UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Bachelor as Computer Engineering

BACHELOR THESIS

Simulation of Ecosystem Evolution and Avoidance of Extinction via a Genetic Algorithm in Unity3D

Author: Raúl Martín Hernández
Advisor: Laura Isabel Climent Aunes

December 2022

All rights reserved.
No reproduction in any form of this book, in whole or in part
(except for brief quotation in critical articles or reviews),
may be made without written authorization from the publisher.

© 20-12-2022 by UNIVERSIDAD AUTÓNOMA DE MADRID
Francisco Tomás y Valiente, n° 1
Madrid, 28049
Spain

Raúl Martín Hernández
Simulation of Ecosystem Evolution and Avoidance of Extinction via a Genetic Algorithm in Unity3D

Raúl Martín Hernández

PRINTED IN SPAIN

To my friends, who made this penance bearable

As a child, I considered such unknowns sinister.
Now, though, I understand they bear no ill will.
The universe is, and we are.

Solanum

Resumen

A menudo encontramos en las noticias informes sobre una nueva especie que se ha introducido en un ecosistema extranjero, informes sobre como las condiciones ambientales de un ecosistema han sufrido un cambio, como el aumento de temperatura de los mares, o incluso informes sobre otra especie más que se ha extinguido. Esta tésis explica la importancia que tienen este tipo de noticias mediante el uso del modelo presa-depredador, el cual representa la evolución de las poblaciones de dos especies que requieren de un equilibrio muy delicado para coexistir.

Esta tésis se centra en el desarrollo de una aplicación usada para simular la evolución de un ecosistema. Específicamente, se centra en las decisiones de diseño y detalles particulares de la implementación, con un especial énfasis en la optimización de rendimiento. El objetivo que se persigue con esta aplicación es analizar el comportamiento que se presenta en el modelo presa-depredador y, adicionalmente, se proponen ciertos algoritmos que sirven tanto para analizar los resultados de las simulaciones como para tratar de encontrar configuraciones que proveen un ecosistema estable, en el que las probabilidades de que sus especies se extingan son muy bajas.

Aunque el propósito principal de este trabajo es explorar el modelo presa-depredador, la aplicación también ha sido diseñada para permitir explorar otro tipo de propiedades, como el problema de teoría de juegos conocido como el dilema del prisionero, u otras relaciones que se pueden dar en la naturaleza, como el mutualismo, el parasitismo, la simbiosis, etc. Esto se consigue mediante la codificación de nuevas reglas en el ecosistema o en el comportamiento de sus especies. Un vídeo de demostración [1] que muestra la aplicación resultante está disponible en YouTube.

Palabras clave

Simulación, ecosistema, evolución, algoritmo genético, Unity3D, modelo de presa-depredador

Abstract

We often find in the news reports about a new species that has been introduced into a foreign ecosystem, reports about the environmental conditions of a certain ecosystem having suffered a change, like the rising temperature of the seas, or even reports about another species that has gone extinct. This thesis explains the relevance of these types of news by using the prey-predator model as an example, which represents the evolution of the populations of two species that require a delicate balance to coexist.

This thesis focuses on the development of an application used to simulate the evolution of an ecosystem. Particularly, it covers the design decisions and specific details of the implementation, with an important emphasis on performance optimization. The intent of this application is to explore the behavior of the prey-predator model and, additionally, it proposes some algorithms that serve both to analyze the results of simulations and to try finding a configuration that provides a stable ecosystem, in which there is a very low probability that its species go extinct.

While the main focus of this work is the exploration of the prey-predator model, the application has been developed to also allow the exploration of other properties, like the game theory problem known as the prisoner's dilemma, or other intricate relationships that also appear in nature, such as mutualism, parasitism, symbiosis, etc. This is achieved via the codification of new rules in the ecosystem or the behavior of its species. A demonstration video [1] that shows the final result of the application is available on YouTube.

Keywords

Simulation, ecosystem, evolution, genetic algorithm, Unity3D, prey-predator model

Table of Contents

1 Introduction **1**

1.1 Objectives and Motivation 1

1.2 Tools and Resources 2

2 State of the Art **3**

2.1 Unity3D Game Engine 3

2.2 Evolutionary Algorithms' Basics 3

3 Design **5**

3.1 Functional Requirements 5

3.2 Non Functional Requirements 7

3.2.1 Documentation 7

3.2.2 Localization 7

3.2.3 Performance 7

3.3 Use Cases 8

3.3.1 Configuring and Running a New Simulation 8

3.3.2 Analyzing the Results of a Simulation 8

3.3.3 Finding a Balanced Ecosystem Through the Genetic Algorithm 9

3.4 Architecture 9

3.4.1 Simulator 10

3.4.2 Data Serialization and Statistics Retrieval 11

3.4.3 Genetic Algorithm 12

4 Development **13**

4.1 Methodology and Workflow 13

4.2 Implementation 14

4.2.1 Ecosystem Simulator 14

4.2.2 Statistics Gathering and Generation 17

4.2.3 Genetic Algorithm 21

4.2.4 Graphical User Interface 27

4.2.5 Performance Optimization 28

4.3 Continuous Integration 30

5 Experiments and Results **31**

5.1 Experimenting with the Simulator 31

5.1.1 Dropping Fruits Experiment 32

5.1.2 Balanced Constants Experiment 33
5.1.3 Unbalanced Constants Experiment 35
5.1.4 Sharing Resources Experiment 36
5.2 Experimenting with the Genetic Algorithm 37
6 Conclusions **39**
6.1 Future Work 40
Bibliography **42**
Acronyms **43**
Appendices **45**
A Functional Requirements for the Ecosystem Evolution Simulator Application **47**
A.1 Ecosystem Simulator 47
A.2 Statistics Gathering and Generation 55
A.3 Genetic Algorithm 56
A.4 Graphical User Interface 57
B Class Diagrams **61**
C Exploration of Evolving Clusters **65**
D Plants Requirements' Tracker Texture **67**
E User Guide **69**
F Resources for the Ecosystem Evolution Simulator Application **83**

Lists

List of figures

4.1 Perlin Noise sample altered by sigmoid function . . . 15
4.2 Expected result of the species detection algorithm . . . 19
4.3 Species detection algorithm procedure . . . 21
4.4 Prey-predator model's representation . . . 22
4.5 Rabbits vs wolves population evolution . . . 22
4.6 Example codification of a chromosome in the genetic algorithm . . . 24
4.7 Explanation of value encoding mutation in the genetic algorithm . . . 25
4.8 Explanation of crossover methods in the genetic algorihtm . . . 26
4.9 Example of plot generated by the *UIGridRenderer* component . . . 27

5.1 Correspondence between the soil map and the world . . . 31
5.2 Fruits dropping experiment's species detection comparison . . . 32
5.3 Balanced experiment's scene . . . 33
5.4 Balanced experiment's heat maps . . . 33
5.5 Balanced experiment's genes' evolution . . . 34
5.6 Unbalanced experiment's heat maps . . . 35
5.7 Result of an experiment with trees . . . 36
5.8 Result of the experiment with the genetic algorithm . . . 37

B.1 Class diagram of the *Genetic Algorithm* subsystem . . . 61
B.2 Class diagram of the *Ecosystem Simulator* subsystem . . . 62
B.3 Class diagram of the serialization and statistics gathering classes . . . 63

C.1 Possibilities for evolving clusters . . . 65

D.1 Plants requirements' tracker texture . . . 67

E.1 *Main Menu* window . . . 69
E.2 *New Simulation Configuration* window 1 . . . 71
E.3 Visualization of the effect of the spawn map slider . . . 72
E.4 *New Simulation Configuration* window 2 . . . 73
E.5 *Simulation Execution* window . . . 74
E.6 *Load Simulation File* window . . . 75
E.7 Loading window . . . 75

E.8 *Simulation Results* window 77
E.9 *Simulation Navigation* window 78
E.10 *GA Configuration* window 80
E.11 GA configurable stop conditions 80
E.12 *GA Execution* window 82

1 Introduction

This thesis explains the development process followed to build an application called Ecosystem Evolution Simulator (EES) . The structure of the document starts with the motivation and objectives of the project, along with some context via this introduction and the state of the art. After that, the design section explains in detail the requirements, usage, and code structure of the application. The implementation section covers some of the most interesting details of the codification of certain algorithms required, along with some explanations on why they are important. Finally, the document ends with a brief experimentation with the application.

1.1 Objectives and Motivation

Behind the main motivation of this thesis lies the urgent need to understand the fragility of ecosystems in the real world. The news reports that talk about rare species going extinct and the rising temperature of the seas do not sound severe enough to grasp where the consequences of these problems may lead us in the future. To understand the repercussions that these events may have, this thesis covers the development of an application capable of running simulations of a simplified ecosystem. Particularly, the design decisions behind this application are focused on the analysis of the prey-predator model. This model explains the evolution of the populations of two species that require a delicate equilibrium to coexist. If this equilibrium is disrupted by events like, for example, hunting or unexpected migrations, those two species can go extinct, unleashing a domino effect over the whole ecosystem they live in.

Before starting, we first need to understand what a simulator really is. A simulator is a software program or system that is designed to replicate the behavior or functions of a real-world system or process. Simulators are often used in a variety of fields, including engineering, science, and business, to model and test complex systems or processes without the need for physical experimentation. Simulators can be used for a wide range of purposes, such as training, analysis, and prediction. For example, a flight simulator can be used to train pilots, or to test the performance of an aircraft design. A financial simulator can be used to analyze market trends or test investment strategies. Simulators can be based on various types of models, such as mathematical, physical, or statistical models. They can also be

built to varying degrees of complexity, from simple models that simulate a few key variables, to complex models that replicate a system in great detail.

The simulator built in this project aims to generate data for certain problems or properties that can be later analyzed by the user, which is achieved by defining its own set of simplified rules based on mathematical and statistical models to represent the evolution of an ecosystem. Efficiency is a key factor when building non-real-time simulators, and this thesis will also cover some of the techniques exploited to provide this efficient simulator. Besides, a Genetic Algorithm (GA) is used on top of the ecosystem simulator to avoid the extinction of its species. The specific objectives of this thesis are:

O-1.– Develop a simulator capable of recreating genetic evolution over a specific set of species.

- **O-1.1.–** Propose interesting rules for the entities to explore their behaviors and relationships.
- **O-1.2.–** Process the data obtained from a simulation and design the visuals necessary to get a representation that can be used to analyze the results.
- **O-1.3.–** Design the application to be capable of running simulations as fast as feasible.

O-2.– Develop a graphical application around the simulator to simplify the process of its configuration and the interpretation of its results.

O-3.– Build a machine learning model capable of configuring the simulator in order to prevent species from going extinct.

O-4.– Analyze the results obtained via the simulator, aiming to find relationships that appear in nature in the real world, like mutualism, parasitism, symbiosis, etc.

1.2 Tools and Resources

This application has been built with the game engine *Unity3D V2.21.17f* [2], which uses *C#* [3] as a scripting language. Additionally, the language *hlsl* [4] is used to write compute shaders, which can run code in the GPU. The IDE used to implement the application is *Visual Studio 2019* [5], along with *PlasticSCM* [6] as the version control tool integrated with Unity. Other additional resources like icons or fonts are listed in Appendix F.

2 State of the Art

2.1 Unity3D Game Engine

The application is based on the Unity3D game engine, so it is important to understand how it works to follow some aspects of the development process. Unity offers two types of interaction: edit mode and play mode. Edit mode allows the user to build the video game or graphical application, while play mode compiles and runs the application from within the editor to preview the results and allow some testing on the final product.

The most interesting bit comes with the editor mode. Unity is a scripting-based game engine that relies on *GameObjects*, its basic building block. A *GameObject* is a container for smaller blocks called "components," which are directly associated with a script. Each component defines a critical aspect of a *GameObject*, such as its spatial location, physics, particle emitter, renderer, sound source, and so on. Unity manages its objects in a hierarchy, and every object has a *Transform* component, which stores information about the scale, position, and rotation. With this information, unity places each object in a scene. With this hierarchy and scene, the developer can build its graphical application by creating *GameObjects*, placing them in the scene, and adding components to said objects to construct systems, actors, scenery, or any other necessary element. Additionally, Unity offers an API to extend its editor. With it, the developer can design new windows and visualizations to create tools to develop their application faster. Code is not usually executed during the editor mode, but thanks to this API the developer can create helper tools to, for example, manage save files or even test a section of code without running the whole application.

2.2 Evolutionary Algorithms' Basics

Later in the development, a Genetic Algorithm will be involved, so it is important to understand how they work. These algorithms generate a population, a collection of chromosomes that encode possible solutions to the specific problem. With this, the best solutions have a higher chance of having descendants in the next generation. The design of these algorithms depends entirely on the problem they

try to solve. For this, genetic algorithms provide a set of tools called "genetic operators," along with a common pipeline and a set of configurable probabilities and parameters.

Codification Defines the data structure that contains a chromosome to represent a solution to the problem. This structure can be anything, from arrays of bits to binary trees or images.

Objective Function This function measures how good the solution a chromosome represents is. It is entirely dependent on the problem to solve, and it is a key factor to define what the goal of the optimization is.

Selection Method A method used to determine how chromosomes are randomly selected to have descendants in the next generation, usually in proportion to their fitness score.

Crossover Method These methods define how new chromosomes are generated from their parents, which depends on the codification used. Some basic crossover operators have been defined for the most common codifications, usually based on arrays of numbers, like one-point crossover.

Mutation Method Defines how the values within a chromosome are changed at random during breeding. In simple codifications like binary arrays a simple bit flip can serve, but for example with value encoding (real number values) other arithmetic operators appear.

Elitism Defines the proportion of the population with the top fitness score that will survive into the next generation, to ensure that the best solutions found so far are not discarded.

Stop Condition Used to define when the algorithm should consider that a good solution has been found. Some basic examples include a fitness score above a certain threshold, either from an individual or as the population average.

Pipeline The pipeline that GAs implement goes as follows:

1.– Generate a random population of chromosomes.

2.– Score each chromosome with the **objective function**.

3.– Evaluate the **stop condition**:

  3.1.– If condition is met, stop running and save the best chromosome as the solution.

  3.2.– If condition is not met, continue with step 4.

4.– Generate a new population by selecting breeders with the **selection method** and apply to them the desired **crossover method** to get new chromosomes.

5.– **Mutate** randomly some of the chromosomes in the new population.

6.– Add the best chromosomes from the previous generation into the new one, following the **elitism proportion** to avoid losing the solutions that had the best fitness.

7.– Repeat the procedure from step 2 with the new population.

The parameters and probabilities that are included in GAs control a variety of properties of the algorithm: the number of chromosomes in the population, the elitism proportion, multiple simultaneous stop conditions, and the probabilities to control the selection, crossover, and mutation.

3 Design

This chapter presents the analysis and design of the Ecosystem Evolution Simulator application. It elucidates the requirements of the application, together with the main use cases and the architecture the program follows. This chapter only covers what the application is intended to do. For an in-depth explanation of the importance and reasoning behind every subsystem and requirement, see Section 4.2.

3.1 Functional Requirements

The functional requirements that define the behavior of the application can be divided into the following subsystems. Each subsystem can be treated as an isolated block that is in charge of a key aspect of the functionality. While the specific functional requirements are presented in Appendix A, this section will only give a brief overview to have a better understanding of what the application is supposed to do.

Ecosystem Simulator This subsystem refers to all the code that allows the application to run simulations. This is the main subsystem of the application, and defines all the logic that rules over the ecosystem and the behavior and evolution of each entity.

Statistics Gathering and Generation This subsystem is strongly tied to the previous one, and is in charge of storing all the results of a simulation. With these results, one can re-run a simulation or analyze why the simulation evolved into a certain configuration. This subsystem contains all the code referring to data serialization and statistics generation.

Genetic Algorithm The GA subsystem contains all the code that is in charge of finding an appropriate configuration of the simulator in order to avoid the extinction of its species. It is important not to confuse the genetic algorithm with the evolution simulator. The simulator runs a single simulation with a given configuration. The genetic algorithm runs dozens of simulations to find an appropriate initial configuration. Check Section 4.2.3 for details.

Graphical User Interface This subsystem defines all the necessary windows to interact with the EES application, explaining in detail the exact fields and panels that will be present in each of them.

Starting with the *Ecosystem Simulator* subsystem, each entity type contains a set of genes that can define their capabilities, their behavior, their looks, and their energy cost. Although animals were elicited in the requirements, they were not included in this version of the application, so they will be considered in Chapter 6, regarding future development. With this, two types of plants are included in the application: bushes and trees. Plants will require soil and light to survive, gathered through their roots and their leaves, respectively. Their three basic genes will be height, trunk thickness, and crown size. Each of those genes affects how much energy they can gather and how much energy is used by the size of their trunk and their crown. Bushes will have fruits that will need to be dropped to reproduce, with each fruit having an additional energy cost. Thanks to another gene, trees will have the capability of sharing a percentage of their resources with adjacent neighbors connected via their roots. The simulator will be configured before running an experiment, where the user can choose the simulation speed, the value of a set of constants of the ecosystem, and the entities that will be present and their exact spawn conditions, both spatially and genetically.

