Recreation of a Realistic Ecosystem

by Jaume Sanz Sempere

Degree's Final Work

Degree in Video Game Design and Development

Universitat Jaume I

September 2020

Supervised by: Angel Pascual del Pobil Ferré

1. INTRODUCTION

Advances in technology have meant a notable progress in the industry of video games. With stories and graphics that only get more and more realistic, the audience has also become more demanding and, nowadays, quality and veracity are a must for video game creators. The need to produce realistic characters in realistic environments has now spread to all of the parts of a videogame. For that reason, we cannot leave aside any of the elements that appear on the screen.

This research work comes from the need to ensure that all of the elements of a video game are as close to reality as possible. In this case, we have decided to focus on living beings like wild animals and some plants, since we believe that nowadays they lack the realism and fidelity that other video game characters have, such as those who represent people. For this reason, we will recreate an ecosystem in which its living beings behave realistically, with the purpose of making progress in this field and applying it in future games.

In this chapter we will present the objectives that we sought to achieve in the development of this project, as well as the motivations that have led to the main idea of this work and the initial conditions from which it started.

1.1 Motivation

The main motivation of this project is to find a way to improve the artificial intelligence (AI) used in different games to control Non-Playable Characters (NPCs)[1] that represent living beings, like animals or plants, in their wild state. These animals will also have a great importance in the development of their habitat.

Games like the Pokémon Sword and Pokémon Shield are the main source of inspiration for this project, since these were some of the first video games that sought to integrate the “open world” genre by creating a map in which the player could move around freely. In these areas,

the players were able to encounter many different Pokémon depending on the place they were at, the climatic conditions of the area and the moment of the day. However, the creatures' behaviors were quite simple, since they could be seen wandering a concrete area, pursuing the player to attack them or flee from them.

This uncomplicated behavior became a source of frustration for the players, since Pokémon is a world-wide known video game saga that provided its consumers with a lot of information about each one of the species that appeared on the game, many times related to their behaviour and their interactions. However, this was hardly seen on the games because the creatures were too plain and not a lot of effort was put into their creation.

Many games of the same genre apply similar IAs to the fauna of their games, and some may include interactions with other NPCs, special objects or events among their behaviors. This usually helps to create the appearance that the world is actually alive and its creatures are realistic, which can be more than enough for some games, especially the oldest ones. However, there are other types of games which necessarily have to show realistic creatures with realistic conducts and nowadays this is not being carried out adequately, especially nowadays, when games need to be as close to reality as possible for the newest players.

Thus, the motivation of this project is to find the solution to this issue through the design of some IAs that will represent a living ecosystem in which the fauna, the flora and the environment interact with it. We have decided to carry out this project in order to contribute to the progress of video games and artificial intelligence, since we believe more work is needed in order to provide players with a lifelike experience when playing these type of games.

1.2 Objectives

As previously mentioned, the main goal of this Final Degree Project is to find a way to recreate ecosystems that act as close to real ecosystems as possible so that they can be applied to different video games in the future. In order to achieve this, the first point is to analyze which parts make up an ecosystem with the intention of establishing the objectives that this project must achieve to fulfill its goal.
In short, an ecosystem has two main components: a group of living beings related to each other, and the environment in which they carry out their activities[2].
Regarding the relationship between the living beings of an ecosystem, it is necessary to introduce the term "food chain". A food chain divides living beings intro producers (those who generate their own food) and consumers those who need to feed on other living beings). Within the consumers we find four categories: the herbivores (which feed on producers), the carnivores (which feed on other consumers), the omnivores (which feed on both producers and consumers) and finally the scavengers (these feed on the remains of dead consumers)

Regarding the environment, we can create a three-section division. In the first place, there is the day/night cycle, which establishes the hours of light and heat that affect an specific area, the climates, such as rain, snow or drought, and the geographical features that make it up such as rivers, lakes and mountains.

Thus, our first and main objective of this project is based on this definition of an ecosystem. For that reason, we will recreate one that has these characteristics, including the environment and the living beings that inhabit it. Moreover, our other objectives for this project are:

- To create an uneven stage on which living beings can exist
- To implement a day/night cycle that cyclically rotates between these two phases
- To implement a climate system that changes the climate to which the scenario is subjected
- To create a series of producers that generate food

- To create a series of NPCs that can interact with others and with the environment in which they are located by making use of an artificial intelligence that makes the appropriate decisions

1.3 Environment and initial state

At the beginning of this project, I was expecting to be able to dedicate my full time and effort, since I had plenty of hours free to do so. I started planning everything quite early, in order to avoid having to do everything last-minute, since that would be very perjudicial, not only to my project, but also to myself, since I suffer from chronic migraine headaches and anxiety. I was hoping a healthy and relaxed work environment would help my situation and my willingness, since I was very excited to embark in a project as ambitious as this.

At first, I encountered a small problem when planning this project: the fact that Artificial Intelligence was not a field we had studied in depth throughout my degree years, since I had only studied the bases of State Machines [3] and Behavior Trees [4]. Anyway, I was positive i had the necessary knowledge in order to be able to explore other more advanced options, such as mixing Hierarchical State Machines and *Behaviour Trees*. In addition, I was also counting on the support from my tutor Ángel Pascual del Pobli Ferré, an expert in the field of Artificial Intelligence.

Besides, I had also worked with Unity during my years of study, so I was also familiar with other tools, like *NavMesh* from Unity that could help me with movement and pathfinding for my creatures, as well as Probuilder, which could be useful for the creation of scenarios.

2. PLANNING AND RESOURCES EVALUATION

We will dedicate this chapter to talk about the planning that was initially expected to be followed for the development of this project. Furthermore, we will also discuss the amount of payment we would have received for this type of work.

2.1 Planning

It is estimated that a Final Degree Project should have a duration of around 290 without including the report. For this reason, we decided to estimate how long it would take us to carry out this project by calculating the amount of time each part should take in order to make an initial plan.

In order to do that, I had to take into account the fact that I had to do some research about Artificial Intelligence before starting with the actual project, since, as I previously mentioned, I hadn't studied it enough before. I was also going to have to research about the animals I was going to include in the project, in order to be accurate about their behaviours so that I could integrate them better into the ecosystem I was going to create.

In addition, since I was going to have to deal with much more difficult problems that what I am used to because of the complexity of the project, I was not sure about the amount of time I had to assign to some of the parts, since I had never done anything like that before. For this reason, I decided to add extra hours to those *Tasks* I believed would probably take more time, since I believed it would be safer to be prudent.

Tasks	Subtask	Estimation	Real cost	Total (Estimation - Real cost)
Research				20h -25h (+5h)
	Animals and Ecosystems	5h	5h	
	Artificial Intelligence	15h	20h (+5h)	
Design of the NPCs and their behaviours		10h		10h
Environment				30h - 65h (+35h)
	Implementation day/night cycle	6h	10h (+4h)	
	Implementation climate system	6h	10h (+4h)	
	Producers design	2h	5h (+3)	
	Producers implementation	5h	10h (+5h)	
	Research on environment design tools	3h	5h (+2h)	
	Research on pathfinding tools	3h	5h (+2h)	
	Design tests	5h	20h (+15h)	
Implementation of each NPCs' individual basic behaviours				90h - 285h (+195h)
	Creatures Design	5h	5h	
	Creatures Implementation	10h	20h (+10h)	
	Home implementation	10h	20h (+10h)	

	Creatures Movement	10h	30h (+20h)	
	Detection system implementation	5h	10h (+5h)	
	Basic common behaviour implementation	50h	200h (+150h)	
Implementation of each type of NPCs'collective behaviours			0h	50h - 0h
Interaction of behaviours based on interactions between different types of NPCs			0h	80h - 0h
Final Implementation				40h - 0h
	Final environment design	15h	0h	
	Environment building	20h	0h	
	NPCs Implementation	20h	0h	
	Environment correction	10h	0h	
Report elaboration				10h - 20h (+10h)
				300h-405h (+105h)

Figura 1. Table with time distribution

Luckily, in the beginning of the project I could combine some research and implementation *Tasks* in order to accelerate the development of the project, but as it progressed I had to complete some of the *Tasks* in order to continue with the next.

Had I worked with a full team, some tasks, like the ones related to the environment and the development of basic behaviours could have been made concurrently. This way, it would have been easier and faster to check any errors and make sure both teams interact correctly.

2.2 Resources evaluation

If I had been paid for this project, I would have received a total amount of 3850 Euros with a payment of 10 euros per hour.

3. SYSTEM ANALYSIS AND DESIGN

We will dedicate this chapter to the presentation of our project 's requirements, analysis, design and architecture, as well as a first look to the the design of the interface.

3.1 Requirement Analysis

In order to carry out our work we will make an initial analysis of the requirements. For that reason, we will divide requirements into two types: functional requirements, which determine what NPCs and Producers do according to the information they receive, and non-functional requirements, which determine how it must be done regarding execution, safety and/or speed.

Functional Requirements

Aimed at NPCs and Producers.

NPCs:

Input: Climate

Output: Behaviour

Each creature will be assigned one or two beneficial or perjudicial climates that will influence their behaviour while they take place

---

Input: Day phase

Output: Behaviour

Each creature will be assigned a time schedule to perform in, which will determine the phases of the day that are dedicated to activity or rest

Input: Internal data

Output: Behaviour

Each creature comes with a set of internal data that is updated with the passing of time. This data, like hunger, thirst or age, will determine their type of behaviour a given creature has.

Input: Creature nearby

Output: Behaviour

Each creature has a detection system that simulates real senses. If there's a creature within their senses' reach, the creature will act accordingly to the relationship between them.

Input: Producer nearby

Output: Behaviour

Each creature comes with a detection system that simulates real senses. If there's a producer within their senses' reach, the creature will go feed on them if the producer is on their normal diet and is hungry.

Input: Water nearby

Output: Behaviour

Each creature comes with a detection system that simulates real senses. If the creature detects water within their senses' reach, the creature will go drink if they are thirsty.

Input: Food o Water

Output: Internal data update

When a creature eats or drinks, its hunger or thirst bar will be updated respectively. .

Producers:

Input: Climate
Output: Food generation
Each producer is assigned a series of climates or combination of favourable, unfavourable or extra favourable climates that will determine how long the production of food should take or how much it should produce

Input: Day phase
Output: Food generation
In case the producer has run out of food, it will cost a certain amount of day phases to produce again

Input: Creature
Output: Food decrement
When a creature eats from a producer, it diminishes the amount of food available

Non-Functional Requirements

Non-functional requirements for this project are the following:

- The implementation must be efficient in order to be able to work with big groups of NPCs without altering the normal performance.

- Control structures must be reusable for new types of producers and creatures.

- Control structures must be scalable in order to be able to create more complex producers and creatures.

3.2 System Design

The functioning system designed for the elaboration of this work will be presented in the following charts with the purpose of showing the relationships between the different systems they seek to implement, according to the objectives we established before.

