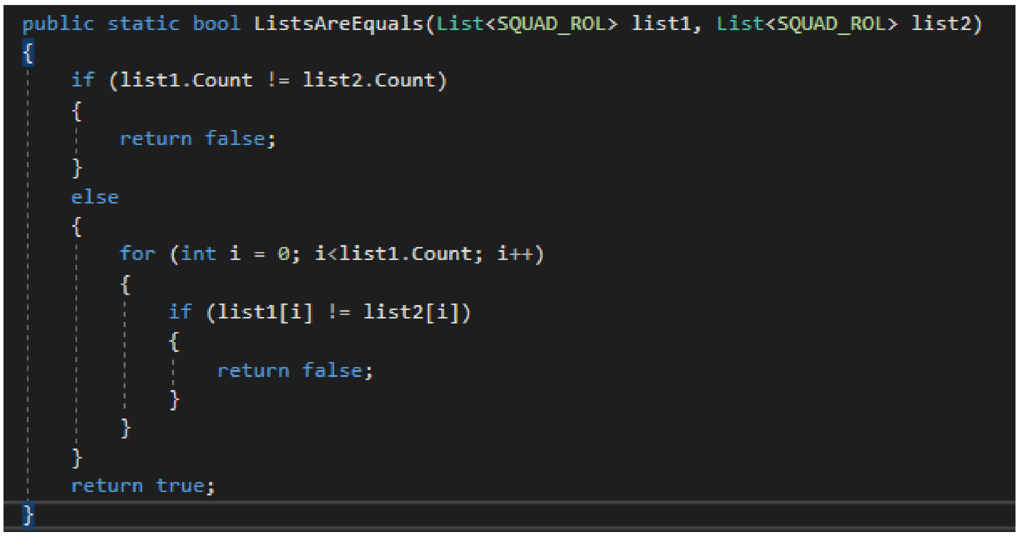
```
if (Physics.Raycast(ray, out hit, Mathf.Infinity, LayerMask.GetMask("Ground")) && !selectingBodyToCover && !selectingWrestlerToChange)
{
    if (SelectorMouseBehavior.IsActive())
    {
        FindObjectOfType<SelectorMouseBehavior>().transform.position = hit.point;
    }
    if (Input.GetMouseButtonDown(0))
    {
        OnGroundClicked?.Invoke(hit.point);
        OnMoveSquadPositionSelected?.Invoke(hit.point);
        OnMovePositionSelected?.Invoke(hit.point);
    }
}
```

Figure 102: Invoke



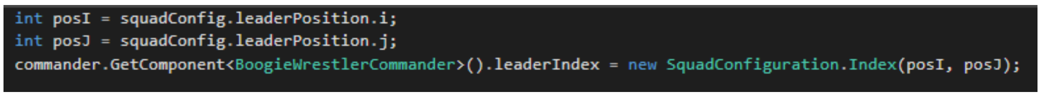
```
foreach (ScriptableObject customSquad in customSquads)
{
    Squad newSquad = null;
    if (team == TEAM.A)
    {
        newSquad = AssetDatabase.LoadAssetAtPath("Assets/Resources/Squads/Allies/" + customSquad.name + ".asset",
            typeof(ScriptableObject)) as Squad;
    }
    else
    {
        newSquad = AssetDatabase.LoadAssetAtPath("Assets/Resources/Squads/Enemies/" + customSquad.name + ".asset",
            typeof(ScriptableObject)) as Squad;
    }
```