Regarding the *Statistics Gathering and Generation* subsystem, the application will save a collection of snapshots, which define an instant of the simulator and can be used both to restore the state of the simulator and to gather statistics. How many (or how often) snapshots are captured in an experiment is decided by the user. The statistics calculated will represent three different types of statistics. The spatial positions of the entities will be presented through a collection of generated images called heat maps, in which each circle represents the position of an entity at a given instant. The evolution of the genes in the population will be presented with the average and standard deviation for each snapshot. The evolution of the properties of the entities (like energy usage, fruits remaining, generation number, etc.) will be presented in the same way as the evolution of the genes. The application will subdivide the statistics of each entity type into different species, depending on the evolution of different tendencies of genetic configurations in their populations.

Finally, the requirements for the *Genetic Algorithm* subsystem define all the settings that can be tweaked to configure an experiment. They also define what the codification of a chromosome is. The simulator has a subset of configurations used to define the normal distribution each gene will use to generate a random initial spawn. That is what a chromosome will encode, storing each required value in a list. Additionally, the definition of the objective function is also provided. It is defined as the number of days each species survived, and an additional score is provided when a species has successfully survived the whole experiment. The exact codification of chromosomes and maximum score of the objective function depend on the configuration of the experiment, as they are generated dynamically.

3.2 Non Functional Requirements

3.2.1 Documentation

NFR-1.– The application will include an offline user guide (Appendix E) that will explain in detail:

- **NFR-1.1.–** what the purpose of each window is.
- **NFR-1.2.–** the meaning of every configuration field in new simulations and GA experiments.
- **NFR-1.3.–** the action resulting from pressing every button.
- **NFR-1.4.–** additional keyboard controls that will not be visually present in the GUI, like the ones used in the simulation navigation mode.

NFR-2.– Along with the application, an overview video will be produced. This video will go through all the use cases defined in Section 3.3 [1].

3.2.2 Localization

NFR-3.– All the content produced in this application will be in English, including the application, the user guide, and the overview video.

NFR-4.– All the unit measures present in the application will follow the International System of Units.

NFR-5.– All the dates present in the application will follow the format DD/MM/YYYY.

3.2.3 Performance

NFR-6.– The recommended specifications to run the application will be:

- **OS** Windows 10
- **CPU** AMD Ryzen 9 3900X 3.8GHz
- **RAM** 16GB DDR4 3200MHz
- **GPU** NVIDIA GeForce RTX 2080 8GB DDR6
- **Disk** M.2 SSD PCIe NVMe

NFR-7.– The minimum specifications to run the application will be:

- **OS** Windows 10
- **CPU** AMD Ryzen 7 2700 3.2GHz (or equivalent)
- **RAM** 8GB DDR4 2666MHz
- **GPU** NVIDIA GeForce GTX 1660 6GB GDDR5 (or equivalent)
- **Disk** HDD SATA3 7200rpm

NFR-8.– The application shall not freeze for more than 3 seconds at any given point in a system that meets the minimum specifications requirement. To achieve this goal, the design of the application will prioritize execution speed over memory usage.

NFR-9.– Simulations with up to $5,000$ entities of any type shall run at a speed of 5 Frames Per Second (FPS) or higher in a system that meets the minimum specifications requirement.

3.3 Use Cases

The application is designed around three main use cases, each of which has a strong relationship with the others. From the main menu, the user can follow any of them with the click of a button. This will guide the user through a set of windows to carry out experiments with the application.

3.3.1 Configuring and Running a New Simulation

The most basic interaction with the application is to execute a simulation. For this, the user will click on the *New Simulation* button in the main menu. This will lead them to the *New Simulation Configuration* window, from which the user will be able to tweak the settings that rule the evolution and define the ecosystem. From this window, the user may tweak all the configurations provided by each of the categories stated in FR-24.1. The user may first decide the duration of the experiment and how the data will be collected. After that, the user may choose an initial genetic configuration and the constants that will rule over the ecosystem and evolution. Finally, the user will decide which entities may intervene in the simulation and their perks and limitations, with the additional possibility of choosing where those entities may spawn.

Once a new simulation has been configured, the user may continue to the *Simulation Execution* window, in which they have relevant information about the progress of the simulation. Additionally, the user has access to information about how many entities are alive and where they are for each entity type. If the user wants to take a look at the scene, they may pause the simulator and access the *Simulation Navigation* window via the *Navigate Simulation* button. Finally, either if the simulation has finished executing or if the user desires so, they may click the *Save & Exit* button to dump all the saved snapshots to disk and generate the statistics for further analysis, which leads to the second use case.

3.3.2 Analyzing the Results of a Simulation

After running one or more simulations, the user may want to analyze the results to understand what happened in the experiments and why. For this, the user is granted a set of analysis tools to ease the process. First, they may access the *Load Simulation File* window via the *Saved Simulations* button in the main menu. In this window, the user may recognize their experiments by the name they were given, the date and hour of the simulation, the entity types involved in the experiment, the duration of the experiment in simulation time, and the number of snapshots taken.

Once a saved file is selected, the user will have access to the *Simulation Results* window, in which they may compare the evolution of entity types one to one. The gathered data from the entities provides the user with an answer to three basic questions: where were the entities spatially, which was their genetic configuration, and what was their state (in terms of energy usage, available resources, etc.)

The user can use this information to analyze the relationships that different entity types have had with one another. This may provide interesting results about how different entity types show behaviors that also appear in nature, like mutualism, parasitism, or symbiosis. Other types of relationships may appear when analyzing interspecies of the same entity type. One example is the emerging behavior resulting from the trees' sharing gene, defined by FR-10.1. Additionally, if the user desires so, they may take a look at the scenery of the simulation at any given snapshot via the *Navigate Simulation* button, which leads to the *Simulation Navigation* window, in which they can inspect the genetic configuration and properties of each entity individually. The user may have a better understanding of the state of the simulator at a given snapshot with a glance at the scene, as every gene of every entity type has a visual repercussion. In Chapter 5 there is an example of a brief study on some of these relationships tested during the development of the application.

3.3.3 Finding a Balanced Ecosystem Through the Genetic Algorithm

During the analysis carried out by the user in the second use case, they might find some species going extinct. In this case, the user has two options: return to the first use case to fine-tune the simulation in an attempt to resolve the extinction issue, or rely on the Genetic Algorithm to find such a balance automatically. The user will access the *GA Configuration* window through a button in the main menu. From this window, the user can configure the experiment as they did in the first use case, but this time leaving the initial genetic configuration and spawn maps to the GA. The user may also decide how the GA will operate, tweaking all the configurations provided by the panel defined in FR-29.2. Once the user is done, they will access the *GA Execution* window through the *Start Training* button.

During the experiment with the GA, the user will be informed about the progress of the evaluation of the current chromosome and the average configuration of the whole population via a plot showing the mean and standard deviation of each gene. Finally, either if the experiment is finished due to a stop condition or if the user desires so, they may exit this window and save the best result found by clicking the *Save Best & Quit* button. This takes the user back to the *Main Menu* window, where they may want to run new simulations, again following the first use case, but this time with the obtained results in the GA. The user will be able to run a simulation with the exact same configuration as the previous experiment thanks to FR-22.2.

3.4 Architecture

This section covers the class structure that the application follows. The class diagrams are presented in Appendix B. Note that both the class diagrams and explanations are simplifications of the actual structure of the application, making them much more clear and easy to follow.

3.4.1 Simulator

The class diagram for the *Ecosystem Simulator* subsystem is presented in Figure B.2. It has been compacted to fit on a single page. Let's break it down to understand the structure. At the top center of the figure, we have the main class *Simulator*, which is in charge of controlling the whole system and sending instructions to the rest of the classes. It also holds references to all the data that is present in a simulation. Note that the simulator implements the Singleton pattern so that only one instance of the class may exist at the same time and every other class can have access to it. The *Simulator* class also holds references to other helper classes that will be covered shortly. It also relies on a dictionary that associates each entity type with a helper structure called *EntityDataAssociation*.
This structure holds references to several classes and data specific to each entity type. Apart from the 3D object and other control variables, the structure keeps references to three other objects:

Entity Spawner It is in charge of finding random positions in the world following a spawn map generated by the *PerlinTextureNoise2DGenerator* class.

Crossover Methods A class that holds all the reproduction methods in terms of gene inheritance, as defined by FR-6.1.1. It also contains a public method that invokes the delegate crossover method with two entities and generates a child.

Entity Manager This class is responsible for holding a pool of entities of a specific type. It is also used to generate random populations, to invoke update methods over said populations, and to control breeding following its offspring configuration.

Once we understand how entity managers work, we can examine the entity type tree. The first thing to notice is that the base *Entity* class implements the interface *IEvolutionable*, which grants an entity a list of genes, a reference to its entity manager, and a method to mutate its genes. This class also holds common data required for every entity type, such as their age, generation number, and energy usage. Additionally, this class defines the basic update methods used by the entity managers to control the population. The *AliveUpdate* method controls the behavior of the entity, and the *RecalculateRequirements* method evaluates its survival conditions.

As we descend the class tree, we come across two abstract classes, one for animals and one for plants. As animals were not included in this version of the simulator, let's focus on the *Plant* class. In this class, we have attributes to track the state of the plant in terms of its requirements, and properties to get the root and leaves radii, as those depend not only on the genes but also on the age of the plant. There is a method called *BalanceGenes*, which is in charge of avoiding plants that do not follow a somewhat realistic genetic configuration, as stated in FR-8.6. Another method called *CalculateReproduction* is used to decide when a plant should drop a seed and reproduce, as they are asexual and no crossover method is involved in the process. All plants include in their list of genes their thickness, height, and leafiness. Below the class tree we find the specific classes for plants: bushes and trees.

These classes override the update methods to define their behavior, survival conditions, and reproduction. The *BushEntity* class has some logic linked to the fruit genes, while the *TreeEntity* class holds some data regarding its specific genes. Every gene class holds its current value and has a reference to a static configuration used by every instance of its type. With this, genes can be mutated and randomized following these static configurations. A helper manager class is used to control and reference every *GeneConfig* instance. This will help the GA later on when it requires switching these configurations for its operation.

Finally, another helper singleton class called *PlantManager* is used for the plants to calculate their requirements. This is necessary because we need to compare each plant with the rest to know how much light and soil they have access to. This class is accessed by the *Simulator* class prior to an update of the simulation to set the energy available for every plant. This class implements an advanced technique of computing that takes advantage of the GPU to obtain its results faster. This topic is covered in detail in Section 4.2.5 about performance optimizations.

3.4.2 Data Serialization and Statistics Retrieval

The classes that integrate the *Statistics Gathering and Generation* subsystem are presented in Figure B.3. This figure is composed of two halves: the top half is related to the serialization of a simulation, and the bottom half is the serialization of constants, configurations, and statistics generated. At the top-left corner, there are three interfaces. In Figure B.2 we can see four classes that implement the *IMementoOriginator* interface, which means they are capable of loading and saving its state. Each of those classes has a respective class in Figure B.3 that implements the *IMemento* interface, which exposes which exact fields are necessary to define their save state. If we now focus on the top-right corner of Figure B.3, we can follow the whole tree of mementos to see how an instant of a simulation is represented, from a simulation memento to the entity manager mementos to the entity mementos along with their gene mementos. The *EntityMemento* class implements the *IDataGatherable* interface, which grants it a static property used to define which genes and properties can be gathered from the class. This is used later to collect all the statistics generated for an entity type in an automatic manner.

At the bottom half of the figure, there is the hierarchy of configurations used in a simulation, complying with FR-19. These configuration memento classes all implement two methods to automatically load and save the configuration of the simulator. The *SimulationData* class is serializable and stores all these configurations, plus the statistics generated for each entity type, when a simulation is saved to files. This is done via the *EntityStats* structure, which is used to track the heat maps, the entity count, the species, and the plot data. This class contains a member called *gatheredData* which is a Dictionary<K,T>, where K is a *GatherableDatum* and T is a list of plot lines, one for each species of the entity type. With all this, a simulation save file is composed of a collection of *SimulationMemento*s, a serialized *SimulationData* instance, and the heat map images.

3.4.3 Genetic Algorithm

Figure B.1 shows the class diagram that the *Genetic Algorithm* subsystem follows. At the top left corner of the image there is the *GAManager* class, which is in charge of running the algorithm, storing a history of the shape and progress of the population of chromosomes, and saving the best result found. This class has a reference to its active population. The *Population* class is used to invoke methods over the whole population, like evaluating the fitness of each chromosome, crossing over the population, or mutating it. Additionally, this class provides a constructor that generates a totally random population. The *Chromosome* class has some utility methods like mutate, calculate fitness, and apply its configuration to the simulator. This class relies on the method *EvaluateChromosome* to run a simulation in order to calculate its fitness. Chromosomes save their genome as a list of values. To identify which configuration corresponds to which value, the class has a static list that indicates the *GAGeneType* of each index. This class identifies a value in the list with the name of the entity gene and the type of value it encodes, from the *GAGeneValueType* enum.

The implementation of crossover methods follows the same pattern as in the *Ecosystem Simulator* subsystem. The best result found in an experiment with the genetic algorithm is saved into two separate files: *GAResultStats* and *GABestChromosomeData*. The reason why the data is split into two files is because the first class is used to identify what result is stored in the second file, and the second class stores all the configurations of the simulator. This prevents the *GUI* subsystem from freezing, as unnecessary data is not loaded until it is actually required.

4

Development

This chapter covers the topic of the development of the application. Firstly, there is a brief summary of what the development process consisted of. After that, there is an in-depth look at specific important implementation decisions. Along with the implementation comes an explanation of the optimization process for the simulator. Finally, a small section explains the testing procedure that was followed.

4.1 Methodology and Workflow

The development of the application followed an agile methodology, in which each iteration increased the functionality of the application in small bits. The basic workflow consisted of: analysis of the problem, design of possible solutions, selection and coding of the optimal solution, testing the implementation, and finally integrating the result into the application.

The tasks were structured as a tree of requirements, in which the upper levels had more priority due to functional or structural dependencies. This resulted in a specific ordering of the subsystems mentioned in Section 3.1, giving a certain priority to each requirement. The *Ecosystem Simulator* subsystem and the *Performance Optimization* requirement were considered first, as they were strongly dependent on one another, although the later one would require more attention in following iterations. Once a basic version of the simulator was implemented, the GUI subsystem escalated in priority and took its place, as configuring a simulation for testing purposes was starting to get convoluted. In this phase, special attention was paid to which settings were to be configurable by the user and which ones would remain constant and unknown to them. After that, the *Statistics Gathering and Generation* subsystem was implemented to make it possible to analyze the results of a simulation. The *Genetic Algorithm* subsystem was implemented last, as all the previous functionalities were required to evaluate its correctness.

The implementation process for a task started by breaking it apart into smaller ones that could be approached in a couple of hours each. The leaves for these trees of tasks were implemented first so that they would be merged bottom-to-top later on. There were several iterations over every subsystem, as every one of them had an impact on the rest. The final iterations were highly focused on improving

the performance of the application and bug-fixing, besides other minor quality-of-life improvements to the usability of the application, like refactoring the GUI or designing additional interactions to get the most from the application.

4.2 Implementation

In this section there is an in-depth look at some of the decisions that arose in the development of the application. These decisions answer the questions of why some requirements are important, why the solutions selected are appropriate, and how they are implemented. The content of this section is divided into the four subsystems defined in Section 3.1. An additional subsection is focused on performance, whose purpose is to comply with NFR-9.

4.2.1 Ecosystem Simulator

The most interesting algorithms implemented in the simulator arise from the behaviors entities exhibit as a consequence of a gene. As the only entities implemented in this version of the application are bushes and trees, there is a detailed explanation of how their survival conditions were coded.

Spawn Maps

Before any entity behavior is coded, we need entities to be instantiated at the beginning of a simulation. To do so, spawn maps are implemented. A spawn map is a texture image that represents a top-down view of the world. The brightness of each pixel determines the likelihood that an entity will spawn in that position.

The world could be populated with entities in entirely random positions, but there is a problem with that. The only way a user can interact with the simulator is by setting the initial conditions. Giving the user more control over this initial configuration provides a margin for designing experiments with more interesting results. Taking advantage of Perlin Noise, we can generate spawn maps with clusters of a specific size, shape, position, and density. A sample of a spawn map generated with Perlin noise is shown in Figure 4.1.

Unity provides an implementation of Perlin Noise in which, given a pair of coordinates (x, y), we get a value in the range $[0, 1]$ that will be used to define the brightness of the pixel. This value is processed via a parameterized sigmoid function

$$\sigma(x) = \frac{(Max - Min)}{1 + e^{-(x-T)D}} + Min$$

$$pixel_{xy} \leftarrow Clamp(\sigma(Unity.PerlinNoise(x, y)), 0, 1)$$

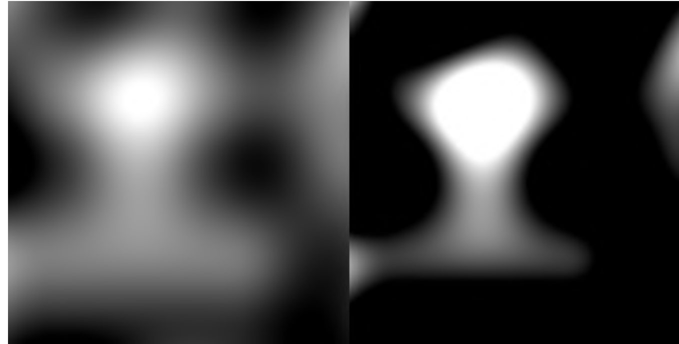


Figure 4.1: The image on the left shows a random sample of Perlin Noise. The image on the right shows the same random sample but transformed with the parameterized sigmoid function.

where D is the dispersion, T is the threshold, and Max and Min define the convergence of the sigmoid. With this transformation we can control the properties of the spawn maps, by altering their maximum and minimum values, and the density and scale of the clusters.