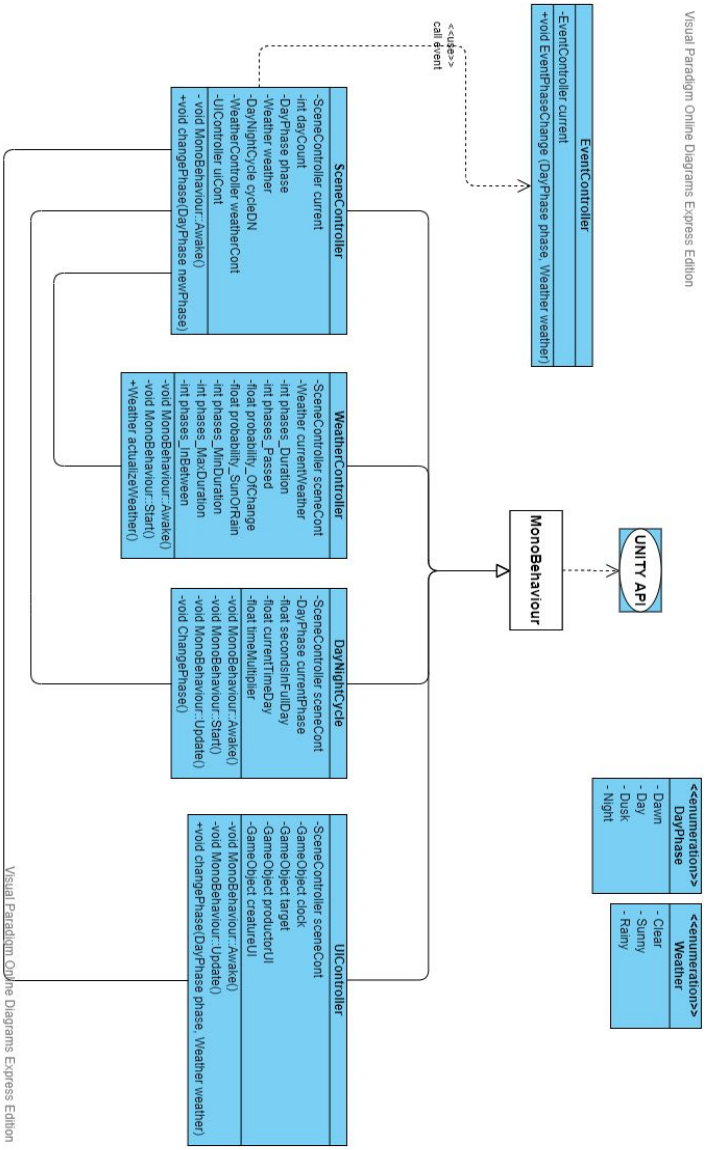

Figura 2. Class diagram of SceneController, WeatherController, DayNightCycle and UIController

This first class diagram [Figura 2] shows the functioning of one of the bases of the project: the day/night cycle and the climate system. A *SceneController* will be in charge of managing everything related to the environment's information and processes. In order to do so, it will be assigned a climate and a daily cycle controller. In addition, it will also be assigned with a user interface controller to manage part of the information that is shown on the screen. All

these controllers are classes derived from MonoBehaviour, the basic class of Unity that links it with its API.

The *DayNightCycle* is in charge of the day/night cycle, for which, given the length in seconds of how long we want a cycle to last, it is divided into 4 parts: dawn, day, afternoon and night. Whenever the phase changes, a message will be sent to the *SceneController.*

After this, the *SceneController* will send a message to the *ClimateController*, which will select one of the three climates at random within a few parameters before returning it.

Once these two data are obtained, the *SceneController* will activate the phase change event to which both the producers and the creatures are endorsed..

The second diagram (Figura 3) shows how the Producers work. As we can see, it is a simple scheme, since it has a Producer class and a *Controller*.

The Producer class descends from the *ScriptableObject* class, which is a special class that serves as a data container. This data can be edited from Unity's own editor, making it much easier to create different types of producers from a single database.

The *ProducerController* is in charge of the behavior of the producers. For it to work, it needs to have an object of the producer class in order to obtain the necessary data. One of the benefits of *ScriptableObjects* is that when an object contains another of this type it does not generate a copy but a reference, so any changes made to the Producer object will affect all references to it, even while at runtime. The *ProducerController* also endorses to the phase change event, which will activate the data whenever it is activated.

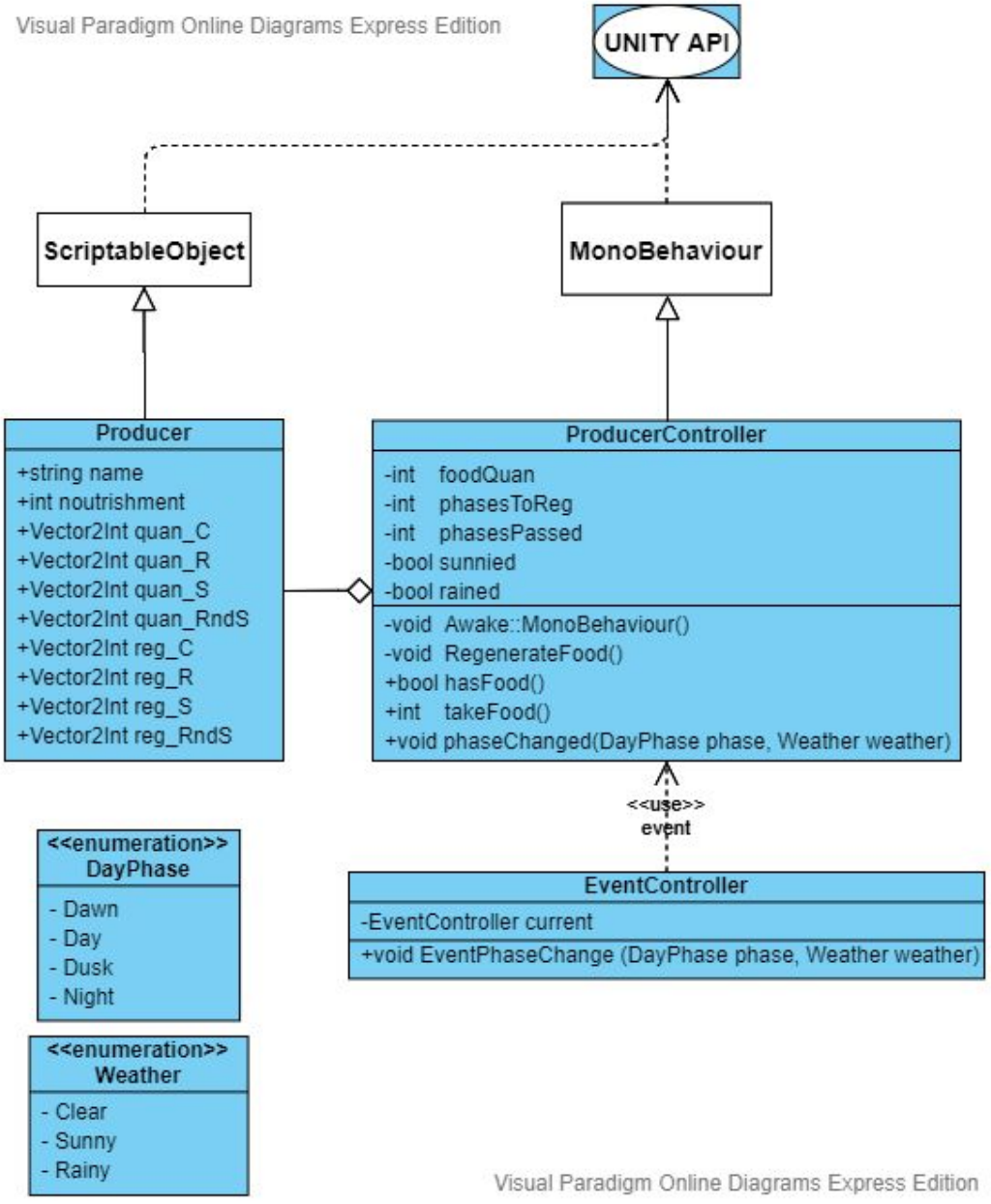

Figura 3. Class Diagram of ProducerController

Next, we can see the class diagram (Figura 4) that exemplifies the functioning of the creatures. The base structure is similar to the producers', with a Creature class that descends from *ScriptableObject* and contains all the information of the type of creature and a

CreatureController that manages the information and which descends from *MonoBehaviour*. In this case, the controller has other types of information, such as the resting place, the groups they are a part of or areas where they can find food or water. In addition, the creatures have other three more controllers, which are responsible for managing specific information and whose results will be passed to the main controller. These controllers are: the *SenseController*, whose job is to recreate the senses of sight and hearing by using a CollisionSphere, an object descended from the collider class that detects the objects it hits or that are within its volume, the *InteractionController*, which is in charge of checking when the creature's body interacts with another object thanks to a *BoxCollider* that works in the same way as a *SphereCollider* but with the volume of a cube, and, finally, a *BehaviourController*, which is in charge of managing the creature's artificial intelligence.

The *BehaviourController* will be in charge of receiving the different triggers detected by the other controllers of the creature to update the behavior according to them. It also needs a *BlackBoard*, a type of object that acts as a bridge to pass specific information for the action to be performed between the different controllers to the artificial intelligence.

As we have mentioned before, the artificial intelligence that controls the creatures is a *Hierarchical State Machine* that includes *Behaviour Trees*. The structure of this can be seen represented in Figura 4.

The *HierarchicalStateMachine* class is formed from *States* connected to each other through Transitions, which need a condition to be activated. Both states and transitions contain *Tasks*, the base class from which the behavior trees are formed. Finally, we have the *SubStateMachines*, a class that inherits from the *State* class, but has the same functionality as the *HierarchicalStateMachine*. In short, we have a *HierarchicalStateMachine* inside another *HierarchicalStateMachine.*

As we have previously mentioned, the *Tasks* are the base of the *Behaviour Trees*, the structure unit that form the *behaviour trees*. As their name states, *Behaviour Trees* have a “tree” structure, in which each *Task* represents an action that can give as result SUCCESS if the *Task* is carried out successfully, FAILURE when is not or RUNNING if it is still too early

to know the result. For that reason, some nodes are necessary in order to divide the tree in its different branches. These are the class *Composite*, that contains and works with several children *Tasks*. According to how they do it, we have two big groups: Sequences, which try to execute each of their children in order until one of them fails or ends, and Selectors, which execute their first child and only execute another one if the first one fails until it gets it right. Each one of these types has a pair of variants: one that is random and picks its children randomly (in this case it can even repeat one several times) and the non-deterministic one, which organises its children randomly but executes them normally.

Other types of node that only have one child are the Decorators, which add variation to the behaviors. There is also Limit, that determines the number of times its child can try to complete a *Task* (return the value RUNNING), the REPEATER, whose child executes itself once the *Task* is over, the Invert, that inverts the value of the SUCCESS and FAILURE and vice versa, the Succeeder, which returns SUCCESS when its child finishes the task and the UntilFail that makes its child repeat the execution until it fails.

From the *Task* class we will also create the "leaf" nodes of the tree, that will determine concrete actions such as moving to another place, calculating which is the source of food or any other action that we might need.