Figure 103: AssetDatabase.LoadAssetAtPath



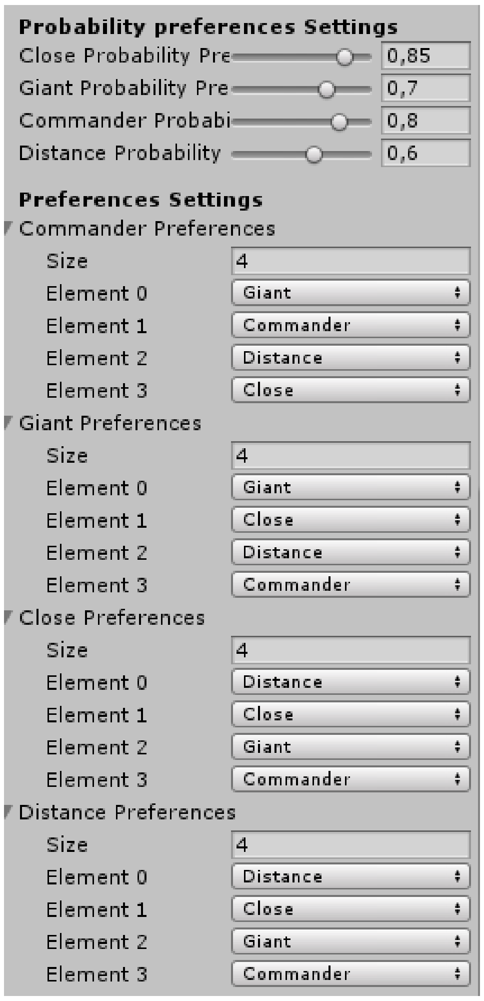
```
public static bool ListsAreEquals(List<SQUAD_ROL> list1, List<SQUAD_ROL> list2)
{
    if (list1.Count != list2.Count)
    {
        return false;
    }
    else
    {
        for (int i = 0; i<list1.Count; i++)
        {
            if (list1[i] != list2[i])
            {
                return false;
            }
        }
    }
    return true;
}
```

Figure 104: Custom comparer



```
int posI = squadConfig.leaderPosition.i;
int posJ = squadConfig.leaderPosition.j;
commander.GetComponent<BoogieWrestlerCommander>().leaderIndex = new SquadConfiguration.Index(posI, posJ);
```

Figure 105: Creating new Index using constructor



Probability preferences Settings

Close Probability Pre	0,85
Giant Probability Pre	0,7
Commander Probabi	0,8
Distance Probability	0,6

Preferences Settings

Commander Preferences

Size	4
Element 0	Giant
Element 1	Commander
Element 2	Distance
Element 3	Close

Giant Preferences

Size	4
Element 0	Giant
Element 1	Close
Element 2	Distance
Element 3	Commander

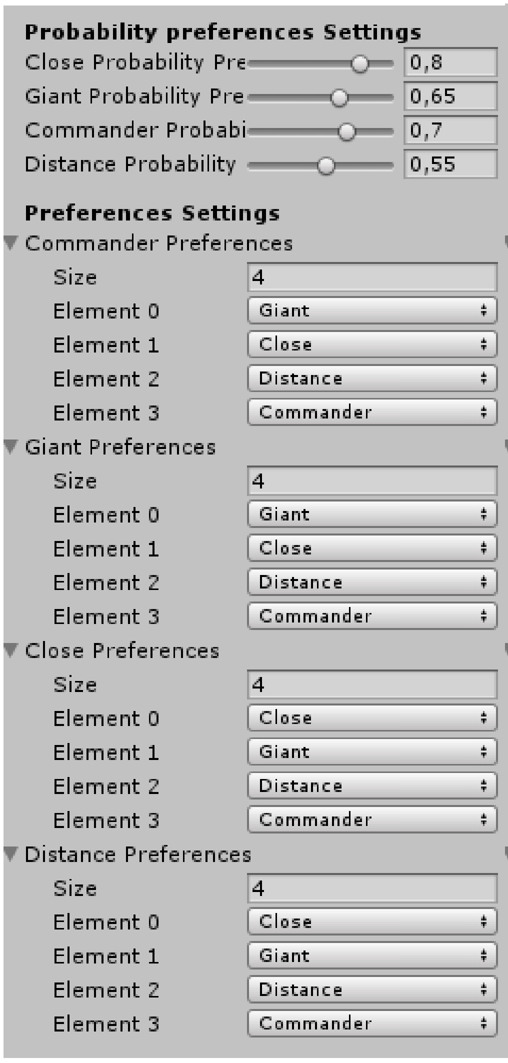
Close Preferences

Size	4
Element 0	Distance
Element 1	Close
Element 2	Giant
Element 3	Commander

Distance Preferences

Size	4
Element 0	Distance
Element 1	Close
Element 2	Giant
Element 3	Commander

Figure 106: Wrestler allies configuration



Probability preferences Settings

Close Probability Pre	0,8
Giant Probability Pre	0,65
Commander Probabi	0,7
Distance Probability	0,55