To use a spawn map, the roulette wheel algorithm implementation shown in Algorithm 4.1 is used, considering each pixel of the texture as an element of the input list.

```
Input   : A list L of size N containing tuples (value, weight)
Output: The index of a randomly selected element.
1  //L_i[1] corresponds to the weigth of the i-th element in the list
2  weightsSum ← Σ_{i=0}^{N−1} L_i[1];
3  weightIdx ← Random(0, weightsSum);
4  accWeight ← 0;
5  for listIdx ← 0 to N − 1 do
6      accWeight ← accWeight + L_listIdx[1];
7      if accWeight >= weightIdx then
8          break;
9      end
10 end
11 return listIdx;
```

Algorithm 4.1: The roulette wheel algorithm takes a list of values with weights, and chooses one at random, being more likely to be selected the elements with greater weights. This algorithm is also known as the Fitness Proportionate Selection, used as a selection method in genetic algorithms.

Prior to running the algorithm, we need to first generate the list of tuples containing the coordinates of each position on the map and the spawn probability value. In this step, we make sure to discard all spawn positions that are not valid, like those that are out of bounds or inside the lake that is in the middle of the map. The coordinates whose spawn probability value is zero are also discarded. The next step is to run the roulette wheel algorithm as many times as there are entities to spawn.

Some minor details have been taken into account when spawning entities. To avoid several entities spawning in the same position, during the roulette wheel algorithm, the positions that are selected are discarded from the input list. Furthermore, a grid-like pattern in the spawn is avoided by adding some small random variation in both coordinates when the tuples are generated. In order to prevent entities from overlapping with one another, the entity spawner defines a variable that states the minimum distance between spawns.

Plants Soil Calculations

One of the basic requirements for plants is to gather nutrients through their roots, which depend on the soil available where the plant is located. To calculate how much soil a plant has access to, the world has a constant soil map generated with Perlin Noise. This map was generated with the same tool explained in Section 4.2.1 regarding spawn maps. The amount of soil available to a plant is equal to the sum of the values of all the pixels in the soil map within the plant's radius. If a pixel is shared between several plants, its soil is evenly distributed amongst all of them.

This algorithm has to be run for each step of the simulation; therefore, it is important to make it as efficient as possible. The first step is to generate a new texture that, for each pixel, gets the value of the respective pixel in the soil map texture and averages it by how many trees have access to it. After that, each plant simply has to add up all the values that are in its root radius in the helper texture. The value of each pixel can only be in the range $[0, 1]$, so a constant factor is used to scale the value. The name of the simulation constant is 'MaximumSoilPerSquareUnit", which is later used to calculate the maximum soil per pixel. An example of the helper texture is present in Figure D.1(b).

Plants Light Calculations

Plants also need light to survive. The light a plant has access to is defined by FR-8.3.2. The first approach that comes to mind is to calculate the difference between the area of each plant and all the plants that are higher than the first one, but the complexity of calculating the difference between a circle and a cluster of other circles is too expensive. This algorithm is also run for every step of the simulation, so the current implementation takes advantage of another texture and discretizes the area of the circles. There is a trade-off between accuracy and performance.

The implemented algorithm sorts all the plants by their map height in descending order. The map height of a plant depends on its height gene, its age, and its 'y' coordinate in the map. Then, for each plant, a circle is painted in a texture with radius equal to their leaves gene, as shown in Figure D.1(c). For every pixel painted, the plant gets a proportional amount of light. If a pixel has already been painted, then it is not accounted. The value provided by a single pixel is scaled with another configurable constant in the simulator, called 'LightPerSquareUnit', in the same way it is done with soil calculations.

Trees Sharing Gene

The first idea for this gene was to connect all trees by their roots, and then they would share a percentage of their available resources through such connections, encouraging the possibility of smaller trees growing underneath big trees. To achieve this goal, a Connected-Component Labeling (CCL) algorithm [7] was implemented to calculate which trees were interconnected. An extra effort was put into optimizing this algorithm. However, in large simulations, this resulted in the interconnection of all trees into one single cluster, and trees with a sharing percentage of 0 dominated the population, which defeated the purpose of running this algorithm in the first place.

The utility of this gene has been altered to only consider direct neighbors, which helps with the problem mentioned before. This change means that the CCL algorithm is no longer used, so the logic has been reworked in the following way. Each tree calculates how many neighbors it has, and then evenly distributes the percentage of the resources it shares. The value of the gene determines what percentage of the tree's resources it shares with its neighbors. To find the neighbors of each tree, instead of comparing the distances of every pair of trees, which would have a cost of $O(N^2)$, a spatial dictionary is generated and updated when a tree is born or dies. The spatial dictionary *Dictionary(K,V)* is built at the initialization of the *PlantManager* class, being *K* a pair of integer coordinates, and *V* a list of every tree whose coordinates' floor are equal to *K*. This way each tree can simply read which trees are within its roots range, highly simplifying the cost of the algorithm to $O(N)$.

4.2.2 Statistics Gathering and Generation

The generation of statistics is an expensive operation, and some of those statistics require data from the whole simulation at once to be calculated. This is why the statistics are gathered and generated all at once at the end of a simulation, when all the snapshots have been saved. Saving the state of the simulator at each step would make save files occupy a lot of memory. This is why the EES provides control over how many (or how often) snapshots are taken during the simulation. These snapshots represent a trade-off between the accuracy of the statistics, and the memory usage and time spent calculating statistics. This section goes through which data defines a save file for a simulation and how it is serialized. After that, there is an explanation of how each statistic is generated.

Data Serialization

A saved snapshot must be able to represent a simulator state in a specific time step, so that a non-random operation in the simulator and the saved snapshot produce the same result. This is important to point out because we can save some memory in the save files by deciding which data is required to meet this condition. There are two cases in which data is not required to define a simulator state. In the first case, some data can be obtained by performing an operation on other actual critical data.

This is the case for entity properties. As an example, plants store their height as a property, but this value can be obtained via the plant's age and the value of its height gene. In the second case, some data is not required at all because it plays no role in the evolution or the ecosystem. This is the case for the exact coordinates of the fruits a bush has. These positions are not accounted for in any operation other than rendering the fruits on the screen. In this example, only the number of fruits is required to be stored in memory.

The approach to defining a snapshot of a simulation is via the Memento design pattern [8]. This design pattern provides the capability for a class to save and load its state without exposing its fields to other classes. Another reason why this design pattern is useful is because we can decide which fields of the class need to be stored, which accomplishes the save file size constraint explained previously. The Memento design pattern defines two key elements: the *Memento* class and the *Originator* class. This design pattern has been modified in the application by replacing the *Memento* class with a generic *IMemento* interface, so that any desired class can define its own memento class. Furthermore, only a class can be the originator of its own memento type, reassuring encapsulation. This is achieved via the following redefinition for the *Originator* class:

public interface IMementoOriginator<T,M> where T:IMementoOriginator<T,M> where M:IMemento<T>

Statistics Gathering

To better understand what happened during a simulation and why entities evolved the way they did, a form of visual representation of the data gathered is required. The basic questions that arise are how many entities there are, where they are, what their genetic configuration is, and what their properties are. To answer all these questions, the statistics stated in FR-20 are gathered and/or generated. All those statistics are fairly straightforward to compute, calculating the mean and standard deviation of each type of gene and property, and painting a texture with the position of the entities in each snapshot.

However, a particularly complex data analysis algorithm has been developed to accomplish FR-20.1, about species detection. In some simulations, we might find that some gene statistics show a really high standard deviation. This probably means that within a population of a specific entity type, there are different species that survived. An example might be a simulation in which there is one species of tree with a thin and short trunk and another species with a thick and tall one. This leads to a plot of the statistics in which the mean value of the genes is the average of both species and has a high standard deviation. This is why FR-20.1 is elicited, and a data analysis algorithm has been developed to identify these species.

The data that can be used to discern between species is composed of the entities' genes, their position in the world, and the instant in time that they live in. This problem could be treated as a spatio-temporal evolving clusters algorithm, but the spatial information has been discarded.

This means that if two species evolve into the same genetic configuration, they can be treated as the same even when they are spatially distant from one another. There is no ground truth about which entities belong to which species. This means that an unsupervised data analysis algorithm for clustering is required. There isn't much research in temporal evolving clusters' algorithms, but there is high potential in the field. Some articles take a look into this field by researching how bacteria evolve [9], or by proposing models to understand embryonic development [10]. These articles analyze how the clusters evolve in the context of their specific problems. In the case of species recognition in the simulator, we can take a similar approach.

The exploration on evolving clusters presented in Appendix C allows us to define the rules of an algorithm whose result can be visualized as a forest of trees of species, as shown in Figure 4.2. The model proposed to solve the species detection problem in the simulator has three well-differentiated steps. The first step analyzes each simulation snapshot individually, running a clustering algorithm. The second step builds the species tree by linking the clusters of each snapshot to the ones found in the previous one. A specific set of rules is implemented to decide if a cluster detected is the direct successor of a previous one, if it is a subspecies, or if it is a new species. The final step is to assign species identifiers to each entity in each snapshot. This last step allows the generation of statistics for each species found separately.

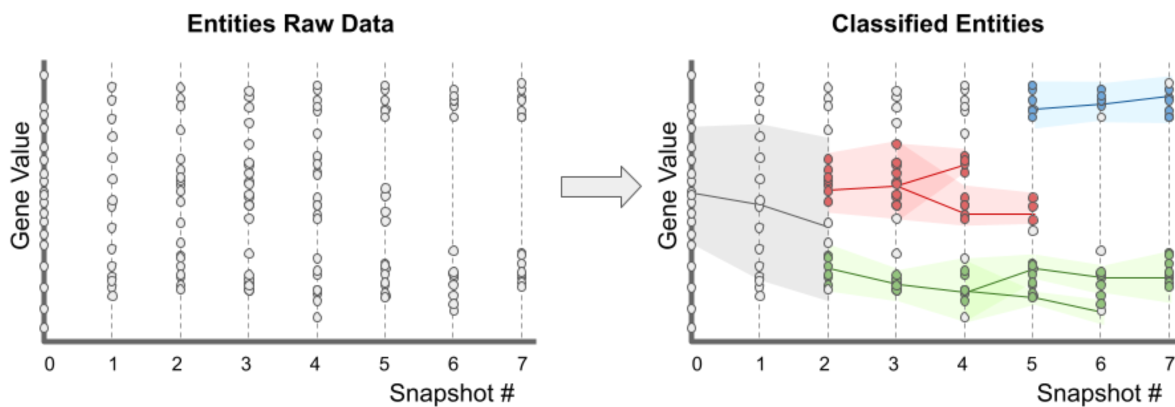


Figure 4.2: Each circle in these plots represents the gene value of an entity in a given snapshot. The first plot shows the raw data that is gathered during a simulation, and the second plot shows the expected result for the species detection algorithm. Lines represent the mean gene value of a species or subspecies, while the shaded areas show the standard deviation. Gray results are shown when no species are found in a snapshot, and represent the statistics of the whole population.

An important decision is what clustering algorithm to use at the first step of the model. The centroid-based clustering algorithm KMeans [11] is not ideal for this problem. One of the most important drawbacks is the need to choose a value for its hyper-parameter K automatically, which is not trivial. There are some techniques that allow this automated search. With the elbow method [12] we can score how good a K value is by comparing the intra-cluster and inter-cluster distances. This means that KMeans has to be run several times with different K values, which is unacceptable, as this process has to be

carried out for every snapshot, and it would take too long to compute. Additionally, KMeans poses some other disadvantages, like finding clusters with varying densities or with non-hyper-spherical shapes [13]. Another category of clustering algorithms is hierarchical clustering, which is commonly used in real life categorization of species. This type of clustering builds a dendogram, a special kind of binary tree, in which each level determines the best clustering of the data with a given distance function.
However, to decide the final clustering it also requires finding a cutting point K in the dendogram, so in its place, the implementation in the application uses the Hierarchical Density-Based Spatial Clustering of Applications with Noise (HDBSCAN) algorithm [14]. This clustering algorithm is a density-based approach to clustering, and solves the problem of having a K hyper-parameter. HDBSCAN has two hyper-parameters: minimum points and minimum cluster size, but with some experimentation with the data generated in the simulator, they were fixed to a constant value that grant a proper result. The simulator benefits from the relatively fast execution time this algorithm has and obtains the centroids of the clusters of entities for every snapshot.

The second step of the species detection model builds the species tree and creates an association table between the clusters of each snapshot and the species they belong to. The species tree is built from left to right. HDBSCAN may not find any clusters in a snapshot if the data it contains is too scattered. This is pretty common at the beginning of a simulation, which means that a few of the first snapshots in the simulation will not have clusters. The first clusters that are found are considered the root of each species' tree. Once there is at least one species, the centroids of the following snapshots are linked to their closest previous centroid. However, if a new centroid is too far from any previous centroid, it is considered a new species. To define this distance, the model counts on the new 'Maximum Child Distance' hyper-parameter. This hyper-parameter takes advantage of the characteristic of the solution space having a finite n-dimensional volume. This property can be used to encode distances as a percentage of the largest distance possible in the solution space. The consequence of this is clearly pictured in Figure 4.3, where clusters 'A4' and 'A5' are not linked, as there is a big leap in the centroids' values, thus giving birth to a new species.
To identify species and subspecies, they are given names in the shape of versions. New species, like the ones found in snapshots 2 and 5 in Figure 4.3, are named with X.0, being X the number of root species found. The rest of species versions follow the format X.C.C.C..., where X is the root version they belong to, and C determines which child number they are of their parent version. For example, species 2.3 means that this species is the third child of species 2.0.

An important detail to point out is that the example shown in Figures 4.2 and 4.3 is one-dimensional, with a single gene value. This algorithm is executed with n-dimensional data, where n is the number of genes the entity type analyzed has. This can lead to a situation in which two different species have a similar value in a specific gene but are considered different because they differ in other genes. This can make the representation of data a bit convoluted, as some species trees may overlap in the plots, which complicates the analysis of a simulation.

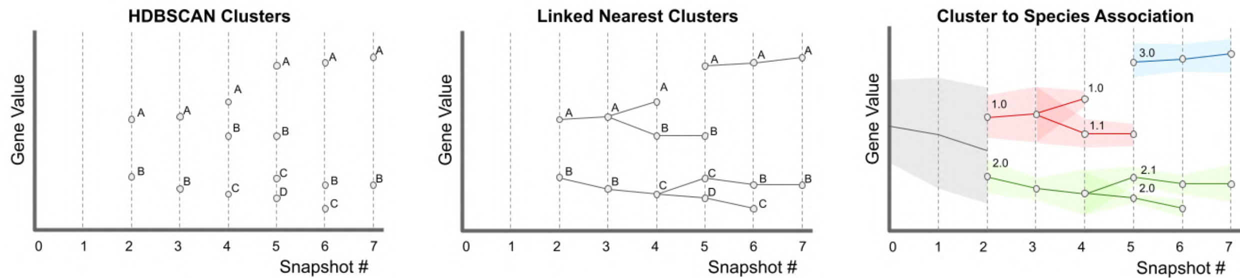


Figure 4.3: This figure shows the steps carried out to solve the species detection problem. In particular, it shows the creation of the species tree and the association between the HDBSCAN's clusters and the species versions they belong to. Clusters are named after the tag they have and the snapshot they sit in, e.g., 'B2' or 'C5'.

This approach is not perfect, and there is still room for improvement. A lot of noise is generated in the simulation due to mutation, which can worsen the results obtained in the clustering step. Furthermore, HDBSCAN has some difficulties detecting clusters with varying densities. Some experimentation can be made with distribution-based clustering algorithms, such as the Gaussian Mixture Model [15], as evolution tends to generate data that follows normal distributions. Additionally, the inclusion of the discarded spatial information of the entities can be used to test how the clustering behaves, and if it helps obtaining more consistent clusters. One of the key actions that can improve the results is running a Principal Component Analysis (PCA) to detect unimportant genes and discarding them from the data presented to the clustering algorithm. Some experiments with bushes made this obvious, as the fruit genes were designed to interact with animals, which are not present in this version of the application. This means that these genes currently have little repercussion on their survival function, hence they are disrupting the algorithm.

4.2.3 Genetic Algorithm

While running simulations, we can find situations in which some species go extinct. This might be due to one species dominating the whole area, making survival for other species impossible. The explanation for this situation can be easily pictured with the prey-predator model's imbalance problem. When the prey-predator model is balanced, their population sizes cycle over time, with the predator's population size lagging behind the prey's [16]. Figure 4.4 depicts this cyclical behavior.