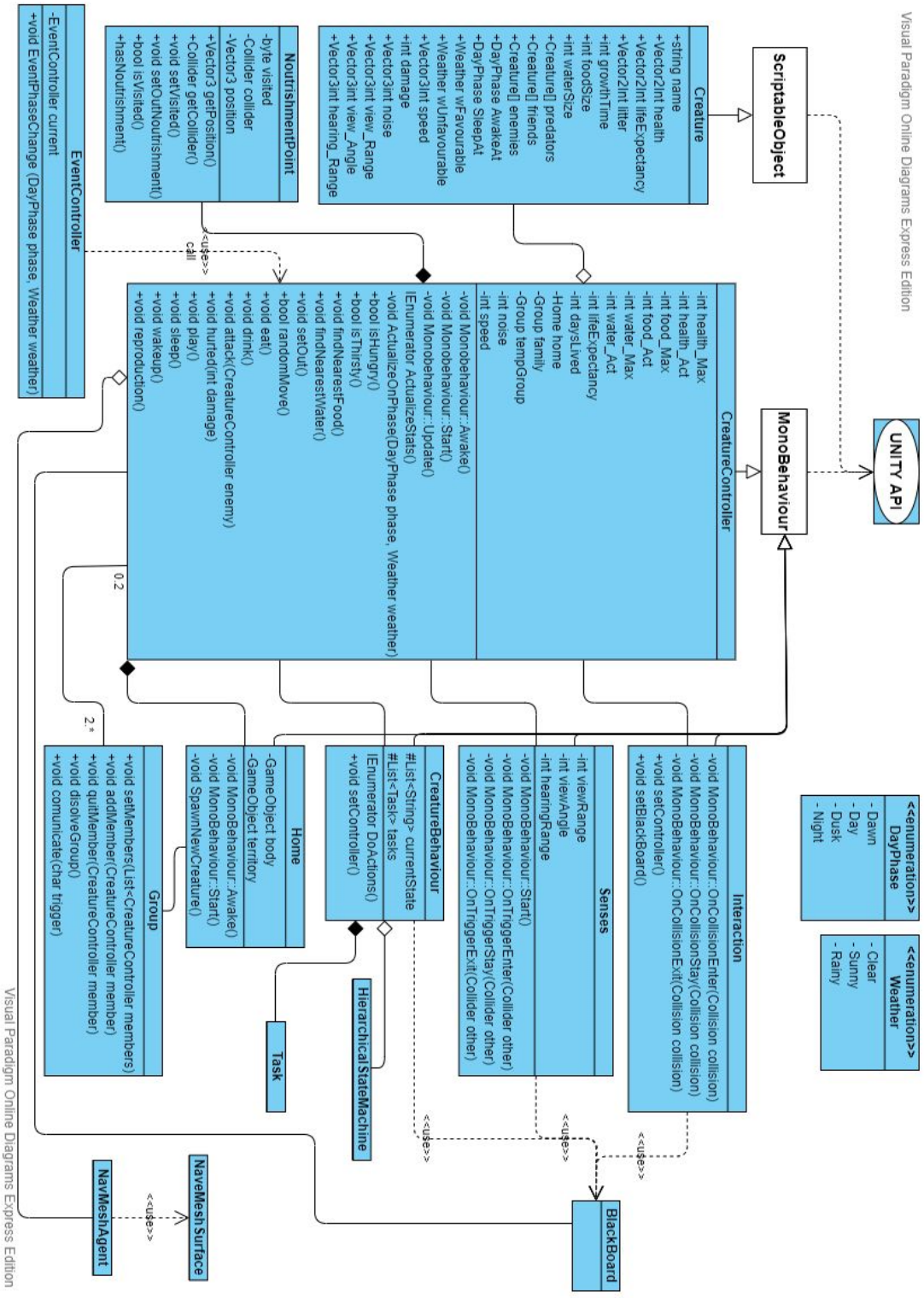

Figura 4. Class Diagram of CreatureController

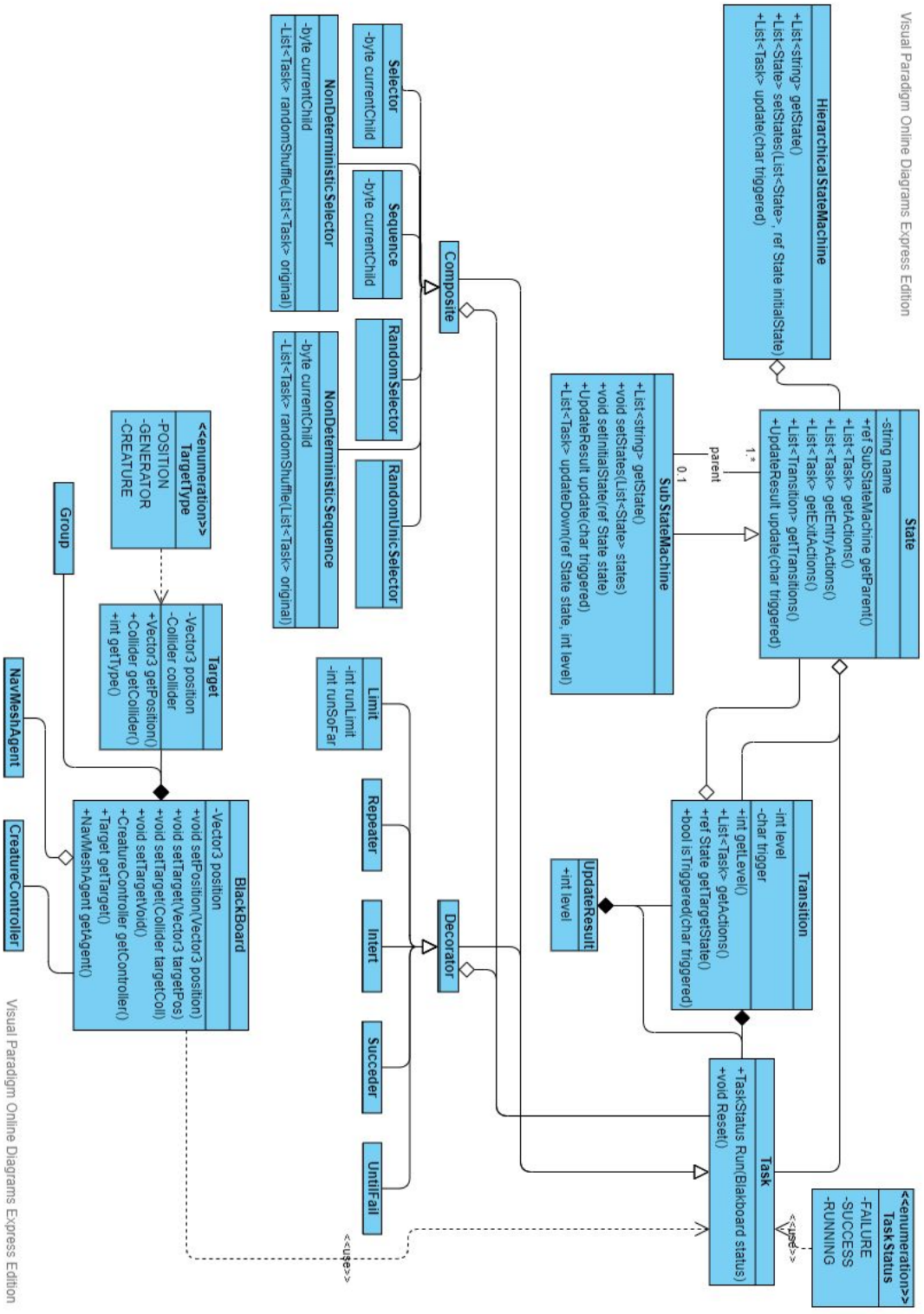

Figura 5. Class Diagram of Hierarchical State Machine

Finally, we have a diagram that shows the functioning of some derived classes (Figura 5). The classes derived from Creature add a list of the foods that they consume. In the case of Creature_Herbivorous, a list of Producers, in the case of *Creature_Carnivorous*, another creature, and in the case of *Creature_Omnivore*, both. *Creature_Scavenger* determines that the creature feeds on the remains left by another creature when it dies.

Next, we can also find the Egg class, derived from *ScriptableObject*, which contains the information about the egg and the Creature that will come out of it. *EggController* is the one that manages the behavior of the egg, like for example how many days it will take to hatch or the heat it needs in order to fulfill its *Task*.

Then, we have two group-derived classes that are mainly oriented to the familiar structures: the *Leader_Group*, which contains a Creature that guides the group, and the *MultiLeader_Group*, which has at least two Leaders that guide the group.

Finally, we have the Home class, the place where the creatures go to rest or to give birth. These contain one or more colliders that determine the territory that belongs to the creatures that inhabit it. This is also useful in order to know about the sources of food and water that are found within this territory or a creature that is found inside or outside of it. The Territorial class derives from this one, and it is designed for creatures that inhabit larger territories with multiple places to rest and that can add more territory to their domains by taking it away from other creatures.

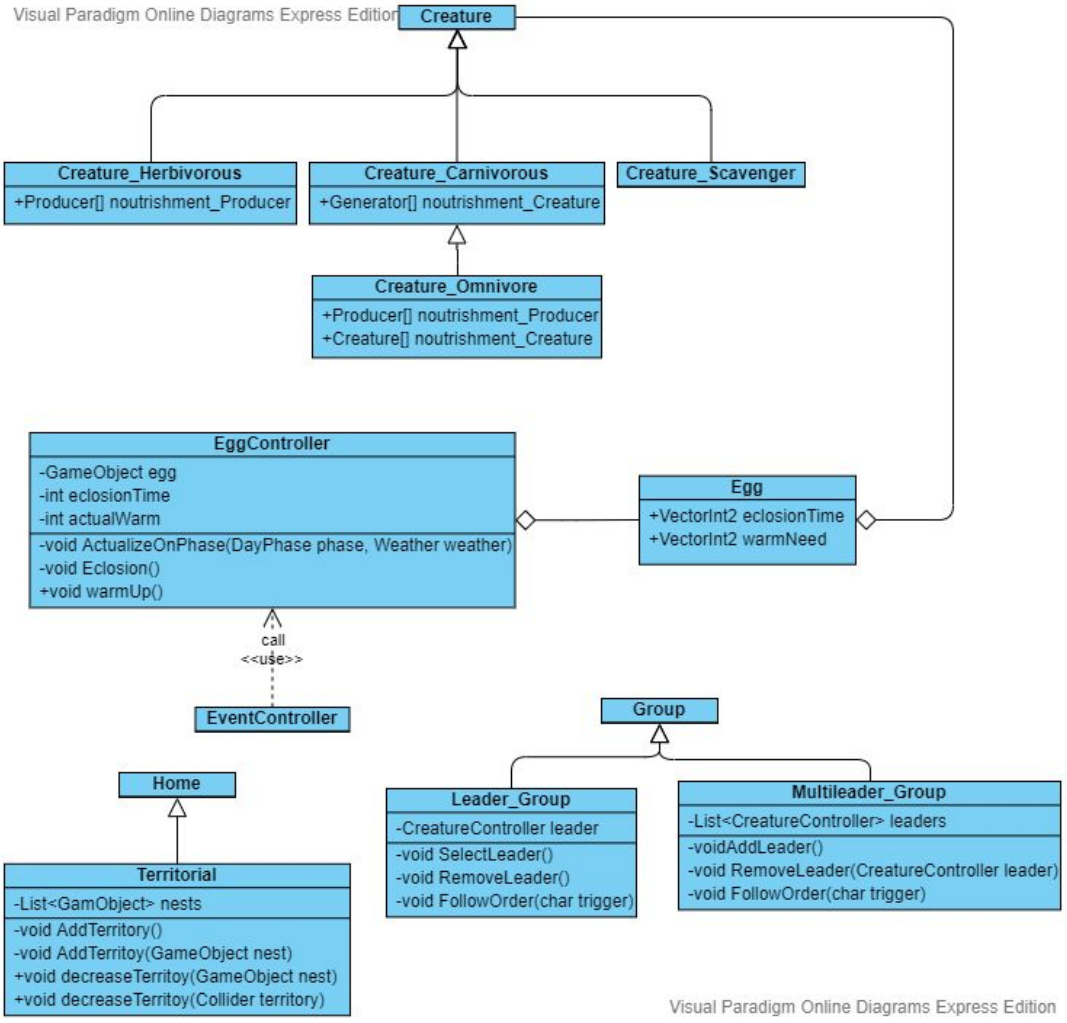

Figura 6. Class Diagram of some derived classes

Although each creature will have its own behaviors, the basis from which they all start will be practically the same. This means that the State Machine used by each creature will have more or less the same structure, which can be seen in the following chart. The *State Machine* (SM), called *Behaviour* is formed by the state *Sleep*, the one that starts the *SM*, and the *Sub Machine* (SbM) *Active*.