Preferences Settings

Commander Preferences

Size	4
Element 0	Giant
Element 1	Close
Element 2	Distance
Element 3	Commander

Giant Preferences

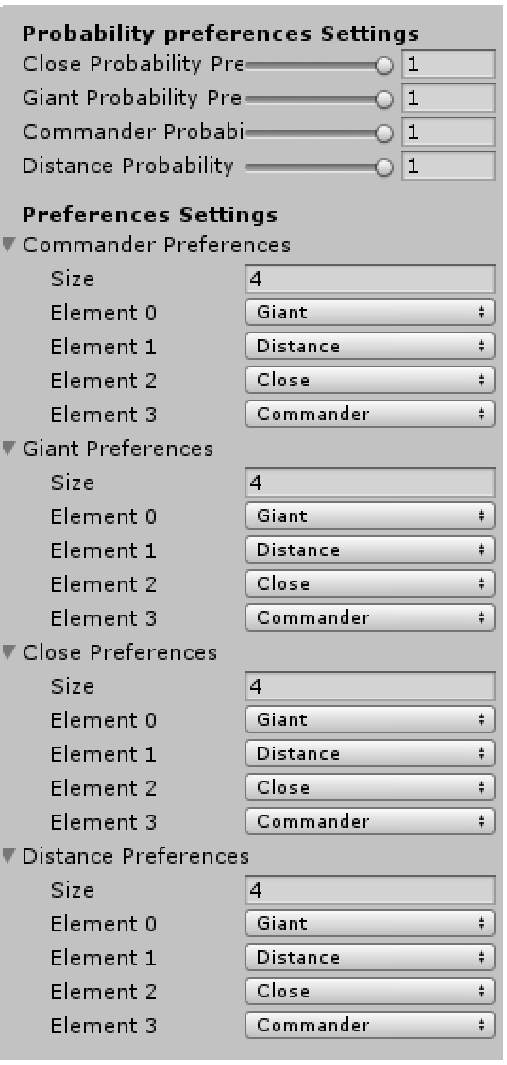
Size	4
Element 0	Giant
Element 1	Close
Element 2	Distance
Element 3	Commander

Close Preferences

Size	4
Element 0	Close
Element 1	Giant
Element 2	Distance
Element 3	Commander

Distance Preferences

Size	4
Element 0	Close
Element 1	Giant
Element 2	Distance
Element 3	Commander



Probability preferences Settings

Close Probability Pre	1
Giant Probability Pre	1
Commander Probabi	1
Distance Probability	1

Preferences Settings

Commander Preferences

Size	4
Element 0	Giant
Element 1	Distance
Element 2	Close
Element 3	Commander

Giant Preferences

Size	4
Element 0	Giant
Element 1	Distance
Element 2	Close
Element 3	Commander

Close Preferences

Size	4
Element 0	Giant
Element 1	Distance
Element 2	Close
Element 3	Commander

Distance Preferences

Size	4
Element 0	Giant
Element 1	Distance
Element 2	Close
Element 3	Commander

Figure 107: Wrestlers config. allies win (left)

Figure 108: Wrestlers config. enemies win (right)

List of figures

Figure 1: Debris Obstacle 15
Figure 2: Energy Obstacle 16
Figure 3: Multipath 17
Figure 4: Multipath Hierarchy 18
Figure 5: Set of corridors 18
Figure 6: Multipath demo [Video 1] 19
Figure 7: Cleaners action 1 20
Figure 8: Cleaners action 2 21
Figure 9: Cleaners action 3 21
Figure 10: Cleaners demo [Video 2] 22
Figure 11: Collectors action 2 24
Figure 12. Collectors action 3 25
Figure 13: Collectors action 4 25
Figure 14: Collectors action 5 25
Figure 15: Collectors demo [Video 3] 26
Figure 16: Clues 27
Figure 17: All clues placed 27
Figure 18: Explorers action 2 28
Figure 19: Explorers action 3 and 4 29
Figure 20: Explorers action 5 part 1 30
Figure 21: Explorers action 5 part 2 30
Figure 22: Explorers action 6 30
Figure 23: Explorers demo [Video 4] 31
Figure 24: Position calculation 32
Figure 25: Detection system 35
Figure 26: Targets detection 35
Figure 27: Distance detecting Commander 36
Figure 28: GoToEnemy() 37
Figure 29: AttackEnemy() 38
Figure 30: Emergency behavior 1 39
Figure 31: Emergency behavior 2 39
Figure 32: Emergency behavior 3 39
Figure 33: Squad.cs 40
Figure 34: Unity editor Create/Squad 40
Figure 35: Squad in-game creator panel 41
Figure 36: Squad in-game creator demo [Video 5] 42
Figure 37: All the configurations 42
Figure 38: AssignConfiguration() 45
Figure 39: Cleaners parameters 48
Figure 40: Cleaners variation demo [Video 6] 49
Figure 41: Collectors parameters 49
Figure 42: Collectors variation demo [Video 7] 50
Figure 43: Explorers parameters 50
Figure 44: Explorers variation demo [Video 8] 51
Figure 45: Select configuration 51