This relationship is delicate, and altering it in any way can lead to both species' extinction. Real-life environments show this same problem, but thousands of different species are involved, and a small disruption of its equilibrium may have fatal consequences because this issue cascades over several relationships of species. A wildfire, a harsh winter, or even a one-degree change in sea temperature can disrupt the delicate equilibrium that evolution established in the first place. Species dying or migrating to areas where they should not be endanger their ecosystems.

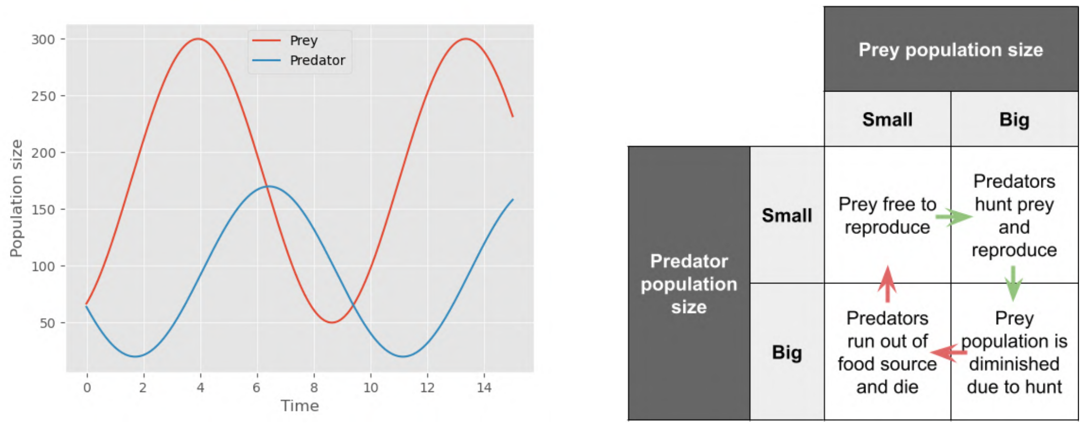


Figure 4.4: The plot on the left shows how the populations of both prey and predators evolve over time, using a sine function for simplicity. The table on the right explains the cyclic behavior of the prey-predator model.

Additionally, finding such a balance in the first place is not a negligible task. An example of the prey-predator model's imbalance problem is represented in Figure 4.5, which takes rabbits and wolves as prey and predators, respectively. The rabbit's extinction begins at the first time mark as a result of the wolf population becoming too large. Wolves extinguish themselves at the second time mark because they run out of food sources.

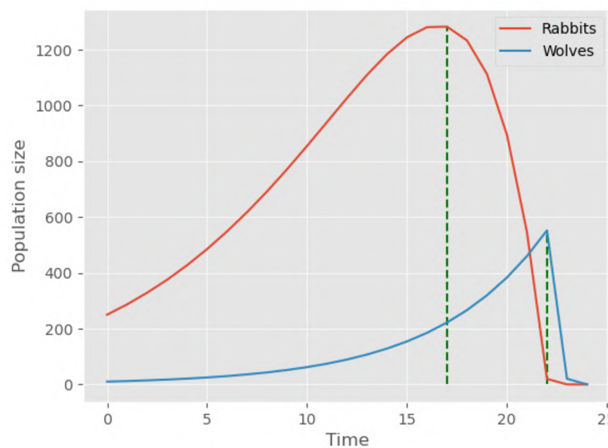


Figure 4.5: This figure represents a possible imbalance between the population sizes of rabbits and wolves over time, in which both species go extinct.

The solution proposed to this situation in the EES is to find an appropriate initial configuration of the entities, which will grant a lower chance of extinction of species in the experiment. To achieve this, a Genetic Algorithm [17] is used.

Codification

Firstly, here are some definitions to avoid confusion with the terminology, as we are mixing together an evolution simulator with a genetic evolution algorithm. The GA benefits from the evolution of a **population**, which contains **chromosomes**. A chromosome encodes a subset of initial settings for the simulator, which are called **genetic configuration**. The complementary subset of initial settings for the simulator is the **simulator configuration**. The GA is used to find a suitable genetic configuration given a simulator configuration to provide equilibrium between the entities, hence solving the prey-predator model's imbalance problem.

A genetic configuration contains a list of each **gene configuration** involved in an experiment. Which gene configurations are selected depend on what entity types are selected. A gene configuration has a mean and a standard deviation, which are used at the beginning of the simulation to spawn the random entities, and a mutation intensity, which dictates how strongly the gene can mutate during a simulation. These value types have hard-coded limits, and the chromosome is in charge of keeping them in their respective ranges. Each value type has a different minimum and maximum value.

Mean The value limits for the mean data type are directly related to the value limits of the gene it belongs to $[gene_{min}, gene_{max}]$. These values are hard-coded into the simulator to comply with FR-6.2. For example, the *bush height* gene's range is set to $[0.35, 2]$.

Standard Deviation The possible values for the standard deviation data type are in the range $[0, \frac{gene_{max} - gene_{min}}{2}]$. This maximum value leads to a standard deviation in which there is a very high chance that any random sample is at a maximum distance of half the range of values the gene can take.

Mutation Intensity The possible values for the mutation intensity data type are in the range $[\epsilon, 1]$, where ϵ is a small value greater than zero defined in the simulator constants to avoid an immutable gene.

A chromosome is defined as a genetic configuration, a list of gene configurations; therefore, value encoding is the appropriate choice. As each gene configuration has three well-differentiated value types, they cannot be mixed, so all operations have to be made element-wise. The codification of a chromosome is dynamic, as the gene configurations contained in the genetic configuration depend on which entity types are selected to be tested in the simulation. This simplifies the problem when not all the entities are selected, resulting in faster executions for less complex simulations. An example of codification is shown in Figure 4.6.

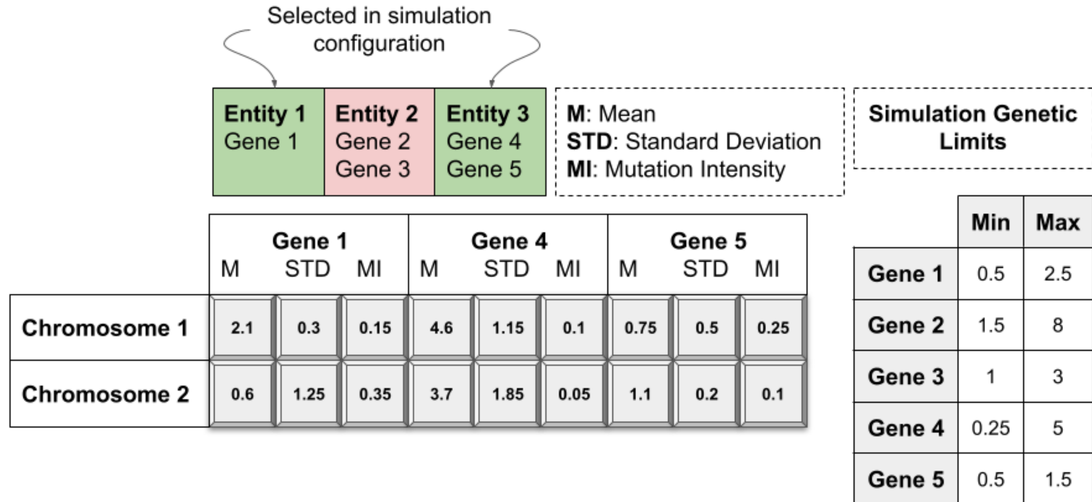


Figure 4.6: This figure represents the codification of two chromosomes, given a sample simulation configuration. Three sample entities are defined, but as entity 2 is not selected, its genes (2 and 3) are not present on the chromosome.

Objective Function

The objective function scores how good a chromosome is, where a high score is better for the survival and reproduction of chromosomes. To calculate the fitness of a chromosome, a simulation is run with a given simulation configuration and the genetic configuration proposed by the chromosome. The fitness score is measured by how many species survived the simulation and for how many days. This means that the maximum fitness score varies depending on which entities are selected in the simulation configuration and how many days the simulation lasts. The fitness function is calculated as

$$f_0 = \left(W \times D \times \sum_{e}^{Entity\ Types} Alive(e, \frac{D}{S}) \right) + \left(\sum_{e}^{Entity\ Types} \sum_{f=1}^{\frac{D}{S}} Alive(e, f) \times S \right)$$

$$Alive(e, f) = \begin{cases} 1 & \text{if there are entities of type } e \text{ alive at frame } f \\ \\ 0 & \text{otherwise} \end{cases}$$

where D is the total number of days the simulation runs, S is the simulation speed in days per frame, and W is a scaling factor that is used to give more weight to surviving the whole simulation rather than only surviving for a portion of the days.

Initial Population

An initial population of chromosomes is generated by giving each element a value in its restricted range, as explained in the codification subsection. This randomness at the first epoch of the algorithm provides a broad exploration of the solution space. In the following epochs, the mutation helps keep the algorithm from getting stuck at the local minimums of the objective function.

Selection

The selection method is the roulette wheel algorithm, which has the same implementation as explained in Algorithm 4.1. The chromosomes with a higher score in the objective function have a greater proportional chance of having descendants. Additionally, one chromosome can be selected several times. Due to the randomness involved in this operator, there is a chance some chromosomes with high scores will not make the cut. To avoid losing the best solutions, elitism is also implemented to keep them present in the population. The chromosomes saved by elitism are not affected by crossover or mutation.

Mutation

If a chromosome is randomly chosen to be mutated, each element of the chromosome has a probability of being mutated. If a value is mutated, a new value is obtained via a normal distribution $N(\mu, \sigma)$, where μ is the old value and σ is a standard deviation proportional to the range the value can have. The standard deviation is calculated as $\frac{val_{max} - val_{min}}{2} \times M$, where M stands for mutation intensity, a constant value defined in the genetic algorithm settings. A visual representation is shown in Figure 4.7.

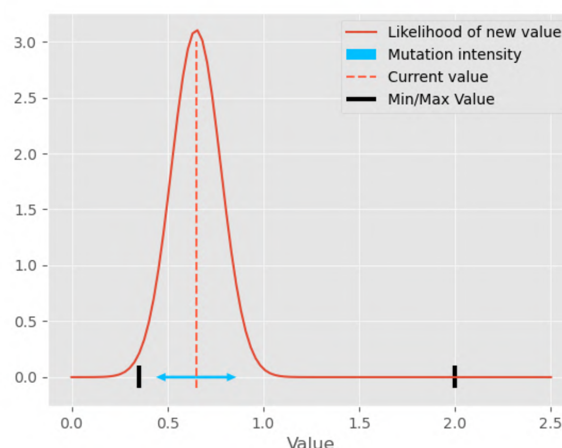


Figure 4.7: Representation of how a value of a chromosome is mutated in the GA.

Crossover

The crossover method to be used can be selected in the application's *GA Configuration* window. As explained in the codification, each element of the list of values in a chromosome has to be treated separately from the rest, as they pose some value restrictions. Luckily, the following crossover methods guarantee that values cannot exceed their imposed limits. A visual representation of the different crossover methods is shown in Figure 4.8 to better understand how these operators work.

Discrete A child is generated by copying each value from one of its parents at random. A second child is generated by inverting the selection of the first one.

Average Two equal children are generated by averaging the values of both parents.

Weighted Average For each element of the child chromosome, the average of both parents is calculated, weighted towards one of the parents by a random weight in the range $w_0 = Rand(0,1)$. The second child uses $w_1 = 1 - w_0$ as the weight factor.

One Point A child is generated by copying all the values from one parent up to a random point in the chain, and then the rest of the values are copied from the second parent. The second child is the inverse of the first one.

Parents

Parent 1	Chromosome 1	2.1	0.3	0.15	4.6	1.15	0.1	0.75	0.5	0.25
Parent 2	Chromosome 2	0.6	1.25	0.35	3.7	1.85	0.05	1.1	0.2	0.1

Discrete Crossover

Child 1	Chromosome 1	0.6	0.3	0.15	4.6	1.85	0.1	1.1	0.2	0.1
Child 2	Chromosome 2	2.1	1.25	0.35	3.7	1.15	0.05	0.75	0.5	0.25

Average Crossover

Child 1	Chromosome 1	1.35	0.78	0.25	4.15	1.5	0.08	0.93	0.35	0.18
Child 2	Chromosome 2	1.35	0.78	0.25	4.15	1.5	0.08	0.93	0.35	0.18

Weighted Average Crossover

	W	0.2	0.3	0.85	0.25	0.9	0.15	0.5	0	1
Child 1	Chromosome 1	0.9	0.97	0.18	3.93	1.22	0.06	0.93	0.2	0.25
Child 2	Chromosome 2	1.8	0.59	0.32	4.38	1.78	0.09	0.93	0.5	0.1

$w_i = rand(0,1)$
$c1_i = p1_i * w_i + p2 * (1-w_1)$
$c2_i = p2_i * w_i + p1 * (1-w_1)$

One Point Crossover

Cross Point (after the fifth value)

Child 1	Chromosome 1	2.1	0.3	0.15	4.6	1.15	0.05	1.1	0.2	0.1
Child 2	Chromosome 2	0.6	1.25	0.35	3.7	1.85	0.1	0.75	0.5	0.25

Figure 4.8: Visual representation of the result of crossover operations over two sample chromosomes.

Additional Settings

The EES provides some additional flexibility to the GA to decide how it will behave during a test run.

Population Size How many chromosomes conform a population.

Probabilities The probabilities for crossover, chromosome mutation, chromosome value's mutation, and elitism can be tweaked by the user, along with the chromosome values' mutation intensity.

Stop Condition The GA has several stop conditions implemented. The user can choose the maximum number of epochs, the objective maximum population score, and the objective mean population score. The algorithm will stop if any of those three conditions is met. If the user does not want to use the objective mean or maximum population score stop conditions, they can set those values to a number greater than 100%.

It is important to emphasize that the objective function is really expensive, and it is the responsibility of the user to configure the simulations to run as fast as possible. After some experimentation, a single simulation that lasts for 20 seconds has an acceptable duration. Slower than that might defeat the purpose of the genetic algorithm, and parameter tuning by hand might be a better solution to the problem.

4.2.4 Graphical User Interface

The implementation of the GUI subsystem is fairly straightforward, and deserves almost no attention. Appendix A.4 defines the details of each window's content and navigation, and Appendix E shows the final appearances of each window and their explanations.

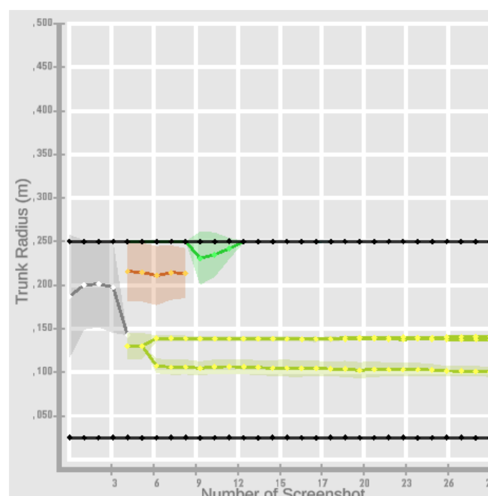


Figure 4.9: This figure shows an example plot generated with the *UIGridRenderer* component.

Anyhow, an interesting implementation detail arose in the *Simulation results* window. Unity does not provide any component to render grids and plot points and lines in the UI. Therefore, a new component called *UIGridRenderer* has been implemented to cover this functionality. This component required explicit coding for rendering regular polygons, lines, and quadrilaterals, stating the triangles and vertices that define those shapes. With these basic shapes, the component was used to render grids, axis ticks, and plot lines and dots. The component, shown in Figure 4.9, is highly configurable, defining the size of the grid, the scale, the number of ticks, and the colors and thickness of every element. The application chooses the best configuration for the component dynamically, depending on the selected data.

4.2.5 Performance Optimization

Why performance is important

One of the key aspects of the development of the EES was performance optimization. In the field of simulations, there are two main categories. The first is real-time simulation, in which a scene is rendered at the same time a simulation step is calculated. This means that these simulations should sustain a constant execution time, usually related to a frame rate of 30 or 60 FPS. The second category is non-real-time simulations, whose objective is retrieving the results as early as feasible. The focus of this type of simulation is on the results, not the process itself. That is why in the EES the scene is not rendered during a simulation, because it takes a significant amount of execution time, and we are interested in analyzing the evolution. Furthermore, there is another incentive to improve the execution time, and that is the usage of the Genetic Algorithm, where a simulation is run for every chromosome. This means that dozens of simulations are run per experiment, hence the need to make the simulations run even faster.

How to know what to optimize

It is important to keep in mind the need to design efficient solutions during the development process, but this is a double-edged sword. The effort to make code efficient is not always necessary, and can worsen the development process by wasting time on unnecessary optimizations and even by making some code less readable and sustainable. Unity provides a profiler tool that can be used to analyze several details of the application, like how much time a method takes to execute, how many times a method is executed, how much memory is used, garbage collection, and many other statistics. This tool helps understand where the bottlenecks lie in the code and which code segments require more attention, as a small optimization in code that is executed many times can have a great impact on the final performance. During the development of the EES, several inspections have been carried out with the profiler to seek out performance leaks and areas where optimizations might have a great impact.

Key optimizations in the Ecosystem Evolution Simulator

The most basic optimization for running simulations faster is to not show the user the progress, saving execution time in rendering graphics and calculating statistics. However, having a black screen for five minutes would not be appropriate for an application. Therefore, some basic statistics are calculated to show the user the progress of the experiment. This selective information is fast to compute and grants the user a sense of roughly knowing how the simulation is going.