From the *Sleep* state it is only possible to pass to *Chill*, the initial State of the *Active* SbM, which represents the awakening of a creature. *Active* consists of two other SbMs, which are *Chill* and *Triggered*, and the states *Tired* and *Alert*. From *Chill* we can access both the *Alert* state, if it detects a creature nearby, and *Tired* state, when it is low on life or it is close to the time to rest, and from which you can also pass to *Alert*.

From *Alert* we can directly access to *Chill* if the creature does not feel any danger, to *State Interaction* in *Chill* if the creature nearby is friendly or a relative, or any of the states contained in *Triggered*: *Run* if it is dangerous or cannot be fought or *Attack* if it is weaker or a prey.

Inside *Triggered*, *Run* can go into *Attack* if there's no chance of running and *Attack* can go into *Run* if the creature believes it is going to lose. From *Triggered* you can only access *Alert* as a precaution in case there is more danger.

Finally, there is the *SbM Chill,* that contains the states *Satisfied*, *Hungry*, *Thirsty* and *Interaction*. *Satisfied* is the initial state of *Chill*, which represents the behavior of the creatures when all their needs are met, so it can be passed to both the *Hungry* state when it needs to eat or the *Thirsty* state when it needs to drink. Both *Hungry* and *Thirsty* can go into *Satisfied* state if their respective needs have been met or into each other if they need to meet the other. *Interact* has already been explained above, although it is necessary to explain its transitions as well. These are from *Interact* to *Chill* when the interaction ends or *Thirsty* and *Hungry* if one of these needs has to be covered during the interaction.

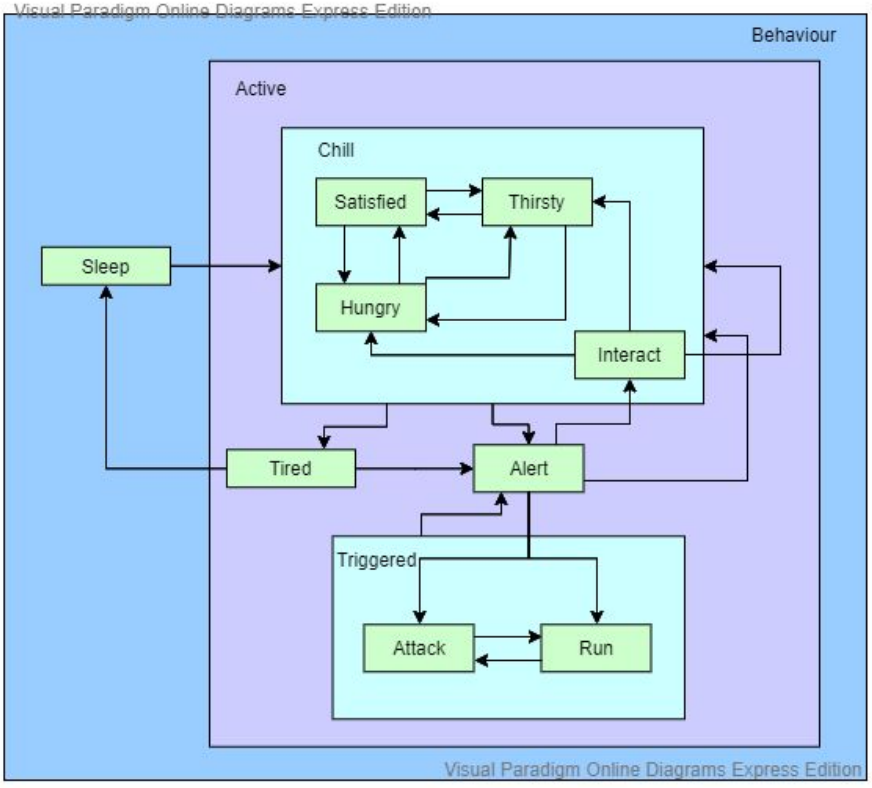

Figura 7. Base State Machine's diagram

3.3 System Architecture

I have not taken into account the computer's minimum requirements for this project, since the main goal is to lay the foundations for the running of an artificial intelligence.

3.4 Interface Design

Since the most interesting part about this project is to create an ecosystem in which all of the creatures can be seen performing realistic activities, the ideal interface would be something really simple that would give us the necessary information. For this reason, we are going to use a clean head-up display, initially with only the daily phase and the climate that can be seen in the top left corner.

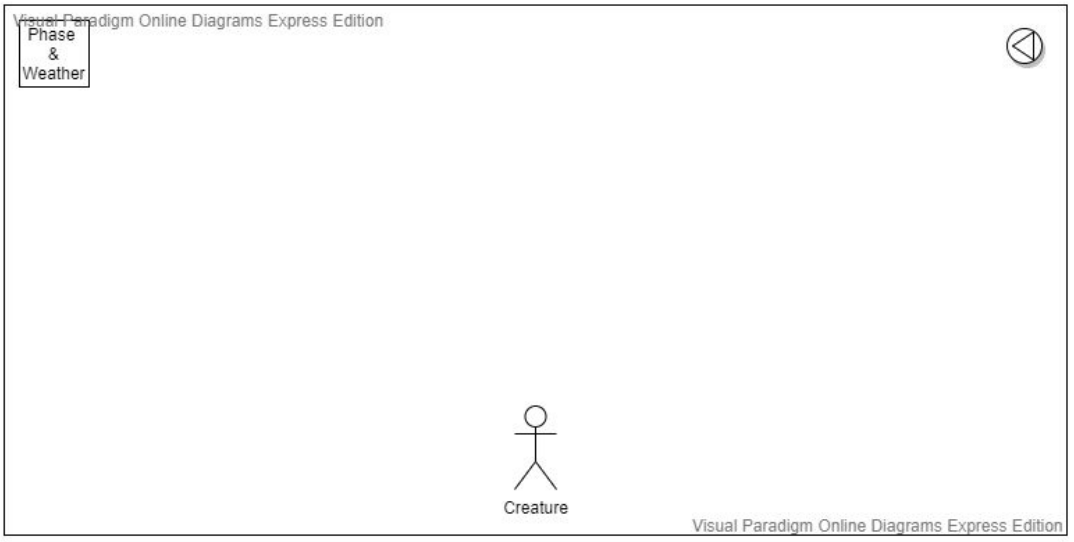

Figura 8. User Interface Scheme 1

The user will be able to move freely into the scenario with a mouse. If the user wishes to check the basic statistics of a creature that is seen on screen, it will only have to make a left click on it and it will display a small additional interface that will show us the species of the creature, the sex and the age and the health, hunger and thirst bars. In addition, the camera will stay fixed on the creature until we make a left click again or just click on the scenario.

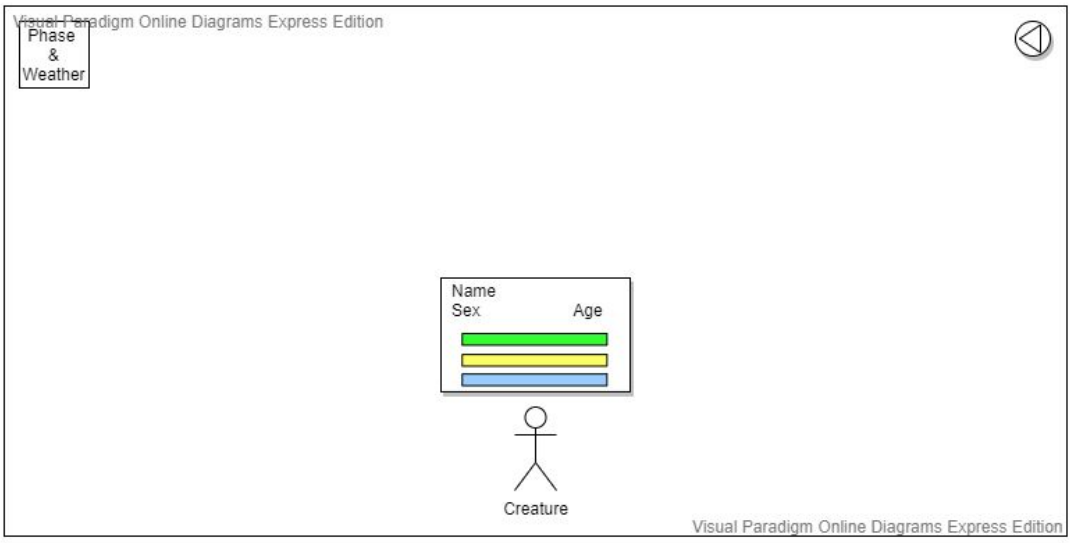

Figura 9. User Interface Scheme 2

We will place a small button in the top right corner that will open or close a new interface when clicked. This interface will have a list of all of the creatures in scene, classified by species. When the user clicks in one of the creatures, the camera will move to their position and it will stay fixed on the creature and display all of its information

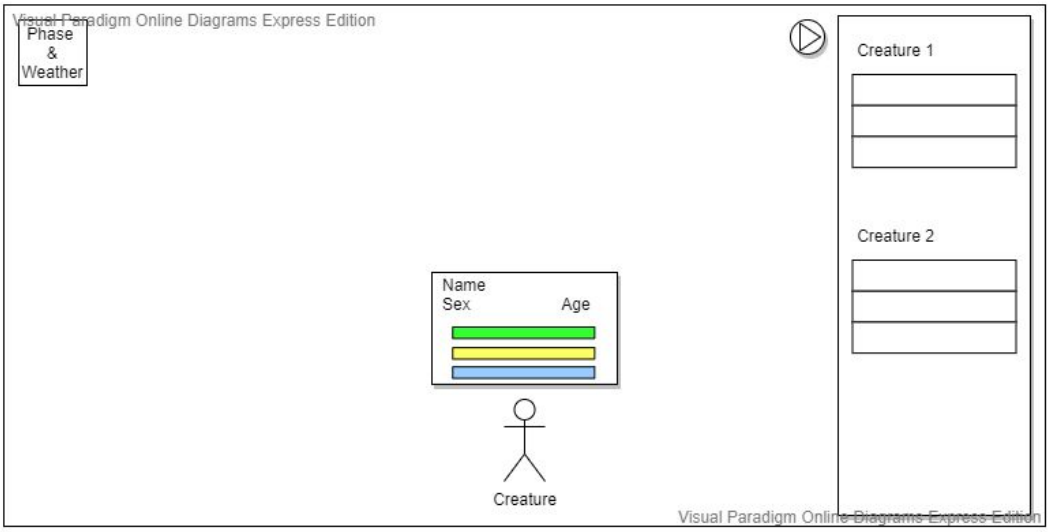

Figura 10. User Interface Scheme 3

4. WORK DEVELOPMENT AND RESULTS

4.1 Work Development

Before the start of the implementation, I looked for information related to how real ecosystems work, as well as state machines and *behaviour trees*. Once I had gathered enough information, I started to design the ecosystem I wanted to create.