Figure 46: Wrestlers variation demo [Video 9] 52
Figure 47: Selection process 58
Figure 48: TryFindingInOurList() 59
Figure 49: CreateCustomSquad(...) 60
Figure 50: Navigation and masks 61
Figure 51: Check destination valid 62
Figure 52: Squad selects body to cover 63
Figure 53: Squad back to player 64
Figure 54: Call to change formation 64
Figure 55: Change formation 65
Figure 56: Assignments 66
Figure 57: Break formation action 66
Figure 58: AssignAsLeader() 67
Figure 59: JoinSquad() 68
Figure 60: Squad actions demo [Video 10] 68
Figure 61: Squad formation changes demo [Video 11] 69
Figure 62: ISaveable 69
Figure 63: SaverManager 70
Figure 64: PortalController 70
Figure 65: Jack 72
Figure 66: Ancient/Lodwig 72
Figure 67: Aidil/Roxie 73
Figure 68: Boogies collectors, cleaners, explorers, respectively 73
Figure 69: Boogie wrestlers squad 73
Figure 70: Enemy mutants 74
Figure 71: Game Scenario 74
Figure 72: Portal 75
Figure 73: Boogies menu selector 77
Figure 74: Keyboard controls 77
Figure 75: Class diagram (Zygomatic, s.f.) 79
Figure 76: AttackTarget.cs 80
Figure 77: Boogie.cs 81
Figure 78: Boogie.cs parameters 82
Figure 79: Obstacle.cs 82
Figure 80: UIController events 83
Figure 81: Subscribe MoveToPosition 84
Figure 82: Popup constructor 84
Figure 83: Constructing popup 84
Figure 84: onClick.AddListener 85
Figure 85: Loading assets 85
Figure 86: DebrisObstacle.cs 86
Figure 87: DebrisObstaclePart.cs 86
Figure 88: ElixirObstacle.cs 87
Figure 89: ElixirObstacleStone.cs 87
Figure 90: CorridorBegin.cs 88
Figure 91: PathCorridorTrigger.cs (Corridor trigger) 88
Figure 92: CorridorEnd.cs 89

Figure 93: SelectCorridor() 89
Figure 94: Save button pressed 89
Figure 95: Creating instance of Squad.cs 90
Figure 96: Saving into assets 90
Figure 97: Cleaners configuration 90
Figure 98: Collectors configuration 90
Figure 99: Explorers configuration 91
Figure 100: Wrestlers configuration 92
Figure 101: MoveToPosition 92
Figure 102: Invoke 92
Figure 103: AssetDatabase.LoadAssetAtPath 93
Figure 104: Custom comparer 93
Figure 105: Creating new Index using constructor 93
Figure 106: Wrestler allies configuration 94
Figure 107: Wrestlers config. allies win (left) 95
Figure 108: Wrestlers config. enemies win (right) 95

Videos

[Video 1] https://www.youtube.com/watch?v=i3ALl0J-fwU

[Video 2] https://www.youtube.com/watch?v=eXGZXzV8twc&t

[Video 3] https://www.youtube.com/watch?v=a_X-DLCVhww

[Video 4] https://www.youtube.com/watch?v=r7-XN3fG6D8&t

[Video 5] https://www.youtube.com/watch?v=1fCK7YiLAKQ

[Video 6] https://www.youtube.com/watch?v=-2RbRhF0zIc

[Video 7] https://www.youtube.com/watch?v=OpZIusVvVMg

[Video 8] https://www.youtube.com/watch?v=F2ePGhU3G0E

[Video 9] https://www.youtube.com/watch?v=o_fmlik6wHM

[Video 10] https://www.youtube.com/watch?v=kGAspqe-LZw

[Video 11] https://www.youtube.com/watch?v=ihSC7ghiUO0

[Video 12] https://www.youtube.com/watch?v=MXA5inP3DLQ