The most expensive algorithms that run during the simulations are related to the plants' light and soil requirements calculations, as explained in Section 4.2.1. As both light and soil calculations require a pass through every plant, they can be performed in the same loop. This makes code less readable, but in exchange, we get a great boost in performance. However, the greatest boost in these algorithms comes from using Unity's compute shaders [18] to generate the textures with the GPU instead of the CPU. This is because hundreds of pixels can be painted in parallel. In the simulator, every pixel in the radius of a plant is painted at the same time. Another significant optimization is to use a single texture with three channels for all calculations in order to reduce the amount of time data is transferred between the CPU and GPU. This same texture is shown to the user as a piece of key information during a simulation, as Figure D.1 portrays. This is the greatest boost that has been obtained in the simulation, with a frame rate up to 37 times faster. Prior to this improvement, simulations with $2,000$ plants would run at around 0.5 FPS, but now a simulation can get up to $10,000$ plants and still keep a steady 5 FPS.

The final key optimization consisted of the implementation of entity pools. Each entity manager has a pool in which entities are created, enabled, disabled, and destroyed automatically. With this technique, Unity's garbage collector has less work to do, as enabling and disabling entities is much more efficient than creating and destroying an entity instance every time one is born or dies. This single optimization had a boost of roughly 2.5 times more FPS.

Aside from these key optimizations, other flaws in the system were detected thanks to Unity's profiler tool. The most important discovery was that retrieving references to singletone instances through their 'Class.Instance' attribute was too expensive. On its own, it is not that big of a deal, but if every entity accessed a singletone instance $1,000$ times in their lifetime, that results in a high cost, which is exactly what the profiler showed. Instead of that, every entity would obtain the singletone reference once and then store it in a private attribute. Other important flaw of the simulator were C# properties, which would make a calculation every time their value would be accessed. Precalculating this value in advance and storing the result also had an important impact on performance. One final memory leak was detected with the profiler, and that was creating the texture shown in Figure D.1 every frame instead of repainting it. This series of actions returned a performance boost of about 4 times faster execution time.

Other minor optimizations included creating manager classes to store constant data, like the class *GenesConfigManager*. This way, the thousands of instances of genes would not need to store these configurations themselves. Another improvement was removing the usage of library *Linq*. This library

provides a very useful stream syntax for iterators, but with some experimentation a great performance cost was found with its usage. Furthermore, some loops can be optimized doing several actions at the same time if implemented with for loops, like filtering and selecting results. Finally, avoiding using square roots when calculating distances. Calculating the square root of numbers is an expensive operation, and it is not necessary when we only need to compare if a distance is greater or smaller than other one. To do so, the comparison can be made with the squares of the distances.

With all this work, the simulator sits in a state in which running experiments does not get very tedious and simulations run at an acceptable speed. There is still room for improvement. To date, Unity is developing a new set of packages under the name DOTS, which has a high focus on instantiating thousands of entities and includes helper classes for parallelism and a burst compiler [19]. Pitifully, this is still in experimental versions, and is not a fully developed technology.
A special mention goes to the loading screen, which is designed to prevent the application from freezing when some data is being loaded or heavy computing, like the generation of statistics, is being carried out. This acknowledgement is made because, in order to prevent the application from freezing, the operations have to be stalled to return control to the application and paint the next frame, hence making the loading process slower.

4.3 Continuous Integration

In a system this complex, where each subsystem has an effect on the rest, testing the implementation in an organized manner is critical. Some functionality is shared amongst several systems, like the simulator being used for both normal experiments and genetic algorithm experiments. This means that fixing an error that appears in one system may create another bug in the second system. To mitigate this problem, a complete functionality check would be carried out each time a development iteration was finished.

Additionally, each individual task that could be isolated from its respective system was implemented in a specialized environment prepared for testing, used to check the correctness of the codification. This isolated environment made it easier to evaluate the correctness of the algorithms, and edge cases were easily controlled. Once this testing was finished, the algorithms were integrated into their respective system and tested again to see how they would react to the conditions of the new environment.

Unity provides an API to build your own editor windows. With this tool, the testing environment was created, giving a script the capability of executing methods even when the application is in edit mode. This was possible thanks to the capability of C# to create custom attributes. This attribute was used to mark methods as executable with buttons, and a small piece of code painted a button for every method it would find with this custom tag. The creation of this attribute also helped with other tasks, like generating spawn maps or the spawn positions of fruits from the editor with the click of a button.

5 Experiments and Results

The content of this chapter can be divided into two main sources of experiments: the simulator and the Genetic Algorithm. On the one hand, in the experiments with the simulator, there are several sources for analysis, starting with finding an appropriate value for the constants of the simulator, and continuing with analyzing the emergent properties and relationships with genes both for bushes and trees. On the other hand, the experiments with the GA will put to the test the efficacy of finding appropriate configurations of the simulator that avoid the extinction of species.

5.1 Experimenting with the Simulator

The first step once the simulator was functional was to balance the values of the constants that defined the behavior of the evolution in the ecosystem. These values were obtained by first defining arbitrary limits to the values of the genes (like the bush maximum height being 2 meters) and later tuning the constants to allow bushes to survive in low soil areas and trees to only survive in high soil areas. To run experiments faster, only the bottom left corner of the map is used, and Figure 5.1 can be used to understand the distribution of soil in said corner.

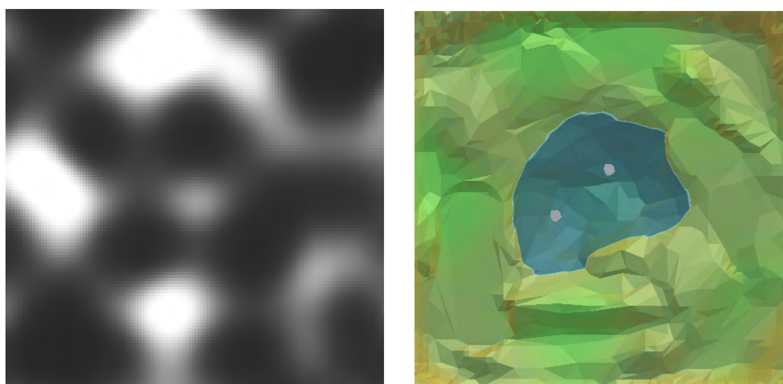


Figure 5.1: This figure depicts the correspondence between the soil map and the world. The brighter green represents a high density of soil in the area, while the brownish green corresponds to low soil.

5.1.1 Dropping Fruits Experiment

Before starting with the experiments with both entity types, bushes showed a problem with their fruit genes. Those genes were designed as a link between them and the herbivores, but as they did not make it into this version of the application, their values had little to no consequences on the survival of bushes. This disrupted the species detection algorithm, as a lot of different subspecies appeared depending on the value of these genes.

The solution that helped mitigate this problem was discarding FR-9.1.4, which forced bushes to waste energy if they were dying, hindering their survival due to the additional energy usage of their fruits. With this change, the species detection algorithm returned simpler classifications. This is clearly pictured in Figure 5.2, where one sample gene is used to show the effect of disabling the fruit-dropping logic. In the simulation where bushes could save themselves by dropping the fruits, dozens of species appeared due to the freedom the genetic configuration of fruits granted, while in the simulation where they could not drop fruits, only a couple species survived. In these plots, it is also apparent that another issue the species detection algorithm has is missing identifications of clusters. This causes a single species to be classified with several labels, which makes them show several colors in all their statistics.

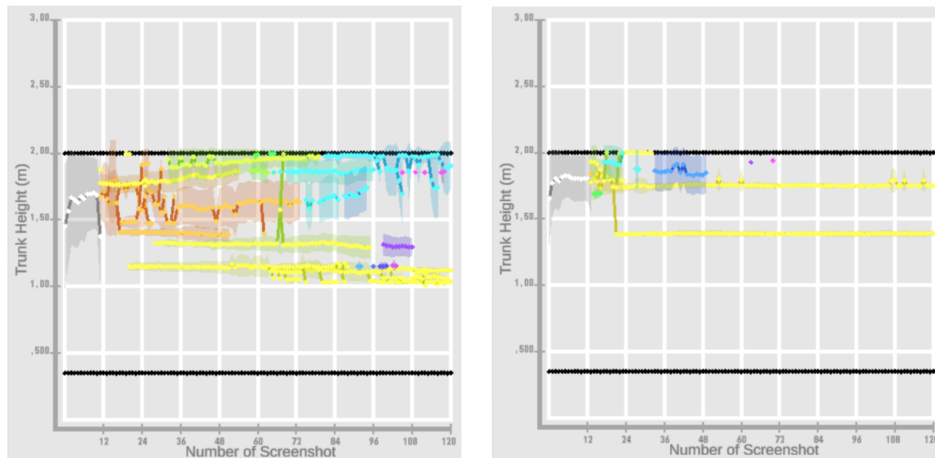


Figure 5.2: Comparison of the species detection algorithm results with dropping fruits on the bushes' grace period on the left and not dropping them on the right.

This behavior appears because, with fruits dropped on the grace period, the disadvantage of having more energy wasted by fruits disappears, which in turn makes any genetic configuration of fruits viable for survival, resulting in the existence of many more species due to the differences in these genes. Furthermore, because it is easier to survive when dropping fruits, the population size is vastly increased.

5.1.2 Balanced Constants Experiment

Once the problem with the fruits is solved, a new simulation is run to analyze the results obtained with the default values of the constants, where trees are spawned in one corner and bushes in the opposite one. The evolution of their populations is pictured with the heat maps in Figure 5.4 and the evolution of their genes in Figure 5.5. Additionally, Figure 5.3 shows the final look of the scene.

Figure 5.3: Final look of the scene in the balanced simulation experiment.

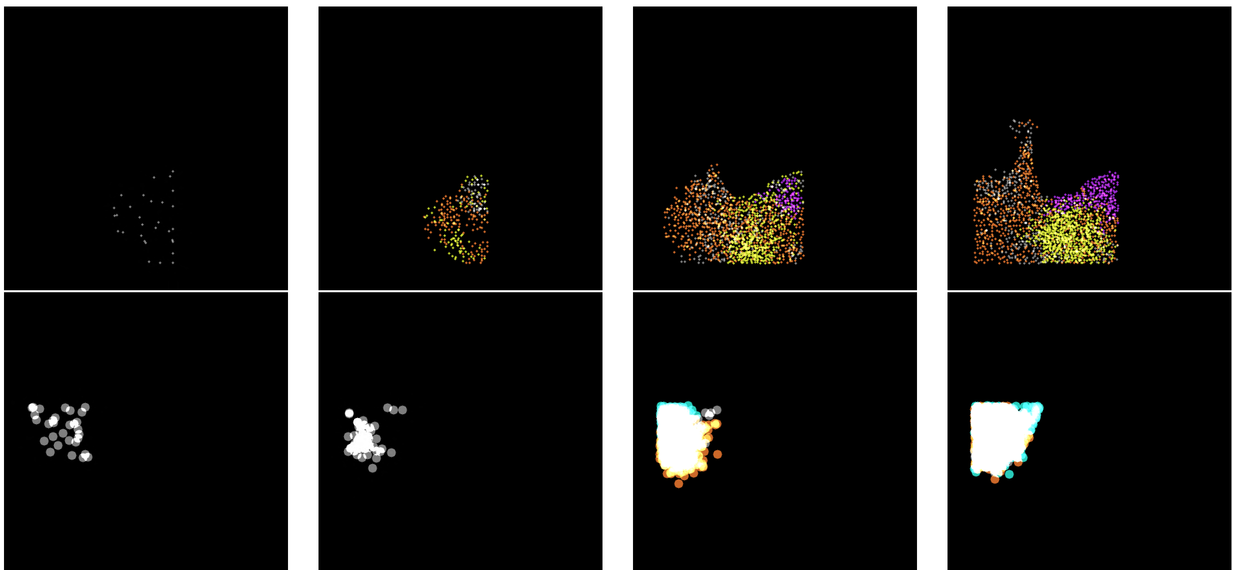
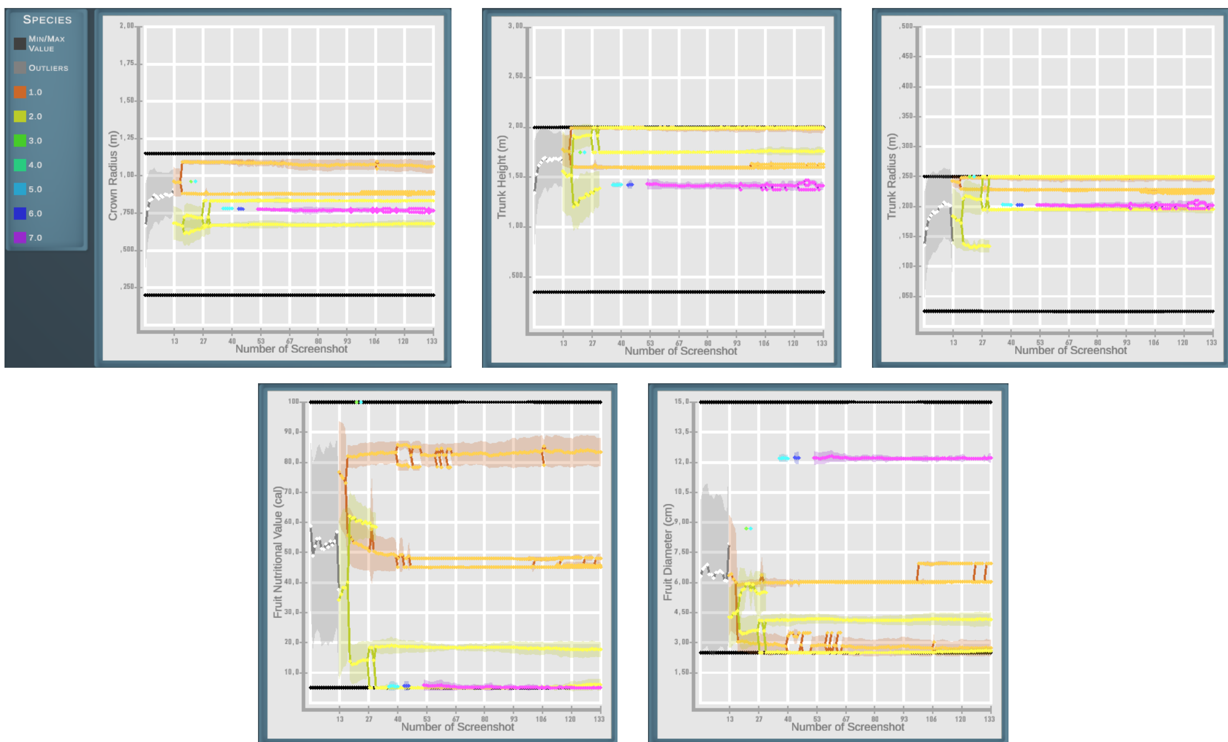
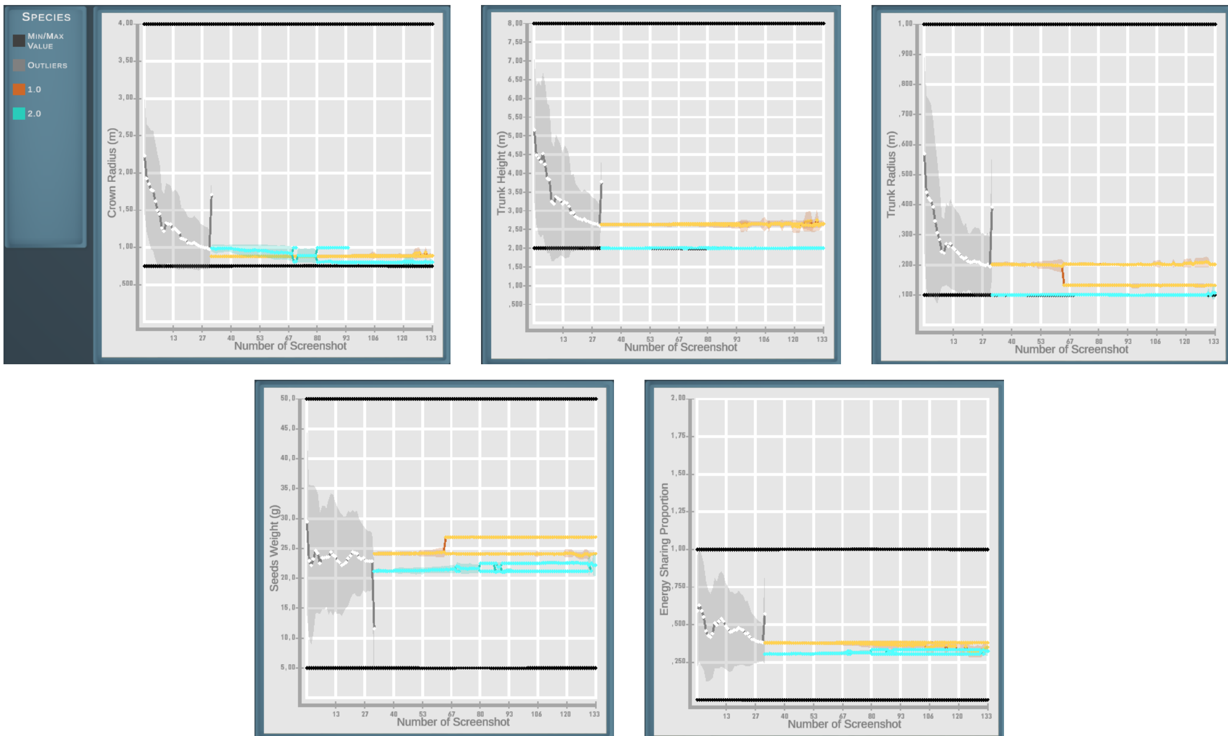


Figure 5.4: Heat maps of several snapshots in the balanced simulation experiment. The top row depicts heat maps of the bushes, and the bottom row depicts heat maps of trees.