My planned ecosystem was formed by three producers and five creatures as a way to include the most important links of the food chain. In addition, I also wanted to represent the three main forms of movement. For that reason, I did some research to observe real animals in order to create the creatures of the ecosystem, which are the following: a hare, representing the herbivores, a snake, representing the carnivores, a fox, representing the omnivores and a vulture, representing the scavengers. Besides, in order to include the movement in water (since the vulture represents the movement in air and the other creatures represent the movement in ground), as well as a contrast with a super predator (represented by a fox, since there are no predators in this environment), I decided to include semi-aquatic rhinoceros as a personal contribution, since these do not exist in nature.

Regarding the producers, I picked grass, a berry bush and seaweed so that the creatures would have a variety to select from.

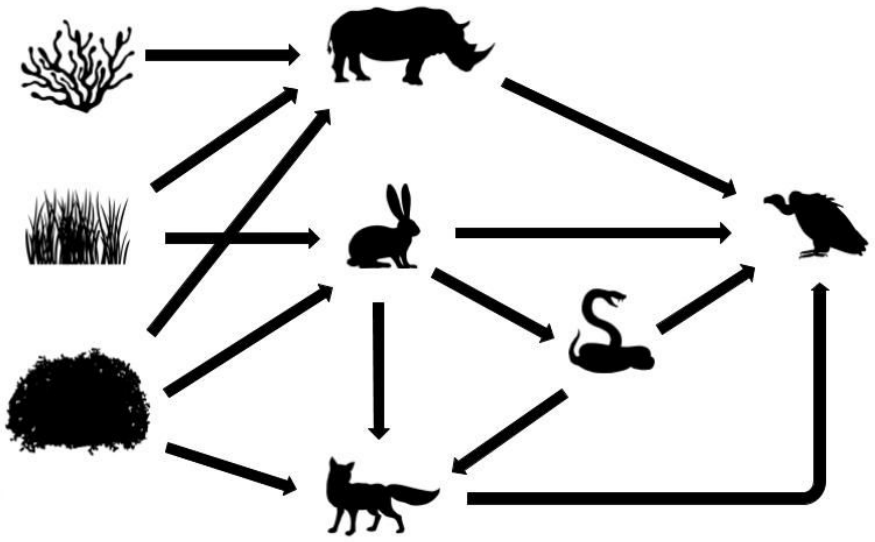

Figura 11. Trophic chain representation

In order for this ecosystem to be in balance and for no link that can break the whole chain to disappear, we will have to take different factors into account. These range from the climate itself and the distribution of the homes of each creature or group in the territory to the speed of reproduction of both creatures and producers and the metabolism of each type of creature. For example, in our case, we should have a greater number of Hares than Foxes and Vipers on a constant basis. On the side of the Vipers it is something simple, since the snakes are creatures of slow metabolism that feed continuously and also have the Fox as a predator. Regarding the Fox, it is somewhat more complex, since they do not have predators, but at the same time they have other sources of food such as the Vipers and berries, since they are omnivores.

Apart from this, as we have previously mentioned, we will have to take reproduction into account, allowing the Hares to reproduce faster and in greater quantities so that they are not easily surpassed in number with respect to their predators, while the latter should have less speed of reproduction or descendants. Another way we will regulate this is by limiting reproduction to only take place if the health of both individuals is above 75% and their hunger and thirst levels are above 33%. In addition, female mammals will not be able to

reproduce if their gestation period is longer than their life expectancy or while their offspring have not fully grown yet.

For example, if Hares proliferate more than they should, this will cause Vipers and Foxes to proliferate, while the amount of food available will decrease. This will cause the number of Hares to drop again over time, as there will be little food and it will be more difficult to reproduce since there will be more predators. This decrease in numbers will progressively lower the number of predators as there is not enough food for everyone until the overall balance is restored.

This movement in the numbers of Hares would also indirectly affect the rhinos, since they share a large part of the diet, and vice versa. The Vulture would be the least affected in all this, as their proliferation is related to the number of creatures that die, so a destabilization in any sense of the ecosystem would benefit them until this is corrected.

All these numerical factors regarding reproduction mentioned above can vary according to the needs we find in a given scenario to adjust the ecosystem.

As we had established before, the daily cycle would be divided into four parts: dawn, day, afternoon and night, and the climate into three parts: clear, sunny and rainy. The amount of time necessary to complete a full cycle will stay open until the debugging phase, where we will check the appropriate length for all functions to be carried out by each creature. We made this decision when making the initial tests, since we needed them to be as quick as possible in order to save time. Initially, we decided 40 minutes would be a suitable amount of time, but, as we said, we will finally establish it once we have made more progress.

However, regarding climate, we actually established from the beginning that the chance of a change in climate in every change of phase would be of a 15%, with a 50% of chances of sun or rain and with a length that would go from 1 phase to 3 cycles (12 phases). Besides, when there's an altered climate, there must be two full cycles of base climate before changing again.

Once we had made these decisions, the next step was to indicate how the climate or the phases might affect each living being, beginning with the producers because of their simplicity.

Taking the real life producers as a model, we can see climate can affect them in the quantity of production of food as well as in the amount of time they can take to produce again. Normally, the rainy weather makes the atmosphere humid, which helps the growth of the producers, and the sunny weather makes them dry out more easily, which makes it more difficult for the producers to grow. However, if a plant receives sun and humidity, it grows and produces food faster.

Grass		
Food Quantity	normal	3-10 units
	sunny	3-5 units
	rainy	8-12 units
	rainy + sunny	15 units
Regeneration	normal	5-7 cycles
	sunny	7-9 cycles
	rainy	3-5 cycles
	rainy + sunny	2 cycles
Nourishment		3 (per unit of food)

Berry bush		
Food Quantity	normal	20-30 units
	sunny	15-25 units
	rainy	25-35 units
	rainy + sunny	45 units
Regeneration	normal	7-9 cycles
	sunny	9-10 cycles
	rainy	5-8 cycles
	rainy + sunny	5 cycles
Nourishment		5 (per unit of food)

Regarding producers, seaweeds are a different case, since they are constantly under water and they receive all of the humidity they could possibly need, so a rainy weather does not really benefit them, since it only takes the much needed sunlight away from them.

Seaweed		
Food Quantity	normal	10-15 units
	sunny	15-20 units
	rainy	5-10 units
	rainy + sunny	10-15 units
Regeneration	normal	5-7 cycles
	sunny	5 cycles
	rainy	5-10 cycles
	rainy + sunny	5-7 cycles
Nourishment		5 (per unit of food)

In the case of creatures, the effect of the phases of the day is more explicit, since each type of creature is adapted to a fixed amount of sun and heat, so they restrict their periods of activity to certain phases of the day. Climate affects them as well, since it can change their behaviour. For example, they might extend their period of activity, reduce it or even stay inactive during the time the climate lasts.

Hare	
Health	30-40
Food	40
Water	40
Life expectancy	12-16 cycles
Growth	4 cycles
Litter	4-6
Speed	10/20/50
Activity period	Dusk-Night
Favourable Climate	Sunny (Activity extended to Dawn)
Unfavourable Climate	Rainy (No Activity)

Rhinoceros	
Health	200-225
Food	250
Water	200
Life expectancy	40-45 cycles
Growth	10 cycles
Litter	1-3
Speed	10/20/30
Activity period	Dawn- Day
Favourable Climate	Rainy (Activity extended to Dusk)
Unfavourable Climate	Sunny (Reduced Movement)

Fox	
Health	75-80
Food	75
Water	50
Life expectancy	25-30 cycles
Growth	6 cycles
Litter	2-4
Speed	10/30/40
Activity period	Dusk-Night
Favourable Climate	Normal
Unfavourable Climate	Rainy (No Activity)

Viper	
Health	20-25
Food	50
Water	50
Life expectancy	15-20 cycles
Growth	4 cycles
Litter	3-5
Speed	10/20/30
Activity period	Day-Dusk
Favourable Climate	Sunny (Activity extended to Dawn)
Unfavourable Climate	Rainy (No Activity)

Vulture	
Health	100-125
Food	100
Water	100
Life expectancy	30-35 cycles
Growth	5 cycles
Litter	3-5
Speed	30
Activity period	Dawn-Day-Dusk
Favourable Climate	Normal
Unfavourable Climate	Rainy (No Activity)

Apart from the relationships established through the food chain, we also seek to recreate other types of interactions between different creatures. We can differentiate these in two groups, having on the one hand the friendly relations which establish that two creatures will behave in a friendly manner between them, and the conflictive relations, in which, on the contrary, two creatures will face each other and will respond by attacking, defending itself or fleeing.

Thus, based on the actual counterparts of the creatures, we have decided to establish that Hares are sociable creatures living in large groups led by a few alpha males and several females, as well as their offspring, inside a territory. They are territorial with members of the

same species that come from different groups, so they will seek to confront intruders if they are not predators. The moment a Hare detects the presence of an intruder, it will let the others closer know this information and so on. The young hares will automatically run away to the burrow, while the adults will face the intruder as long as the intruder is not a predator, the young are in danger or cannot escape.

Continuing with the Vipers, they are solitary creatures with territories reduced to the place where they hide, so they have to venture out of it when hunting. In the case of crossing with a member of the same species, as long as the mating conditions do not exist, they will show which of the two is stronger by making the other one flee.

Foxes are creatures that live alone or in small families of a couple and their offspring, but they are very friendly. This means that they may form small temporary hunting groups with other foxes they encounter or even families if they are of opposite sexes and are not part of one. When there are cubs in the family it is the male who is in charge of hunting while the mother takes care of the cubs.

Rhinoceroses are solitary and very territorial creatures among males and members of different species, often disputing territory among themselves. Females, on the other hand, get along between them, and can share territory and even play with each other. Females can also share territory with other males, not producing confrontations except in the case that she is accompanied by her offspring, in which case she will try to scare off any male. The only species they are not aggressive with is the Hare, being able to play with them and protect them from their predators.

Finally, the Vultures also live in small families of a couple and their offspring. Vultures are not aggressive with each other as long as there is enough food for everyone. If there are eggs or young in the nest, the parents will take turns looking for food to share with the others on their return.

After having established all of these features, we start with the implementation. We start with the simplest but most necessary part: the day/night cycle. Since these project does not need to be visually realistic, we start with a very simple script we can find in the article of Twiik.net [5] and simplifying it. This script receives as an input the desired length of a full cycle and it divides it into four parts.

In order to visualize these changes, we create a scene in Unity3D with a test scenario and an empty object that will serve as a controller. We add the *DayNightCycle* script to this controller, as well as a new script, a *Scene Controller* that will be in charge of managing all of the changes and data. Whenever there is a change in the *DayNightCycle*, the *SceneController* will be reported and it will keep it on the variable *CurrentPhase,* which will modify the intensity of the associated *Light object* [6] as it corresponds.