(a) Evolution of bushes' genes.



(b) Evolution of trees' genes.

Figure 5.5: Evolution of genes in bushes and trees in the balanced simulation experiment. Each sub-figure shows the legend that corresponds to all of its respective plots.

With all these statistics, we can reach several conclusions about the evolution of the experiment. Firstly, although trees had enough time to expand as much as they could, they only managed to survive in their initial high-soil area, while bushes managed to conquer the rest of the available surface. Trees evolved into two similarly small species, one of which was slightly smaller, but both of which had a high resource-sharing percentage, which helped them sustain a dense cluster during the whole experiment. Meanwhile, bushes evolved into almost the biggest genetic configuration they could have. However, this genetic configuration was still smaller than the smallest trees, and that is why trees managed to survive the expansion of bushes. For the energy cost function of plants, it seems that the average best-fit solution is with a trunk in the range of $[1.4, 2.75]$ meters high and $[0.1, 0.25]$ meters thick, depending on how much soil there is available. The size of the crown is tightly restricted to those two first genes by FR-8.6.2, so as a consequence, its radius is in the range of $[0.75, 1]$ meters.

With the obtained plots shown in Figure 5.5 we can also analyze the efficacy of the species detection algorithm. By focusing our attention on Figure 5.5(a) we can see that the algorithm manages to find several species that are quite similar. This can be misleading in some cases, as it looks like some lines should belong to the same species, but if we look at the *Fruit Nutritiousness* and *Fruit Size* genes, we can see that they have strong differences. The same happens in Figure 5.5(b), but here the difference is more subtle. In these plots, we have a really clear visualization of the problem of the algorithm with missing cluster identifications. It occasionally fails to find a cluster in a snapshot, causing a species to be labeled with multiple names due to the discontinuity, as shown by species 5, 6, and 7 in Figure 5.5(a).

5.1.3 Unbalanced Constants Experiment

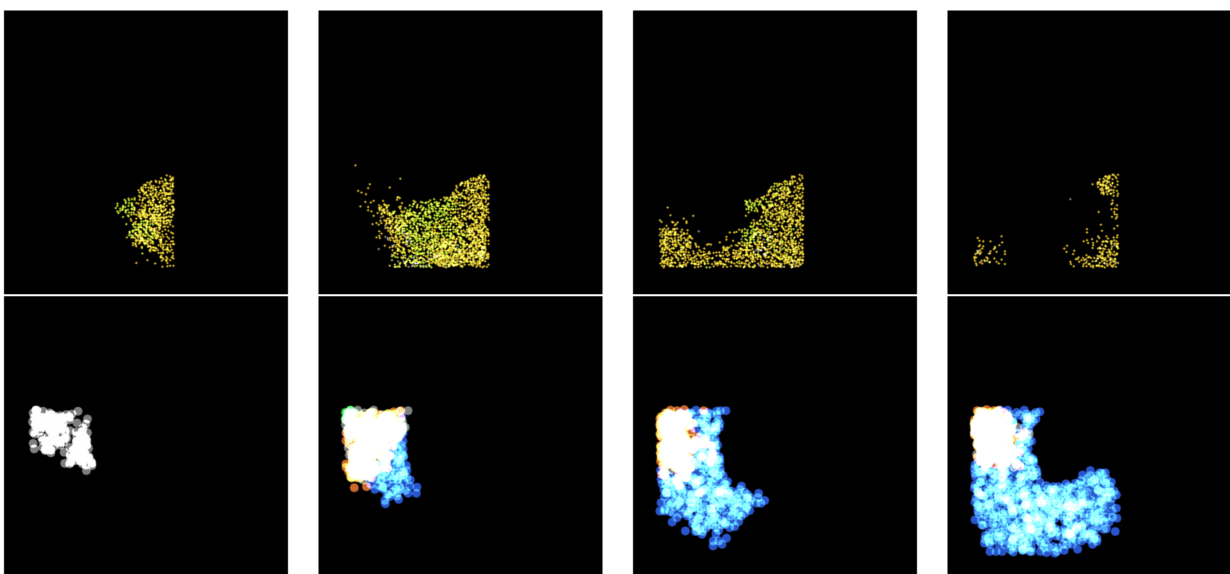


Figure 5.6: Heat maps of several snapshots in the balanced simulation experiment. The top row depicts heat maps of the bushes, and the bottom row depicts heat maps of trees.

While the values for the constants *LightPerSquareUnit*, *MaxSoilPerSquareUnit*, *EnergyReqPerTrunk-CubicUnit*, and *EnergyReqPerLeavesCubicUnit* in the balanced experiment were configured as 1.5, 2, 25, and 2.5 respectively, the unbalanced experiment used values of 3, 8, 20, and 2. With these new values, trees could survive in low-soil areas and dominate the whole available surface, extinguishing bushes with their expansion. This can be seen in the heat maps of both entity types in Figure 5.6.

5.1.4 Sharing Resources Experiment

Regarding the specific genes of trees, the seeds weight does not seem really interesting at first, that is, until we see the results of the evolution. As expected, the evolution of all the species behaves in the same way as the prisoner's dilemma, in which the non-collaborating role takes over the simulation. However, in Figure 5.7 and Table 5.1 we can see that a species managed to survive, and the reason for that is the seeds weight gene.

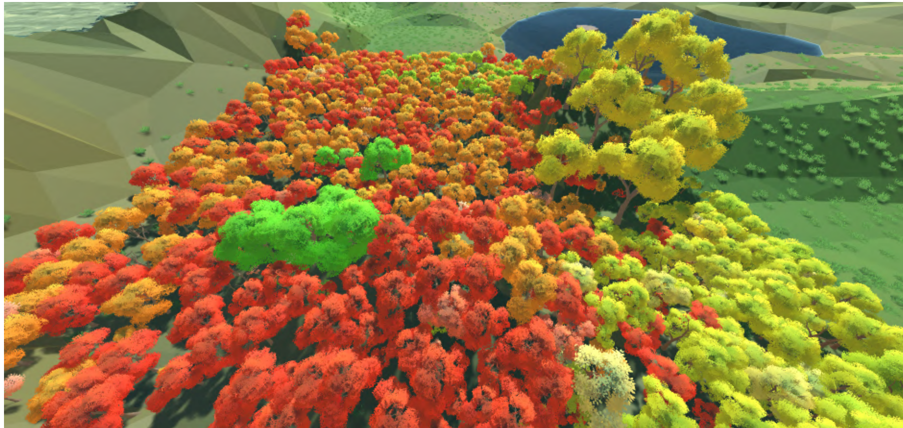


Figure 5.7: This figure shows the resulting scenery of the evolution of trees in an experiment to analyze the behavior of the sharing gene.

Species	Height (m)	Trunk radius (m)	Crown radius (m)	Seeds weight (g)	Sharing (%)
1	2	0.1	0.85	5	0
2	2.2	0.1	0.75	35	45
3	3	0.15	1.5	50	0
4	5.6	0.25	1.85	30	0

Table 5.1: Table that lists the genes of the species found in the same experiment as the one presented in Figure 5.7. Each field represents the average value of a gene in their population. The species number corresponds to: 1 is the red species, 2 is the yellow and short species, 3 is the green species, and 4 is the yellow and tall species.

When the seeds have little weight, a species can spread and cover a large area, but when the seeds are heavier, they tend to create denser clusters. This made it harder for the dominant species to take over the area the others were occupying. Additionally, the benefit of the sharing gene also took place here, as sprouts can grow under a mature tree, as they get light from their neighbors, even though sprouts cannot get it themselves, because they are completely covered by taller trees. This argument is also supported by species 3 and 4. They also managed to survive the expansion of species 1 thanks to their seeds weight, but their clusters are much smaller because their sprouts cannot survive if they cannot access light by themselves.

5.2 Experimenting with the Genetic Algorithm

The intent behind the GA was to solve an imbalance problem that would have arisen from the prey-predator model that the animals represented, but they were not implemented in this version of the simulator. However, the prey-predator model can be reformulated to describe the interactions between two species of plants competing for resources in their environment. This new resources-competition model can also show an imbalance as one species can disappear as a consequence of the second one dominating the whole area. In the case of the simulator, the amount of available resources is always constant, and what the plants area actually competing for is the space they occupy to gather those resources. In the previous experiments with the simulator, the constants that define the plants' requirements and available resources were adjusted to allow the survival of both types of plants, but to experiment with the GA they were changed to grant an advantage to one of them over the other, giving the algorithm a new type of imbalance to solve. The new values used were the same as the unbalanced experiment explained in Section 5.1.3.

Figure 5.8: This figure shows the resulting scenery of the evolution of trees and bushes in an experiment to analyze the yielded result of the genetic algorithm over the prey-predator model.

With this, the algorithm could be run to find a genetic configuration capable of sustaining both species with the given simulator configuration. Several decisions were taken to achieve this goal by configuring the experiment:

Simulator A high time step (2 hours) and a low percentage of the map (25%) were used to make simulations run as fast as possible. A simulation duration of 30 days guaranteed that one species would go extinct if the imbalance was present.

Entities Configured to have a high mutation (10%), low life span (1 to 2 days), and high reproduction (5 to 8 descendants a day) allowing more generations to explore their survival function.

Survival These are the constants that define the survival conditions of plants, with elevated requirements but plentiful resources as well, which gave trees an advantage over bushes.

Genetic Algorithm Configured with: Weighted Average crossover method, a population size of 20 chromosomes, 10 epochs, no maximum fitness stop condition, and a high mean fitness stop condition (90%).

The execution of this experiment lasted for around 55 minutes, with an average of 17 seconds per simulation. The evolution of the mean fitness score of the population can be seen in the user guide, in Figure E.12. Two new simulations were executed with the specified simulation configuration, one with the default genetic configuration and another one with the results obtained through the GA. In the first simulation the bushes did not stand a chance, as the trees evolved into a low-energy-usage configuration which could survive even on areas with a low soil concentration. However, in the second simulation, the initial genetic configuration provided a spawn of trees that were unable to survive in said low soil areas, which granted bushes an area to dominate, as seen in Figure 5.8. These results were not perfect, since with long simulations trees finally mutated to dominate the whole area. However, bushes managed to survive for longer, which was in fact the objective function of the algorithm.

6 Conclusions

The result of this bachelor's thesis is the Ecosystem Evolution Simulator application, which is capable of simulating the evolution of a set of entity types and provides the necessary tools for their analysis. Particularly, the application has been designed to analyze the prey-predator model, which has served as an example of the fragility of real ecosystems. A stable prey-predator model relies on a really fragile equilibrium that, if disrupted, can end in the extinction of several species in its ecosystem. However, as animals were not implemented in the version of the application presented in this thesis, the prey-predator model has been reformulated as another mathematical model called 'resources competition' to adjust for plants. In this model, we compare how two types of plants (and their different subspecies) compete for the constant resources there are in the ecosystem. With this new model, we can also see how a slight difference in the environment can give a population an advantage over the rest and dominate the ecosystem, while the rest of the populations go extinct.

Although the Genetic Algorithm was originally designed to solve the imbalance of the prey-predator model, with the resources competition model it also served its purpose by finding initial genetic configurations for the simulator that diminished the probability of an entity type dominating over the rest. However, the GA has to be used wisely, because the selected objective function is not cheap to compute, as it runs a simulation to find the score of a single chromosome. This poses two problems: the execution time required to find a result can be drastically increased by a poor configuration of the experiment, and the intrinsic randomness of a simulation can yield results that are not desired. Although those problems are acknowledged, they were not solved in this thesis. To solve the first problem, it would be necessary to boost the performance of the simulator itself, which still has a lot of room for improvement. This could be done by implementing a series of changes in the design of the simulator in terms of parallelism, or even by redesigning the rules that define the behavior of entities to make them less expensive to compute. This last point is important, as the majority of execution time is spent in the *PlantManager* class calculating the available resources of each plant. The second problem is the reason why, in the experiments with the GA in Chapter 5, the maximum score stop condition was disabled, as in a lucky simulation some plants would manage to survive until the last frame, although their species were about to go extinct, giving the chromosome a perfect score. To solve this problem, several simulations could be run per chromosome evaluation, which would be nonsense as the execution time

would be drastically increased. This is why it is a better solution to improve the performance of a single simulation, discard the maximum score stop condition, and let an experiment run for more frames or more epochs to ensure the result found is adequate.

With all this, the objectives listed in Chapter 1 have been successfully accomplished. In addition, some design decisions were made to broaden the simulator's capabilities beyond the main goal of analyzing ecosystem fragility. The most relevant inclusion was the resource-sharing gene introduced in trees, which showed how different combinations of genetic configurations managed to survive. This provided evidence for the fact that natural selection does not simply find the "best" solution. The configuration that could maximize the population size of trees is a high sharing percentage, as this managed to get denser clusters by especially helping in the sprout phase. The problem was that in this situation, a non-sharing tree had higher chances for survival, but with its coming generations, the benefit of being a non-sharing tree in a sharing population of trees would vanish as the population of the second diminished. This results in a population of trees unable to form dense clusters, which would additionally have a lower probability of survival in their sprout phase. Hence, the natural selection turned the ecosystem into an environment in which it is harder to survive than in the initial situation.

6.1 Future Work

It is important to note that the version of the application shown in this thesis is in its early stages, and there is a wide room for improvement and optimizations planned for the future. A portion of the application was designed with the objective of exploring interesting relationships between entities like the prey-predator model, sharing benefits, cost-gain balance, mutualism, parasitism, etc. For this, the next step would be to implement those requirements stated in Appendix A.1 about the inclusion of animals in the simulator. Additionally, other interesting experiments can come from modifying the configuration of a simulation once it has already been started, like, for example, the introduction of an intrusive species into a stable ecosystem, or the consequences of a decrease in the quality of soil in the environment. Another change that could have an interesting analysis would be a dynamic soil, where plants would wear down the area they live on, and herbivore feces would increase the amount of soil.

Other types of improvements to the application would include increasing the stability of the species detection algorithm and reducing the amount of noise with the non-classified entities, as explained at the end of Section 4.2.2. Other minor future improvements are pending bug-fixing or quality-of-life, like, for example, making the GUI scale with the resolution, or skipping simulations in the Genetic Algorithm experiments if they are stable and not subject to change. The final major improvement would be integrating all the tools that will be provided by Unity's DOTS package once it is released. This package is focused on managing thousands of similar objects, where all of them share the same logic but have distinct data values. This would mean a huge rework of the *Ecosystem Simulator* subsystem would be required, but it would attain a boost in performance that would not go unnoticed.

Bibliography

[1] Zephyr's Simulators, "Ecosystem Evolution Simulator alpha-0.1 PREVIEW", 2022-12, https://www.youtube.com/watch?v=PkzSpne3oLk.

[2] Unity Technologies, "Unity3D 2021.2.17f", 2022-03-24, https://unity.com/.

[3] Microsoft, "C# 8.0", 2019-09, https://learn.microsoft.com/en-us/dotnet/csharp/.

[4] Microsoft, "High-Level Shader Language (HLSL)", https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl.

[5] Microsoft, "Visual Studio 2019", 2019-03-02, https://visualstudio.microsoft.com/.

[6] Codice Software, "PlasticSCM", 2021-12-15, https://www.plasticscm.com/.

[7] L. He, X. Ren, Q. Gao, X. Zhao, B. Yao, and Y. Chao, "The connected-component labeling problem: A review of state-of-the-art algorithms," *Pattern Recognition*, vol. 70, pp. 25–43, 2017. https://www.sciencedirect.com/science/article/pii/S0031320317301693.

[8] "The gof design patterns memory - learning object-oriented design programming," Oct 2017. Accessed: 2022-09-6, http://w3sdesign.com/?gr=b06&ugr=struct#gf.

[9] H. Celiker and J. Gore, "Clustering in community structure across replicate ecosystems following a long-term bacterial evolution experiment," *Nature Communications*, vol. 5, p. 4643, Aug 2014. https://doi.org/10.1038/ncomms5643.

[10] W. Huang, X. Cao, F. Biase, P. Yu, and S. Zhong, "Time-variant clustering model for understanding cell fate decisions," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 111, 10 2014. https://doi.org/10.1073/pnas.1407388111.

[11] "Kmeans - machine learning glossary - google developers." Accessed: 2022-09-6, https://developers.google.com/machine-learning/glossary?hl=en#k-means.

[12] "The determination of cluster number at k-mean using elbow method and purity evaluation on headline news." Accessed: 2022-09-6, https://ieeexplore.ieee.org/document/8549751.