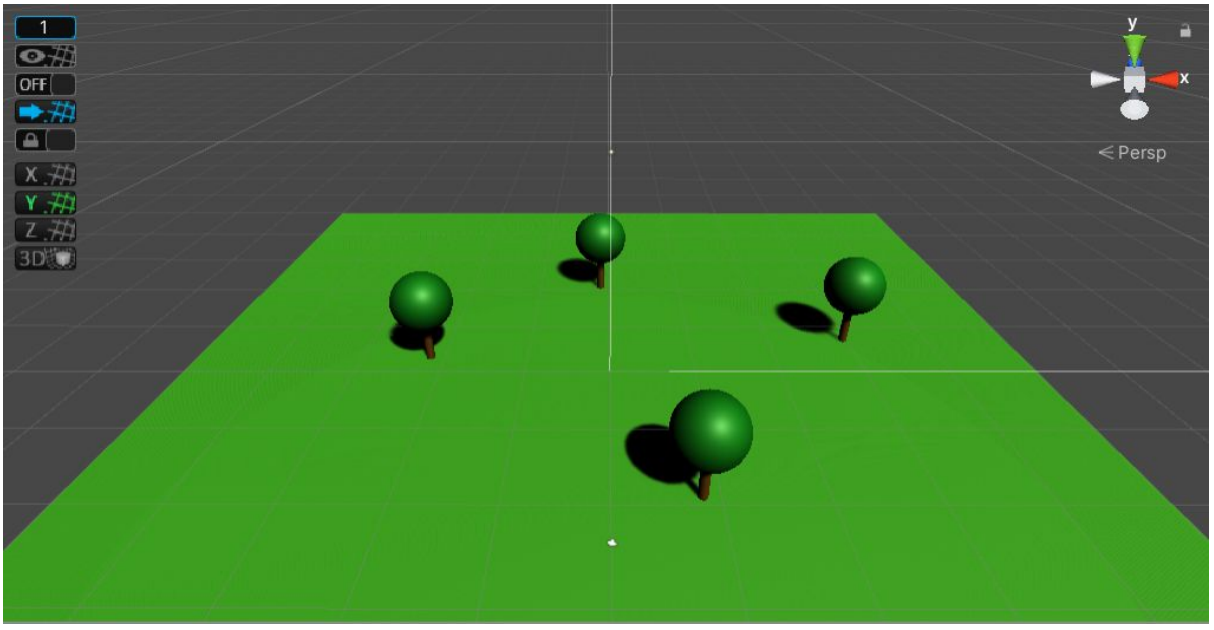

Figura 12. Day/Night cycle implementation scene

Once the day/night cycle is over, we will move on to the climate. For this part, we will create a new script called *WeatherController*. As we mentioned before, this script will represent the changes in the weather that may happen at a certain area in a very simple way while synchronizing them to the changes of the daily cycle. When there's a change of phase, the

SceneController will be in charge of informing the new script, who will check the climate of the previous phase that's kept on the variable *CurrentWeather*, how many phases has this weather laster in the variable *Phases_Passed* and how many phases had it been established to last for in the variable *Phases_Duration*.

Then, it will make a decision based on the data we have explained at the beginning of this section, for which we have created an easy variable in case there may be a need to make small changes. The result of the decision taken by *WeatherController* is returned to the *SceneController*, who will keep it on its own variable *CurrentWeather*. In order to be able to visualize these changes in the climate we will create a system of particles associated to the *SceneController*, wich will be activated when the rainy weather starts and deactivated when it's over. We will make some adjustments to the system of particles so that the particles fall down at our desired speed and to give them a light blue color. In addition, the *SceneController* will also adjust the intensity of the *Light* object so that it gets brighter during sunny phases or darker during the rainy ones.

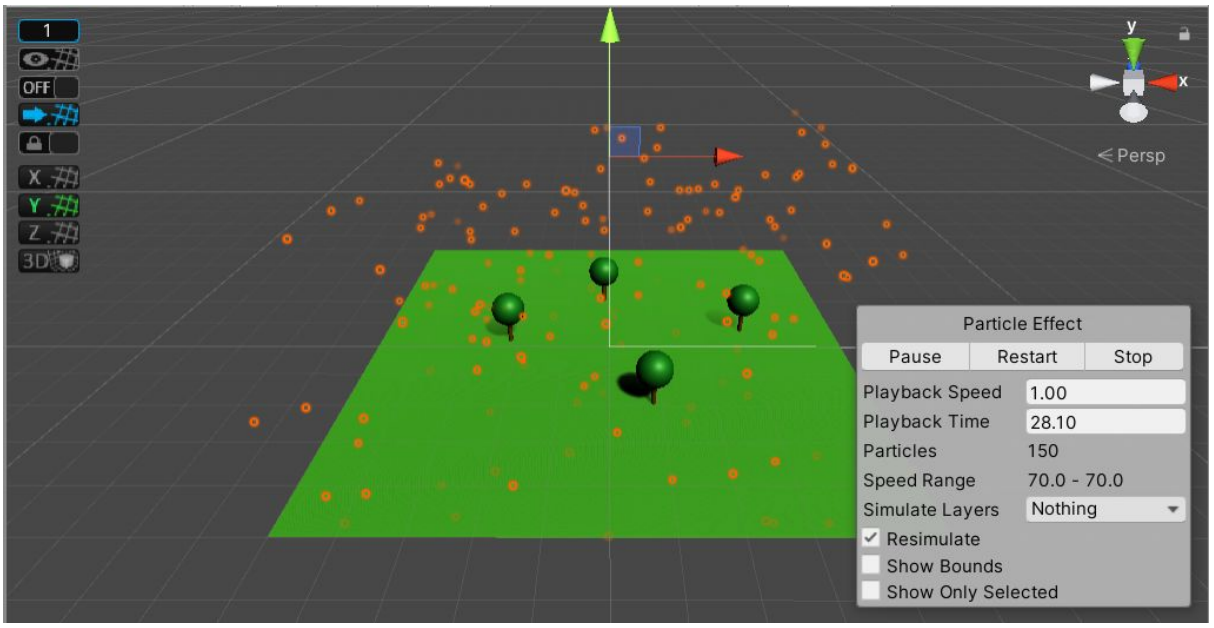

Figura 12. Weather implementation scene

Lastly, we will create a script called *EventController* that will be in charge of managing the calling of events [7]. Events are quite useful for this project, since different Scripts may subscribe one of their functions to one event. Once this event is activated, all of the functions that are subscribed to it will also be activated. In our case, we will create the event *PhaseChange (DayPhase* phase, *Weather* weather) that will be activated by the *SceneController* after obtaining a new phase and climate. These data will also be taken to the event and it will distribute it to all of the subscribed functions.

Now that we have finished with the day/night cycle and climate, we can get to work with the Producers.

Since all generators basically share the same characteristics and behaviors, we decided to use Unity's *Scriptable Objects.* As we mentioned earlier, these *Scriptable Objects* [8]can contain data. This data can be modified from the same Unity inspector, and these changes can be applied to all of the references of this *Object*, even if they are being executed in the project, which makes it a very useful tool. Thus, we create a *Script Productor* that descends from the class *ScriptableObject* and to which we add the following line of code:

```
[CreateAssetMenu(fileName = "Producer", menuName = "ScriptableObjects/Producer")]
```

This will allow us to create new objects like *ScriptableObject* that will use the parameters of *Productor* that we will explain later just by doing a right click, selecting create and then *Productor* at Unity's *Project* window.

Next, we will move on to *Productor*'s data. As we showed in charts x1, x2 and x3, producers have a series of parameters from which they will make a random decision about how many food to generate and how long it will take for them to generate it. This random decision is taken within a range and it takes into account the current weather conditions. In order to simplify the amount of variables necessary we will need each of them to come from the type *Vector2Int* [9], which allows us to keep two full values inside of them. Besides, we will need another variable that will specify the amount of nourishment each unit of food supplies. After

creating the script we can move on to the *Project* window in Unity and create the *ScriptableObject* for each type of producer as we have previously explained and fill it with the data collected in the chart.

Now that we have established the data from the producers we have to create the script that will manage this kind of information: the *ProducerController*. This script comes from *MonoBehavior* and it will contain a *Producer* variable to which we can assign the *ScriptableObject* we want. When we initiate the scene, the function *Awake* from *MonoBehavior* will activate selecting all the models that represent food generated in the *meshes* vector [10] of the object we assign this script to and which will act as a productor. Another function of the *ProducerController* is *TakeFood(int x)*, which will be received as a parameter of full value that will be substracted to the amount of food the producer currently has, stored in the variable *foodQuantity* and returning the appropriate nourishment according to the intake of food.

You cannot take more food than there is, because if you substract the value of x from the current food and is less than zero, the difference will be subtracted before the nutrition is returned. When the amount of total food is zero the meshes will be disabled, making them invisible until we can activate them again, and we will generate a random value for the variable *PhasesToReg*, taking into account the regeneration values in a clear climate.

The function *RegenerateFood*() will be called when the indicated phases of the day have been completed in *PhasesToReg.* The amount of food generated will be obtained randomly within the range associated to the climate that has been on while it was waiting and it will be kept in the *FoodQuantity* variable. We can figure out the climate that has been going on thanks to the Boolean [11] called *rained* or *sunnied.* Finally, we will reactivate the meshes by turning them visible.

Finally, we have the *PhaseChanged(DayPhase* phase, *Weather* weather) subscribed to the *PhaseChange* event, which, after receiving the new phase and climate, will check if all of the *PhasestoReg*'s phases have been completed. If not, it will update the phase meter called *PhasesPassed*. Next, it will check if the new climate is sun or rain and it will update the value

of *PhasesToReg* accordingly, as well as *rained* or *sunnied* so that it does not repeat it until it regenerates. If *sunny* or *rained* are "true", *PhasesToReg* will update with the corresponding value. If the new *PhasesToReg* is inferior to *PhasesPassed* it will be called *RegenerateFood()*. In order to finish with the script we will need a function that returns us a Boolean that tells us if a producer has food or not which we will call *HasFood()* and that will be used by the creatures.

Once the script is ready, we can create the different objects we can associate it to and which will represent a type of producer.

By using the grass object in the scene of Unity, we create an empty *GameObject* [12] we will call *Grass* and, inside of it, we will place another empty object called *Food_Model*, inside of which we will create several cubes that we will deform until the ensemble acquires the shape of grass. Once we've done that, we will add *Grass* to the *ProducerController* script, we add the corresponding *ScriptableObject* in the inspector and the variable *Food_Meshes,* from which we will assign the object we want it to take the meshes from in order to make them invisible, and we assign *Food_Model*. Lastly, we add a *RigidBody* [13] and a *BoxCollider* [14] to *Grass* so that the creatures can detect it. Finally, we only have to transform the *Grass* object into a *prefab* [15] by dragging it from the *Hierarchy* window to the *Project* one.

In order to create seaweed we would follow the same process as we've done with Grass, but the berry bushes would follow a slightly different one. We would basically need to create two children in the *GameObject,* one for the model of the bush and another one for the berries.