[13] "k-means advantages and disadvantages - machine learning - google developers." Accessed: 2022-09-6, https://developers.google.com/machine-learning/clustering/algorithm/advantages-disadvantages.

[14] L. McInnes, J. Healy, and S. Astels, "hdbscan: Hierarchical density based clustering," *The Journal of Open Source Software*, vol. 2, mar 2017.

[15] O. C. Carrasco, "Gaussian mixture models explained," *Towards Data Science*, 06 2019. https://towardsdatascience.com/gaussian-mixture-models-explained-6986aaf5a95.

[16] J. Woodcock, "The predator-prey relationship: An intricate balance," *Adirondak Alamanack*, February 2022. https://www.adirondackalmanack.com/2022/02/the-predator-prey-relationship-an-intricate-balance.html.

[17] M. Obitko, “Introduction to genetic algorithms.” https://www.obitko.com/tutorials/genetic-algorithms/index.php.

[18] Unity, “Compute shaders,” *Unity Documentation*, 10 2022. Accessed: 2022-04-16, https://docs.unity3d.com/Manual/class-ComputeShader.html.

[19] Unity, “Dots packages,” *Unity Documentation*, 2022. Accessed: 2022-06-5, https://unity.com/dots/packages.

[20] F.-Y. Kuo, T.-H. Wen, and C. Sabel, “Characterizing diffusion dynamics of disease clustering: A modified space–time dbscan (mst-dbscan) algorithm,” *Annals of the Association of American Geographers*, vol. 108, pp. 1168–1186, 04 2018. https://doi.org/10.1080/24694452.2017.1407630.

[21] P. Doxakis. "HdbscanSharp", 2020-08-06, https://github.com/doxakis/HdbscanSharp.

[22] M. L. Ber. "Unity Asset Store: ’SerializabeDictionary’", 2018-05-18, https://assetstore.unity.com/packages/tools/integration/serializabledictionary-90477.

[23] C. Nolet. "Unity Asset Store: ’Quick Outline’", 2022-03-07, https://assetstore.unity.com/packages/tools/particles-effects/quick-outline-115488.

[24] MalberS Animations, "Unity Asset Store: ’Poly Art: Animal Forest Set’", 2022-10-12, https://assetstore.unity.com/packages/3d/characters/animals/poly-art-animal-forest-set-128568.

[25] R. Storms. "Unity Asset Store: ’Low Poly Fruit Pickups’", 2018-05-18, https://assetstore.unity.com/packages/3d/props/food/low-poly-fruit-pickups-98135.

[26] Polytype Studio, "Unity Asset Store: ’Lowpoly Environment - Nature Pack Free’", 2022-07-28, https://assetstore.unity.com/packages/3d/environments/lowpoly-environment-nature-pack-free-187052.

[27] ’junkohanhero’, "Font: Animals Are Like People", https://www.1001freefonts.com/animals-are-like-people.font.

[28] Typodermic Fonts Inc., "Monospaced Font: Monofonto", 1999, https://www.1001fonts.com/monofonto-font.html.

[29] UI Here, "Scatter Plot svg vector icon", https://www.uihere.com/free-icon/scatter-plot-svg-vector-icon-162734.

[30] The Oxygen Team, "Actions office chart area Icon", https://iconarchive.com/show/oxygen-icons-by-oxygen-icons.org/Actions-office-chart-area-icon.html.

[31] Freepik, "Special Lineal color icons", https://www.flaticon.com/authors/special/lineal-color.

Acronyms

API Application Programming Interface.

CCL Connected-Component Labeling.

EES Ecosystem Evolution Simulator.

FOV Field of View.

FPS Frames Per Second.

FSM Finite State Machine.

GA Genetic Algorithm.

GUI Graphical User Interface.

HDBSCAN Hierarchical Density-Based Spatial Clustering of Applications with Noise.

PCA Principal Component Analysis.

APPENDICES

A

Functional Requirements for the Ecosystem Evolution Simulator Application

A.1 Ecosystem Simulator

FR-1.– Any value that is not explicitly elicited in the requirements of the *Ecosystem Simulator* subsystem will be a configurable constant used to tweak the behavior of the entities and/or environment.

FR-2.– The scenery on which simulations may be run will be static and predefined.

- **FR-2.1.–** The surface of the scenery will be composed of plains and hills.
- **FR-2.2.–** The scenery will be bounded and surrounded by mountains.
- **FR-2.3.–** A lake will occupy the center of the map.

FR-3.– Each entity type will have a life expectancy, determining how old it can get before dying.

FR-4.– Each entity type will have a childhood phase followed by an adulthood phase.

- **FR-4.1.–** An entity may only reproduce if it is in its adult phase.
- **FR-4.2.–** The constant 'MinimumAgeForReproduction' will determine how long a childhood phase may last for each entity type.
- **FR-4.3.–** When an entity is in its childhood phase, its properties and/or capabilities will be proportionally diminished. Each entity type will define its affected properties in their respective requirements.

FR-5.– A set of vital requirements will determine the conditions for their survival for each entity type.

- **FR-5.1.–** The vital requirements of an entity will depend on its age and its genetic configuration. Each entity type will define its vital requirements.
- **FR-5.2.–** An entity will die prematurely if its vital conditions are not fulfilled.
  - **FR-5.2.1.–** Entities will have a grace period in which they will try to comply with their vital requirements before actually dying.
  - **FR-5.2.2.–** Entities that are in their grace period will not be able to reproduce.

FR-6.– Every entity type will have a set of genes, which will be called its genetic configuration.

- **FR-6.1.–** The genetic configuration of an entity will be inherited from its parents.
  - **FR-6.1.1.–** If the reproduction of the entity type is asexual, its children will get a copy of the parent's genetic configuration. However, if the reproduction is sexual, children will get a mixture of the genes of their parents, following a crossover method chosen by the user like the ones defined in FR-21.3.
  - **FR-6.1.2.–** On inheritance, there will be a probability that the value of a gene mutates.

FR-6.2.– The possible values of each gene type will be restricted to a range of values. These imposed limits will be configured to avoid entities that do not feel realistic, like bushes 10 meters high or rabbits with legs doubling the size of the rest of the body.

FR-6.3.– Each gene type will have a mutation intensity factor, which will determine, with a percentage, how big the leap in value can be in the range of values of the gene.

NOTE: Every requirement that defines a gene type in the following entity types will be tagged with the symbol '**[Gene]**', and will state its name, a note explaining why it is interesting, and how it will affect an entity with one or more aspects from the following list:

- Its capabilities, like movement speed, reproduction radius, height, etc.
- Its behavior, like courage, sociability, etc.
- Its looks. This feature will be critical for visually understanding an entity's genetic configuration without having to look at its exact gene values.
- Its energy usage. Usually, genes that increase the capabilities of an entity will have an associated energy cost.

FR-7.– An entity that is born outside the bounds of the map or inside the lake will die instantly.

FR-8.– The simulator will include plant entity types.

FR-8.1.– Plants will have asexual reproduction.

FR-8.2.– Plants will have a reproduction radius within which they may spawn a child. The spawn position of a child plant will follow a 2D normal distribution with a mean equal to the parent position and a standard deviation proportional to this constant.

FR-8.3.– Plants will require light to survive.

FR-8.3.1.– A unit of light will be equivalent to a unit of energy.

FR-8.3.2.– Plants will gather light through their leaves. The amount of light a plant has access to will be calculated as the area their leaves cover in the x-z plane that is not already covered by plants above.

FR-8.3.3.– The size of a plant's crown will portray an energy cost. A percentage of this energy must come only from the light gathering through the leaves. This percentage will be known as the constant 'MinimumLightRequirement.'

FR-8.4.– Plants will require soil to survive.

FR-8.4.1.– A unit of soil will be equivalent to a unit of energy.

FR-8.4.2.– Plants will gather soil through their roots. The amount of soil a plant has access to will be determined by a soil map, which will represent how much soil is available in every area of the map.

A. A plant's available soil will be calculated as the sum of the values in the soil map within its root radius.

B. When several plants share the same area on the soil map, the soil in that area will be evenly shared.

FR-8.4.3.– The volume of a plant's trunk will portray an energy cost. A percentage of this energy must come only from the soil gathered through its roots. This percentage will be known as the 'MinimumSoilRequirement.'

FR-8.5.– Every plant will share the following basic genetic configuration:

FR-8.5.1.– **[Gene]** Leafiness gene. **NOTE:** It will provide a balance between how much energy can be gathered and how much will be used by the leaves.

- Looks: Linked to the plant's crown radius, measured in meters.
- Capabilities: A wider crown radius will grant more area to gather light.
- Energy cost: Directly proportional to the volume of the plant's crown.

FR-8.5.2.– **[Gene]** Height gene. **NOTE:** It will provide a balance between having an advantage over smaller plants in terms of light gathering and the energy cost associated with the volume of its trunk.

- Looks: Linked to the plant's trunk height, measured in meters.
- Capabilities: Taller plants will have an advantage over smaller ones regarding the gathering of light.
- Energy cost: Contributes linearly to the calculation of the energy cost of the trunk's volume.

FR-8.5.3.– **[Gene]** Thickness gene. **NOTE:** It will provide a balance between the length of the roots and how much energy is used by the trunk.

- Looks: Linked to the radius of the plant's trunk, measured in meters.
- Capabilities: The plant's root radius will be proportional to its trunk thickness.
- Energy cost: Linked to the volume of the trunk. Contributes quadratically to the calculation of the trunk's volume.

FR-8.6.– Plants will have an additional set of restrictions on the values of their basic genes so that they keep their pseudo-realistic behavior, as explained in FR-6.2.

FR-8.6.1.– A minimum and maximum height must be set as a function of the thickness.

FR-8.6.2.– A minimum and maximum leafiness must be set as a function of both the height and thickness of the plant.

FR-8.6.3.– Values that exceed any limit will be clamped.

FR-9.– The plant entity type *Bush* will be included in the simulator.

FR-9.1.– Bushes will produce fruits.

FR-9.1.1.– The constant 'FruitGrowthTime' will define the growth speed of fruits.

FR-9.1.2.– A fruit will have an associated energy cost, as a function of its size and nutritional value, times a scaling factor in a constant called 'EnergyUsagePerFruit.'

FR-9.1.3.– Bushes will need to drop fruit to reproduce.

FR-9.1.4.– If a bush is dying, it will drop as many fruits as needed in an attempt to reduce its energy usage.

FR-9.1.5.– Some animals will feed on these fruits. If a bush has no fruits, an animal may feed from its leaves, reducing its capacity for light absorption and making it harder for the bush to meet its minimum light requirement.

FR-9.1.6.– Fruits will spawn pseudo-randomly in between the leaves of the bush.

FR-9.2.– Every bush will extend its basic genetic configuration with the following genes:

FR-9.2.1.– **[Gene]** Fruit frequency gene. **NOTE:** It will provide a balance between having enough fruits to reproduce and feed herbivore animals surrounding the area without wasting too much energy.

- Looks: Maximum number of fruits a bush can grow simultaneously.
- Capabilities: Increased availability to drop fruits in order to reproduce. Bushes with few fruits may lose their ability to reproduce as a result of animals eating their fruits.
- Energy cost: Each fruit has an additional cost to the base energy usage of the bush.

FR-9.2.2.– **[Gene]** Fruit size gene. **NOTE:** It will provide a relationship with animals in which fruits can grow bigger to feed the animals faster. This would result in plants with more remaining fruits.

- Looks: Linked to the diameter of the fruits, measured in centimeters.
- Capabilities: Bigger fruits will grant more energy to animals that feed on them. The reproduction radius of the bush will be inversely proportional to the size of its fruits.
- Energy cost: Contributes in a cubic manner to the cost of a single fruit.

FR-9.2.3.– **[Gene]** Fruit nutritiousness. **NOTE:** It will provide a relationship with animals in which fruits can be more nutritious to feed the animals faster, hence having more remaining fruits. It contrasts with the same feature of the fruit size gene in that the reproduction radius is unaffected by this gene.

- Looks: Decides the type of fruit that is presented in the scene with Lerp(strawberries, cherries, pears, peaches, bananas), where function Lerp(range/options) linearly interpolates between several values or options given a proportion value. This gene is measured in calories.
- Capabilities: More nutritive fruits will grant more energy to animals that feed from them.
- Energy cost: Contributes linearly to the cost of a single fruit.

FR-10.– The plant entity type *Tree* will be included in the simulator. Every tree will extend its basic genetic configuration with the following genes:

FR-10.1.– **[Gene]** Roots interconnection. **NOTE:** This gene will useful to analyze the game theory problem of the prisoner's dilemma with repetition as trees evolve into sharing or non-sharing genetic configurations.

- Looks: Linked to the color of the trunk, decided by Lerp(gray, brown).
- Capabilities: This gene represents the percentage of the resources the tree shares with neighboring trees. A tree lacking resources may benefit from this gene. A tree surrounded by non-sharing trees may die as a result of sharing its resources with them and receiving nothing in return.

FR-10.2.– **[Gene]** Seeds weight. **NOTE:** This gene is interesting to analyze in combination with FR-10.1 to visualize how non-sharing trees might show a greater reproduction radius.

- Looks: Linked to the color of the leaves, decided by Lerp(red, yellow, green). This gene is measured in grams.
- Capabilities: The reproduction radius of the tree will be inversely proportional to the weight of its seeds.

FR-11.– The simulator will include animal entity types.

FR-11.1.– Animals will have sexual reproduction.

FR-11.1.1.– Animals will only be able to reproduce if their vital requirements are met above a certain threshold.

FR-11.1.2.– In order to breed, two animals of the same type but different sexes must find each other, and both of them must be trying to reproduce.

FR-11.1.3.– Each animal entity type will define its offspring as a mean and standard deviation, along with hard-coded minimum and maximum thresholds.

FR-11.2.– Animals will have the vital requirements of hunger and hydration.

FR-11.2.1.– These vital requirements will be proportional to the size, the genetic properties, the age, and the type of animal.

FR-11.2.2.– Animals will hydrate by drinking from the lake.

FR-11.2.3.– Animals will feed from plants if they are herbivores or from other animals if they are carnivores.

FR-11.2.4.– These vital requirements will be on a timer. Once an animal eats or drinks, the respective timer will have its remaining time increased. If any timer gets to zero, the animal will die.

FR-11.2.5.– The hunger and hydration meters will decrease faster when an animal is performing actions like fleeing, hunting, or breeding.

FR-11.2.6.– Non-hungry animals will not hunt other animals or eat plants.

FR-11.3.– Animals will require time to rest and recover their energy.

FR-11.3.1.– If an animal is not well rested (fatigue value is below a threshold), it will find its capabilities diminished, e.g., movement speed, visual acuity, etc.

FR-11.3.2.– If the fatigue value gets to zero, the animal will fall asleep in place.

FR-11.4.– Animals will defecate when their hunger meter drops a set percentage.

FR-11.4.1.– If an animal is a herbivore, there is a chance that a plant will grow where the animal defecated. This new plant will show a crossover of up to two plants the animal has eaten previously.

FR-11.5.– A dead animal will leave its corpse behind for a set amount of time. This corpse may serve as food for other animals.

FR-11.6.– Each animal type will have its behavior defined by a Finite State Machine (FSM) , which will propose the priority of the actions of said animal. e.g., hunting, breeding, fleeing, drinking... These actions may be influenced by knowledge of the environment or detection of elements by the senses of vision and hearing.

FR-11.7.– Every animal will share the following basic genetic configuration:

FR-11.7.1.– **[Gene]** Movement speed gene. **NOTE:** This gene will show a balance between the need to flee or hunt and the energy cost that comes with moving faster.

- Looks: Linked to the size of the legs.
- Capabilities: Grants the animal more speed, measured in meters per second.
- Energy cost: Accelerates the depletion of the hunger and hydration meters in a cubic proportion.

FR-11.7.2.– **[Gene]** Visual acuity gene. The field of view of animals covers 180 degrees in front of them. This gene determines the maximum radius of this field of view. **NOTE:** This gene presents a relationship between energy usage and the efficiency of the senses of an animal.

- Looks: The eyes of the animal will have lighter colors with higher visual acuity. The colors will depend on the type of animal.
- Capabilities: Grants an animal a greater radius of field of view, measured in meters.
- Energy cost: Accelerates the depletion of the hunger and hydration meters in a quadratic proportion.

FR-11.7.3.– **[Gene]** Hearing acuity gene. This gene determines the radius at which this animal might hear things in its surroundings. Animals may be able to figure out what made the noise. Each sound has a different probability of being recognized or misunderstood. **NOTE:** This gene presents a relationship between energy usage and the efficiency of the senses of an animal.

- Looks: Linked to the size of the ears.
- Capabilities: Grants the animal a greater radius of hearing, measured in meters.
- Energy cost: Accelerates the depletion of the hunger and hydration meters in a quadratic proportion.

FR-12.– The animal entity type *Rabbit* will be included in the simulator.

FR-12.1.– Rabbits will feed on fruits and leaves from the bushes.

FR-12.2.– Rabbits will dig burrows.

FR-12.2.1.– Rabbits will remember in which burrow they were born. A rabbit will wander around its burrow when it wants to breed.

FR-12.2.2.– If a rabbit is being hunted and finds a burrow nearby, it will run to hide inside.