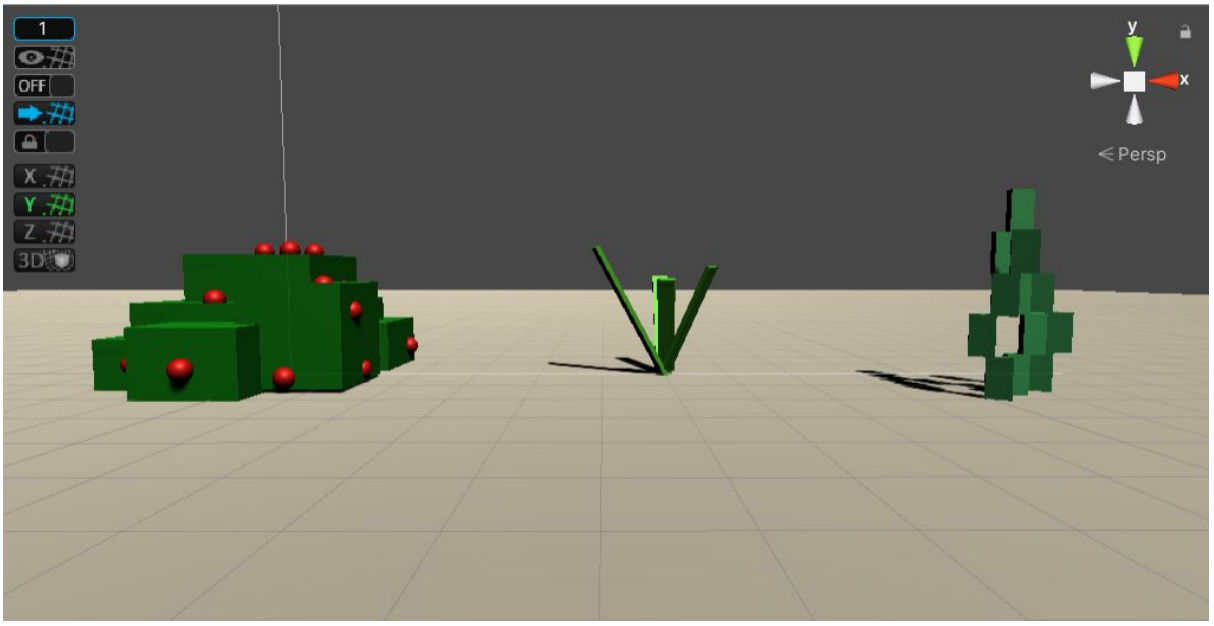

Figura 13. Bush, Grass and Seaweed models

Now that we have completed the base of the project, we need to begin with the implementation of the creatures.

Frist of all, we need to make the creatures move, because it would be impossible for them to complete any of the behaviours we design if they can't move around. We will use the *NavMesh* tool in Unity for that [16]. With this tool, we can create meshes from the scenario through which our *NavMesh Agents* [17] can move. These agents will use these meshes to trace the fastest path from their actual position in the mesh to another one (the so called pathfinding), avoiding obstacles and other Agents on the way.

After creating a new scene we create the scenario we want our objects, in this case, our creatures, to move in. Next, we create an empty object we will call *NavMeshController* and to which we will add the *NavMeshSurface* [18] controller. After hitting the "bake" button in the inspector, it will generate a mesh from the scenario. The bake of the mesh is built taking into account the objects of the scenario, the slopes and the steps. If we want it to only take into account certain objects when creating the mesh, we can simply select them, just as it says in Unity's documents [19].

Next, we have to create any *Object* in the scene, assign the *NavMesh Agent* to it, as well as a script to which we will indicate that when the user clicks on the scenario, that spot will be fixed as a destiny and the creature will move to it.

This is the most basic move we can create, but since we have different types of creatures with different sizes and even some with different ways to move, we need to make it more complex. In the Navigation window inside of Agents, we will create a different type of Agent for each creature, to which we can assign the height and width values of the Agent (since it's shaped like a cylinder), as well as the maximum height of the steps it can overcome and the maximum gradient angle. Next, we will create a prefab for each creature to which we will assign the *NavMeshAgent* component, and inside of ir we will select the corresponding type of Agent.

Next, in a new scene we will create a more complex type of scenario. In order to do so, we will use the ProBuilder [20] and ProGrids [21] assets, available at Unity's AssetStore. After installing them, we will create a new empty object that we will call Scenario. In there, we will create a plan we will call Terrain and that we will modify with the Probuilder tools. In these modifications we will generate a series of depressions in the terrain and then we will separate the original mesh, creating new objects we will place inside Terrain. We will use this to associate a *NavMeshModifier* [22] component to each of these objects, which will allow us to assign a different area to this object.

The *NavMesh* areas mark out the area or the type of terrain inside a mesh, but we can assign higher costs when doing the pathfinding, so that we can avoid going through this terrain as much as we can. We can also decide which areas we want the object to displace in from the Area Mask in the *NavMeshAgent* component of each prefab.

For this project we will need two different areas, apart from those who are given by default: Water with a cost of 3 and Skye with a cost of 0. The Water area will be assigned to the objects we have separated in the initial mesh. We will create a new transparent plan placed slightly above the scenario and we will assign the sky area.

Lastly, before making the bakes we need to take into account that it is possible that I might need to go from one mesh to another. For example, in the case of vultures, they will move from Sky to the ground to feed or rest. We may also want to go from one area from the same mesh to another one that is close but not necessarily connected. To solve this problem we use the *NavMeshLink* [23], componentes that allow us to create connexions between two *NavMesh*, allowing a creature to go from one to another when it enters a limited area, as well as to assign an Area to the link.

In the case of Vulture, we will need to be able to go up or down from any point of the mesh. For that reason, we will create a prefab that will contain a *NavMeshLink* component that we can spawn in any place to adjust it to suit our needs and then destroy it once it's used.

Lastly, for each type of creature we will need to add a *NavMeshSurface* to the *NavMeshController* and adjust it to their characteristics.

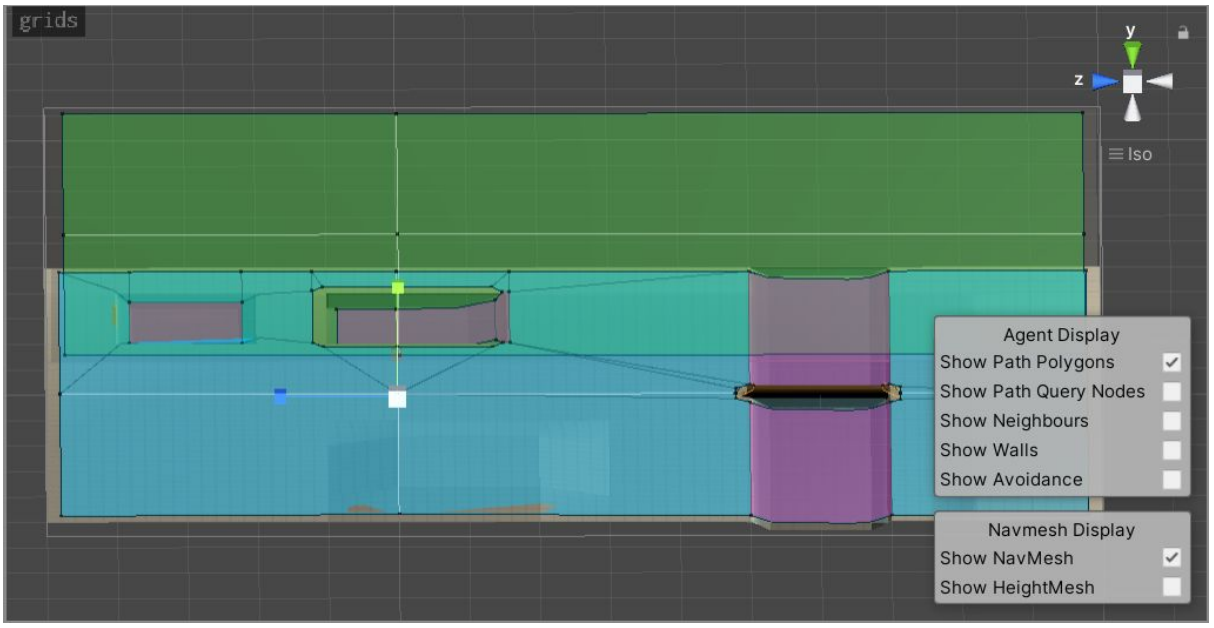

Figura 14. Testing NavMesh Scene

Now that we have sorted out how to create movement, the next step is to create a detection system that imitates a real animal's senses. This is quite simple, since we only need to create a new empty object inside the corresponding prefab of each creature. This empty object will be called Senses and we will add a *Sphere Collider* [24] component with a marked Is Trigger variable and a Script called Senses to it. This script derives from MonoBehaviour and it will detect when an object enters inside the Sphere Collider, then check the distance and direction from the centre of the collider and, if it's within the creature's range of vision, it will figure out what type of object it is. In order to avoid detecting unnecessary objects, we will create a series of *Layers* dedicated to specific objects such as creatures, producers, water, terrain and other creature's senses. We can also detect a creature through hearing, so whenever it enters inside the Collider, we will check how loud the noise is and we will decide if it detects it or not.

Now, we move on to create the collision system. In order to do so, we create an empty object inside the creature's Prefab called *Body*, which will be in charge of carrying the model of the creature. *Body* will have a *Box Collider* associated to it, whose dimensions will be adjusted to the size of the boy. We also add a Script called *Interaction* that will be in charge of detecting when a creature is touching food, water or another creature.

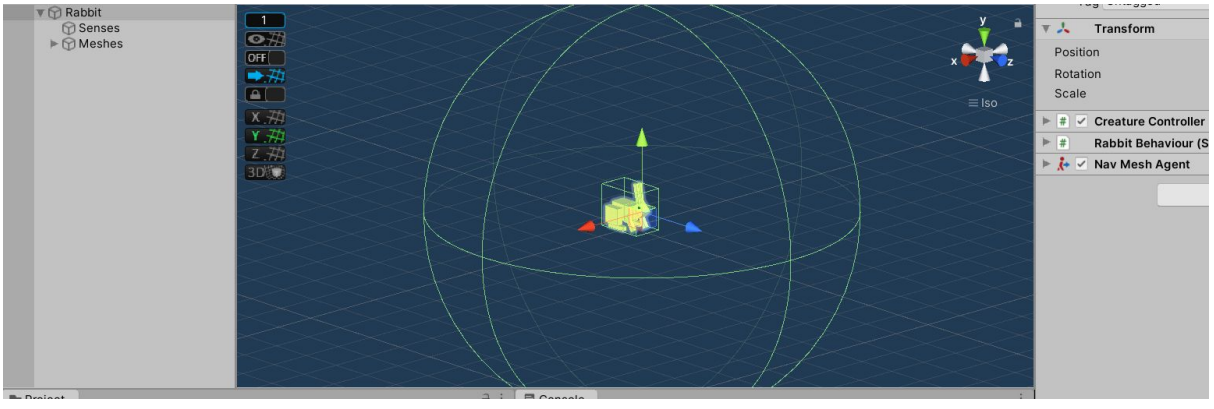

Figura 15. Hase Prefab

Finally, we need to create the rest area. In order to do that, we will create a new prefab that will serve as the base for the other rest areas. This prefab Home will have two children: Meshes, the one that will contain the model and a Sphere Collider, and Territory, the one that will have a *Sphere Collider* and a marked *Is Trigger* associated to it. This *Sphere Collider* will delimit the territory belonging to the inhabitants of the rest area. Home will have an associated script that will be also called Home. This script will be in charge of managing the groups of creatures that inhabit it, saving the information of the places where they can find food and water inside the scenario, as well as spawning new creatures when a pregnant female that is in the group meets the gestation time or when the project starts. In the event that the number of individuals of a particular species is excessively high, the SceneController will warn the different Homes containing this species so that they do not generate new creatures until the number is stabilized. Similarly, SceneController will warn the Homes to forcefully increase the number of individuals in case the number of individuals is too low.

The new creatures will only be spawned when the phase of the day when they begin their activity starts, so they will need to have a function subscribed to the *PhaseChange* event. For this reason, it will need to contain the data from the Scriptable Object of the creature. In addition, it will save all of the generated creatures inside a Group class, and the creatures will get deleted after they have died. Since this script derives from Monobehaviour, it will also use the Awake function, in which it will cast a SphereCollider the size of the territory in order to detect all sources of water and producers in the area the creature can feed from and then pass them on to the spawned creature. Creatures will be able to update these lists with new places where they have found food outside the territory or water.

When a creature grows up, if it has no place within the group it will abandon the group and the Home, going in search of a new one that is not inhabited or that has a group with enough space. If a Home loses all the creatures in the group it will remain empty again.

Now that we've created everything we need in order to build the creature's behaviours, we need to move on to the creation of the Scriptable Object Creature that we will be using as a base to store every creature's information. Its functioning is quite similar to the producer's, but it also contains the information from the charts we showed at the beginning of this

section. Now, we have to create the scriptable object of each creature and fill it with the corresponding data.

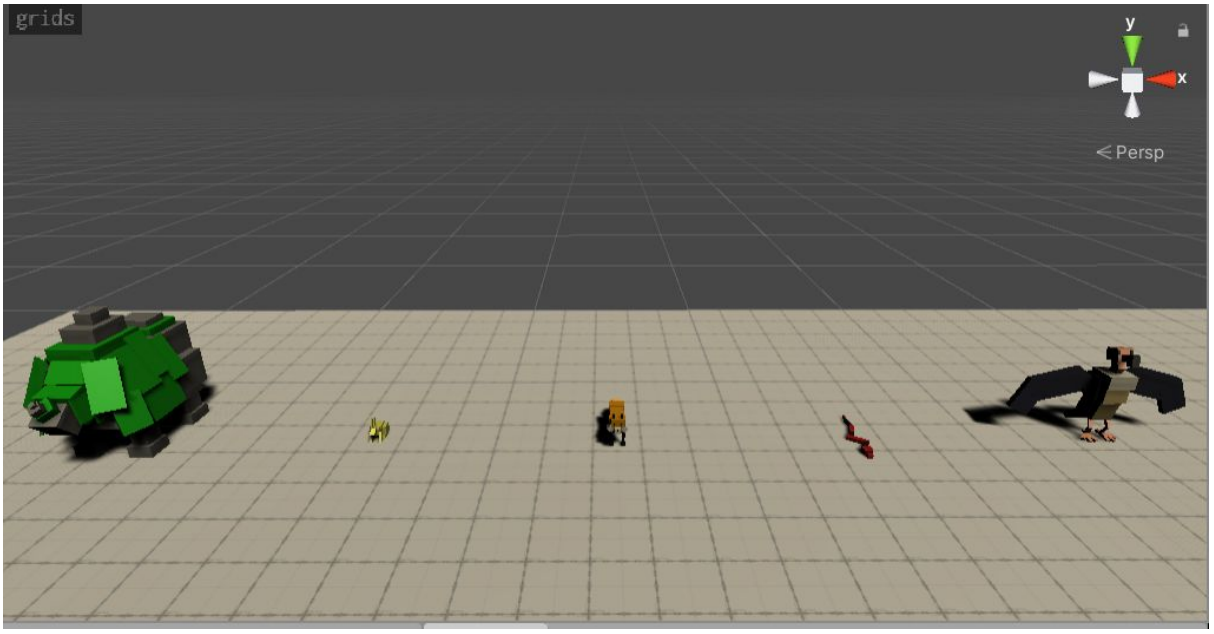

Figura 16. Rhino, Hase, Fox, Viper and Vulture Models

Next, we create the CreatureController script that will be in charge of managing the creature's internal and external data. This script derived from MonoBehviour will take the data from the ScriptableObject in order to establish the maximum life, the amount of food and water etc. at the Awake function. This data should be updated every few seconds, reducing the amount of health, water and food in case there wasn't enough previously. The number of seconds between updates will vary according to the creature's metabolism and may be slower and last longer with less food than other creatures. We will use the IEnumerator [25] for that. IEnumerator are lines of code that execute parallely to the script they belong to so that they don't hinder their own functioning. CreatureController will also be in charge of receiving data about the changes of phase and climate, and for that reason it needs to have a function subscribed to that event.

Before we can implement each creature's behaviour and their controller, we must implement the base of the artificial intelligence. As we have previously mentioned, we intend to use

Hierarchical State Machines in this project, because they can contain other State Machines inside of them and combine them with *Behaviour Trees*. In order to implement the HSM, we resorted to the pseudocode that we can find at *AI for Games* [26] in the chapter dedicated to Decision Making. This pseudocode contains a small error that we had to fix in order to achieve our goal. It was a very simple error: the update function returned us back to the "Action" list instead of an object of the UpdateResult class. Apart from that, we had to adapt the code, since C# does not allow a double inheritance, so it made the SubStateMachine inherit from State and it added the UpdateDown function from the StateMachine. We also had to change the Action class using the Task class, which would be the base of *behaviour tree*s.

We used the code we found in Reddit called Simple *Behaviour Tree* implementation for Unity [27] for the creation of the *Behaviour Trees* after several attempts to implement the pseudocode proposed by the author of the book previously mentioned. We also modified some of the parts of the original code in order to create new nodes, like the *NoDeterministicSelector* and the *NoDeterministicSequence* [28] proposed by Millington, as well as the BlackBoard used for the communication between the the *CreatureController* and the *Tasks*, as well as creating new Leaf nodes with personal actions the creature will share. Now that we have implemented the base of the AI, we only need to create the controller that will manage it.

This controller will be the *CreatureBehaviour* script that derives from *MonoBehaviour* and that will contain a reference to the *CreatureController* of the creature. This script will create a new *BlackBoard* when initiated, and it will also contain a *Hierarchical State Machine* and a list of *Tasks*. *CreatureController* will be in charge of calling the function *updateHSM* from *CreatureBehaviour* each time an event is triggered. The *updateHSM* function will call the update of the HSM, which will study if the trigger received causes any changes in the behaviour of the current state and it will return a list of *Tasks* as an answer. This list of Tasks will be executed by an *IEnumerator* in order to avoid performance problems, recreating the creature's behaviour under the corresponding trigger.

After trying this, we created a new script that inherits from this *CreatureBehaviour* class called *HareBehaviour*, since we are going to try it on this creature. Instead of the *CreatureBehaviour*, we assign *HareBehaviour* the prefab and inside of it we initiate the new HSM with its states, *SubHierarchicalStates*, *Transitions* and *BehaviourTrees*. This test HSM is a simplified version of the one presented in the section 3.2 and it only has the *SubStateMachine* Chill inside of active. We have also withdrawn the *State Interact* from it. Now we create a new scene with a home that spawns a hare, some generators and water.

After many tests and trials, and many attempts to correct errors, we haven't achieved our main goal, since the behaviour does not work completely well, but we do not have any more time to continue with this project.

4.2 Results

After plenty of research and many hours spent implementing, we have completer a functional day/night cycle, although it is not completely correct since we run out of time to properly finish it. We have also managed to create a functional Climatic system that we could expand to other types of climates.

Regarding generators, we have managed to create a functional structure from which we could recreate almost any type of generator. With respect to creatures, we have created a functional detection system based on sight and hearing and a structure from which we could recreate almost any existing creature in the real world as well as their behaviours grosso modo. Unfortunately, we haven't completed the main objective of this project, which was to place several types of creatures in a scenario to interact between each other as they would normally do.

5. CONCLUSIONS AND FUTURE WORK

5.1 Conclusions

The main problem I had with this project is that I was too ambitious. I was trying to do so much with very little time and resources. The situation we live in today didn't help much, because the COVID-19 pandemic has had a huge impact and we have been affected in many aspects. Although I haven't completed my main goal, I am really proud of the work I have done, because I have managed to finish almost everything I had set my mind on doing. I would have really loved to finish this project, because I know I have dedicated myself full-time to it and I know in other circumstances, and with a little bit of help, I could have definitely finished it on time.

5.2 Future Work

I was thrilled to make this project, especially because I really believed I could use my results in future video games. I really hope I can finish it some time in the future and improve many of its parts. I believe I could implement different forms and stats depending on the age or gender of each species of creature or I could improve Artificial Intelligence using Fuzzy Logic and Fuzzy State Machines. I have so many ideas for this project it's such a pity I couldn't make it on time. I know someday I will finish this project, just to prove myself I was capable of doing it and also because I believe it will be really useful in the future.

BIBLIOGRAPHY

[1] Non-Playable Character on Wikipedia:
https://en.wikipedia.org/wiki/Non-player_character

[2] Ecosystem on Wikipedia: https://en.wikipedia.org/wiki/Ecosystem

[3] State Machine on Wikipedia: https://en.wikipedia.org/wiki/Finite-state_machine

[4] Behaviour Trees on Wikipedia:
https://en.wikipedia.org/wiki/Behavior_tree_(artificial_intelligence,_robotics_and_control)

[5] Simplest possible day night cycle in Unity 5:
http://twiik.net/articles/simplest-possible-day-night-cycle-in-unity-5

[6] Unity Documentation 5.3 Light:
https://docs.unity3d.com/es/530/Manual/class-Light.html

[7] What are Events? (C# Basics) - Youtube Tutorial
https://www.youtube.com/watch?v=OuZrhykVytg&ab_channel=CodeMonkey

[8] Unity Documentation 2019.4 - Scriptable Objects
https://docs.unity3d.com/Manual/class-ScriptableObject.html

[9] Unity Documentation 2019.4 - Vector2Int:
https://docs.unity3d.com/ScriptReference/Vector2Int.html

[10] Unity Documentation 2019.4 - Meshes:
https://docs.unity3d.com/Manual/class-Mesh.html

[11] Boolean datatype on Wikipedia: https://en.wikipedia.org/wiki/Boolean_data_type

[12] Unity Documentation 2019.4 - GameObject:
https://docs.unity3d.com/ScriptReference/GameObject.html

[13] Unity Documentation 2019.4 - RigidBody:
https://docs.unity3d.com/ScriptReference/Rigidbody.html

[14] Unity Documentation 2019.4 - Box Collider:
https://docs.unity3d.com/es/2019.4/Manual/class-BoxCollider.html
[15] Unity Documentation 5.3 - Prefabs:
https://docs.unity3d.com/es/530/Manual/Prefabs.html

[16] Unity Documentation 2019.4 - Navigation System:
https://docs.unity3d.com/Manual/nav-NavigationSystem.html

[17] Unity Documentation 2019.4 - NavMeshAgent:
https://docs.unity3d.com/ScriptReference/AI.NavMeshAgent.html

[18] Unity Documentation 2019.4 - NavMeshSurface:
https://docs.unity3d.com/Manual/class-NavMeshSurface.html

[19] Uniry Documentation 2019.4 - Building a NavMesh:
https://docs.unity3d.com/Manual/nav-BuildingNavMesh.html

[20] Unity - ProBuilder: https://unity3d.com/es/unity/features/worldbuilding/probuilder

[21] Unity Manual - About ProGrids:
https://docs.unity3d.com/Packages/com.unity.progrids@3.0/manual/index.html

[22] Unity Documentation 2019.4 - NavMesh Modifier:
https://docs.unity3d.com/Manual/class-NavMeshModifier.html

[23] Unity Documentation 2019.4 - NavMesh Link:
https://docs.unity3d.com/Manual/class-NavMeshLink.html

[24] Unity Documentation 2019.4 - SphereCollider:
https://docs.unity3d.com/ScriptReference/SphereCollider.html

[25] Unity Documentation 2019.4 - IEnumerator:
https://docs.unity3d.com/ScriptReference/MonoBehaviour.StartCoroutine.html

[26] AI for Games by Ian Millington, third edition

[27] Simple Behaviour Tree implementation for Unity:
https://www.reddit.com/r/gamedev/comments/46ywep/simple_behaviour_tree_implementation_for_unity/

[28] Nondeterministic algorithm on Wikipedia:
https://en.wikipedia.org/wiki/Nondeterministic_algorithm