FR-12.2.3.– If a rabbit senses (with its vision and hearing) that there are too many rabbits wandering around its burrow, it will walk away to dig a new burrow in a less crowded place.

FR-12.2.4.– A rabbit will go inside its burrow to rest.

FR-12.3.– The offspring of a rabbit will have a mean of 6 rabbits and a standard deviation of 3. The offspring will be clamped to the range $[3, 9]$.

FR-12.4.– Every rabbit will extend its basic genetic configuration with the following genes:

FR-12.4.1.– **[Gene]** Family size gene. **NOTE:** This gene will analyze which density of rabbit populations has a better chance for survival. A dense cluster of rabbits may be easier for predators to hunt, but rabbits will find it easier to breed with a partner. Also, a dense cluster may imply food sources depleting faster.

- Looks: Linked to the color of the fur, decided by Lerp(white, brown).
- Behavior: This gene will influence the decision of a rabbit to dig a new burrow due to the previous one being too crowded. This gene will be measured by the number of rabbits in the sensory area (vision and hearing) when a rabbit is in its burrow.

FR-12.4.2.– **[Gene]** Courage gene. **NOTE:** This gene can show a relationship with FR-12.4.1, in which rabbits with a smaller family size might have higher courage.

- Looks: Linked to the size of the tail of the rabbit. Rabbits with higher courage will have a larger tail.
- Behavior: This gene is measured in meters and is used by the rabbit to decide how far it will wander away from its burrow when searching for food and water.

FR-13.– The animal entity type *Wolf* will be included in the simulator.

FR-13.1.– Wolves will feed on rabbits.

FR-13.2.– Wolves will not have a fixed home. They will form wolf packs to live together, hunt, rest, and breed.

FR-13.2.1.– Each wolf pack will have a leader, who will decide where to go and when to eat, drink, breed, or rest.

FR-13.2.2.– When a wolf pack catches prey, they will remain in place until they have finished eating.

FR-13.2.3.– If two wolf packs come across each other, there will be a probability they fight for territory. This probability will be proportional to the hunger and thirst meters of the leaders of each wolf pack.

FR-13.3.– The offspring of a wolf will have a mean of 2 wolverines and a standard deviation of 1. The offspring will be clamped to the range $[1, 5]$.

FR-13.4.– Every wolf will extend its basic genetic configuration with the following genes:

FR-13.4.1.– **[Gene]** Lone wolf gene. **NOTE:** This gene will analyze wolf pack sizes. In combination with FR-13.2.3, this gene might show interesting interactions between several wolf packs and territory control.

- Looks: Linked to the height of the hump. Wolves that are lonelier will show a small hump, while the opposite occurs in bigger wolf packs.
- Behavior: This gene is measured as the maximum number of wolves in the pack. This gene controls how many wolves a wolf can stand in the same wolf pack. If a wolf pack has more wolves than this gene, the wolf will abandon the pack and become a lone wolf. If a wolf encounters another lone wolf or wolf pack, it may decide if it wants to join or not, depending on this gene.

FR-13.4.2.– **[Gene]** Leadership gene. **NOTE:** Being a leader will grant a wolf the capability to decide which action is more convenient to its vital requirements, but other wolves might challenge him to death.

- Looks: Linked to the color of the fur, decided by Lerp(white, brown, black). Lighter colors show more submissive behavior, while darker ones are linked to dominance.
- Behavior: This gene will be used to determine which wolf will be the leader of a pack. The wolf with the highest leadership gene will lead the rest. If a second wolf is worth the same as the leather, they may fight to death to determine who will be the new leader.

FR-13.5.– The outcome of a fight between two wolves will be decided by a random choice proportional to the wolves' age, movement speed, fatigue, and levels of hunger and thirst.

FR-14.– The animal entity type *Deer* will be included in the simulator.

FR-14.1.– Deer will feed on leaves, fruits, and grass. Grass will be less nutritious, but it will be available on the whole world.

FR-14.2.– Deer will always wander the same area of the map, while avoiding being too clustered with other deer.

FR-14.3.– Deer will be able to fight wolves off if their male population is greater than the wolf pack itself.

FR-14.4.– The offspring of a deer will have a mean of 1 fawn and a standard deviation of 0.5. The offspring will be clamped to the range $[1, 3]$.

FR-14.5.– Every deer will extend its basic genetic configuration with the following genes:

FR-14.5.1.– **[Gene]** Sociability gene. **NOTE:** This gene will show a behavior similar to the one explained in FR-12.4.1, where there will be a relationship between the hazards and rewards of a dense population of deer.

- Looks: Linked to a brighter fur color. Less social deer will have pale fur, while more social deer will show a more vibrant fur color.
- Behavior: This gene is measured in deer in sensory radius, and will determine when a deer feels an area is too crowded, so it will move to a new area with fewer deer. It will also determine how far from the herd of deer it will go.

FR-15.– The user will be able to configure the simulator before a simulation is run.

FR-15.1.– A simulation will have a unique name.

FR-15.2.– A simulation will have a finite duration.

FR-15.2.1.– A simulation will run for a set number of time steps.

FR-15.2.2.– The user will be able to configure how long a simulation will run by setting the 'SimulationDuration' constant.

FR-15.2.3.– The system will calculate how many time steps are required as a function of the maximum time and the value of a time step.

FR-15.3.– The user will be able to decide which percentage of the world will be used in the experiment.

FR-15.4.– The user will be able to configure all the constants that define the rules of the ecosystem and its evolution.

FR-15.4.1.– The user will be able to configure the mutation probability of entities and genes separately.

FR-15.4.2.– The plant constants the user will have control over are:

- Light available, measured in energy units per square meter.
- Maximum soil available, measured in energy units per square meter.
- Energy units required per trunk cubic unit.
- Energy units required per leaves cubic unit.
- Minimum light requirement, measured in percentage.
- Minimum soil requirement, measured in percentage.
- Maximum time not meeting the requirements, measured in hours.

FR-15.5.– The user will be able to configure which entity types will be present in a simulation.

FR-15.6.– The user will be able to configure the initial spawn conditions of every entity type.

FR-15.6.1.– The user will be able to choose how many random entities of each entity type will be spawned at the beginning of a simulation.

FR-15.6.2.– The user will be able to configure the spawn position of each entity type via a spawn probability map generated with Perlin Noise.

FR-15.6.3.– The genetic configuration of each spawned entity will be a random sample obtained via a normal distribution. The user will be able to configure the shape of the distribution by setting a mean and a standard deviation.

FR-15.7.– The user will be able to set a mutation intensity for every gene for every entity type.

FR-15.8.– The user will be able to configure the constants that define the behavior of each entity type that will be present in a simulation.

FR-15.8.1.– The configurable plant constants will be:

- Life expectancy, measured in time.
- Minimum age for reproduction, measured in time.
- Reproduction probability, expressed as how many times in a given time period, for example, five times every two days.

FR-15.8.2.– The specific configurable constants for bushes will be:

- Energy usage per fruit factor.
- Fruit growth time.

FR-16.– The user will not be able to configure the simulator once a simulation is running.

A.2 Statistics Gathering and Generation

FR-17.– During a simulation, the application will save the state of the simulator several times for later analysis. These saved states will be called snapshots.

FR-17.1.– While configuring a new simulation, the user may decide either how many or how often the snapshots are captured.

FR-18.– A snapshot must be able to define the state of the simulator with precision. For this purpose, a snapshot will save:

- The simulation time.
- A timestamp of when the snapshot was taken in real time.
- The state of every entity existing in the simulation, either alive or dying.

FR-19.– A saved simulation file will consist of a collection of snapshots, the constants configuration, the initial genetic configuration, and the simulator's maximum time and speed.

FR-20.– Statistics will be generated via a set of snapshots to give a better representation of the evolution of an experiment. The statistics will be calculated for each entity type independently.

FR-20.1.– For each entity type, the genetic evolution data from its population will be used to calculate how many different species lived during the simulation. Every species will have a unique name and color identifier.

FR-20.2.– The simulator will calculate the mean and standard deviation of every gene and property of every species found.

FR-20.2.1.– The common properties for every entity type will be age and generation number.

FR-20.2.2.– Plants will extend their properties list with:

- Available soil.
- Available light.
- Soil requirement.
- Light requirement.
- Energy usage.

FR-20.2.3.– Bushes will extend their properties list with their number of remaining fruits.

FR-20.2.4.– Trees will extend their properties list with:

- Number of adjacent neighbors.
- Additional soil obtained from their neighbors.
- Additional light obtained from their neighbors.

FR-20.3.– The simulator will generate a heat map image for every snapshot. A heat map will be a spatial representation of a species' population density, in which brighter colors imply more density.

FR-20.4.– An additional global heat map will be generated to average all the heat maps generated for each snapshot.

A.3 Genetic Algorithm

FR-21.– A GA will be used in the simulator. The goal of this GA will be to find some initial configuration that will aid evolution in preventing populations of entities from becoming extinct.

FR-21.1.– A chromosome in the genetic algorithm will propose an initial genetic configuration for the simulator, as defined by FR-24.1.

FR-21.2.– The objective function will account for the survival of every entity type for the longest possible time during a simulation.

FR-21.3.– The GA will provide some flexibility and let the user choose:

- The population size.
- A crossover method from the list: discrete, average, weighted average, and one point.
- The crossover probability.
- The mutation probabilities for chromosomes and their individual values.
- The elitism proportion.
- The maximum epochs for the stop condition.
- The maximum score proportion for the stop condition.
- The mean score proportion for the stop condition.

FR-22.– The best solution from an experiment with the GA will be saved to be used later by the simulator.

FR-22.1.– Along with the initial genetic configuration resulting from the best chromosome found, the GA will save the simulator configuration and the constants used during the experiment.

FR-22.2.– Once selected in a new simulation, a saved result will automatically configure the simulator.

A.4 Graphical User Interface

FR-23.– The application will have a *Main Menu* window that will serve as an access point to the following windows via three buttons:

FR-23.1.– The application will have a set of windows called *New Simulation Configuration*.

FR-23.2.– The application will have a *Load Simulation File* window.

FR-23.3.– The application will have a *GA Configuration* window.

FR-24.– The *New Simulation Configuration* windows will allow the user to configure new simulations.

FR-24.1.– These windows will organize all the settings in the following categories:

Main settings Configurations listed in FR-15.1, FR-15.2, FR-15.3, and FR-17.1.

Initial genetic configuration Configurations listed in FR-15.6.3 and FR-15.7. It will also show for each gene type what its hard-coded limits are.

Constants configurations Configurations listed in FR-15.4.

Entities configuration Configurations listed in FR-15.5, FR-15.8, FR-15.6.1, and FR-15.6.2. A panel will show the spawn map configuration for any entity type and the selected positions on top of the spawn map image. A button will allow the user to choose a new set of random positions using the spawn map.

FR-24.2.– The only field that will require validation is the name of the simulation to avoid corrupting save files. The validation will make sure that this name is not already in use by another simulation.

FR-24.3.– A sub-menu will list the results obtained with the GA experiments. Each result will show the date it was obtained, which entities were involved in the experiment, and the fitness score in the range $[0, 100]\%$.

FR-24.4.– A new simulation may be started through these windows. This will lead to a new window called *Simulation Execution*.

FR-25.– The *Simulation Execution* window will show the user the progress of a running simulation.

FR-25.1.– The *Simulation Execution* window will show the following statistics about the progress of the simulation that is running:

FR-25.1.1.– A progress bar along a number showing the percentage of the simulation that has been calculated so far.

FR-25.1.2.– The number of snapshots taken so far over the number required.

FR-25.1.3.– The number of frames computed so far over the number required.

FR-25.1.4.– The amount of simulation time computed and the objective simulation time.

FR-25.2.– The *Simulation Execution* window will show an estimate of how long the simulation may last with the current computation time and frame number.

FR-25.2.1.– The computation time will be measured as how many milliseconds took the average last 10 frames to compute.

FR-25.2.2.– One field will show the computation speed in milliseconds per frame.

FR-25.2.3.– One field will show the computation speed in FPS .

FR-25.3.– The *Simulation Execution* window will contain two buttons to stop and resume the simulation that it is running.

FR-25.4.– The *Simulation Execution* window will contain a button to access a new window called *Simulation Navigation*. This button will only be available when the simulation is paused.

FR-25.5.– The *Simulation Execution* window will contain a button to save the results gathered so far and exit the simulation. This button may also be pressed before a simulation is finished.

FR-26.– The *Load Simulation File* window will list every save file in the application.

FR-26.1.– A save file slot will show:

- the name of the simulation.
- the date of the simulation.
- the number of snapshots saved.
- the duration of the simulation in simulation time.
- a list with all the entities involved in the experiment.
- a save file thumbnail.

FR-26.2.– When a save file slot is clicked, a new window called *Simulation Results* will appear.

FR-26.3.– A button will allow the user to toggle between loading and deleting modes. The delete mode will erase the selected slot.

FR-27.– The *Simulation Results* window will display the statistics gathered from a selected simulation file.

FR-27.1.– A slider will be used to choose a saved snapshot to show its specific statistics.

FR-27.2.– A list of buttons will be used to choose from which entity types to display statistics.

FR-27.3.– A button will grant access to the *Simulation Navigation* window.

FR-27.4.– Two panels will display the statistics of the selected entity types. If only one entity type is selected, the second panel will be disabled.

FR-27.5.– A panel will have a button for every type of statistic: snapshot heat map, global heat map, gene statistics, and property statistics.

FR-27.5.1.– The heat map statistics will show the images obtained through FR-20.3.

FR-27.5.2.– For the genes and properties statistics, a plot will show the mean and standard deviation for every species, for a selected entity or property via a combo box. Grid axes will be labeled with the corresponding scale and units.

FR-27.5.3.– A panel will indicate in a field how many species and subspecies were found during the statistics generation process.

FR-27.5.4.– A legend will explain necessary information about the plots.

A. The legend will show which colors correspond to which species in the snapshot heat map, gene statistics, and property statistics.

B. The legend will show which colors correspond to a less or more dense concentration of entities in the global heat map.

FR-28.– The *Simulation Navigation* window will render the scene in which the simulations are carried out by loading a state of the simulator, either by a snapshot or a specific frame.

FR-28.1.– Every entity in the scene will update its model and textures to match what its current properties and genetic configuration dictate.

FR-28.2.– The GUI of the *Simulation Navigation* window will contain:

FR-28.2.1.– a button to capture a screenshot and save it as an image. This saved image will be used as a thumbnail in its save file slot.

FR-28.2.2.– a mini-map, showing a top-down view of the map and the current position of the camera.

FR-28.2.3.– a help panel explaining the controls of the camera.

FR-28.2.4.– a side panel listing the properties and genes of a selected entity. Entities will be selected with a forward raycast from the camera when a key is pressed.

FR-28.2.5.– an exit button to go to the previous window.

FR-28.3.– The GUI in the *Simulation Navigation* window may be toggled on or off with a key press.

FR-29.– The *GA Configuration* window will show the settings required to configure a new experiment.

FR-29.1.– This window will contain all the configurations listed in FR-15.2, FR-15.3, FR-15.4, FR-15.5, and FR-15.8. These configurations will follow the same categories as FR-24.1.

FR-29.2.– This window will show an additional category containing all the settings stated in FR-21.3.

FR-29.3.– None of the fields will require any type of validation.

FR-29.4.– From this window, a new experiment may be started. This will lead to a new window called *GA Execution*.

FR-30.– The *GA Execution* window will:

FR-30.1.– show statistics to help understand the progress of the experiment that is running. Among these statistics, there will be:

FR-30.1.1.– a progress bar along a number, showing the percentage of completion of the evaluation of the active chromosome.

FR-30.1.2.– the number of epoch that is being evaluated over the maximum epochs.

FR-30.1.3.– the number of chromosome that is being evaluated over the size of the population.

FR-30.1.4.– the amount of simulation time computed and the objective simulation time for the current chromosome.

FR-30.1.5.– an estimate of how long the simulation may last with the current computation time and frame number, in the same way as explained in FR-25.2.

FR-30.2.– show a plot that will present to the user how the population of chromosomes is evolving. This plot will be linked to a combo box to select which component of the population to analyze. The options for this combo box will be either the score or any gene type that is involved in the experiment.

FR-30.2.1.– For the score, the maximum, mean, and stop conditions will be plotted. The mean score will also show the standard deviation.

FR-30.2.2.– For any gene, the average initial gene mean, gene standard deviation, and gene mutation intensity will be plotted.

FR-30.2.3.– Any of the previously listed configurations may be toggled on or off.

FR-30.3.– have two buttons to pause or resume the experiment.

FR-30.4.– have a button to save the best chromosome found so far and exit the experiment, either if the algorithm has reached a stop condition or not.

FR-31.– The application will have a loading screen to prevent it from freezing on expensive computational tasks, like loading or saving simulations, generating statistics, or loading the *Simulation Navigation* window.

FR-31.1.– The loading screen will display a message indicating which task is being performed.

FR-31.2.– The loading screen will render an animation to help the user understand that the application is not frozen.

B
Class Diagrams

This appendix contains the class diagrams that the Ecosystem Evolution Simulator application follows. As explained in Section 3.4, the following diagrams are simplified and omit several classes, methods, properties, and variables used internally for the operation of the application. However, this simplification makes the diagrams much more clear and easy to follow.

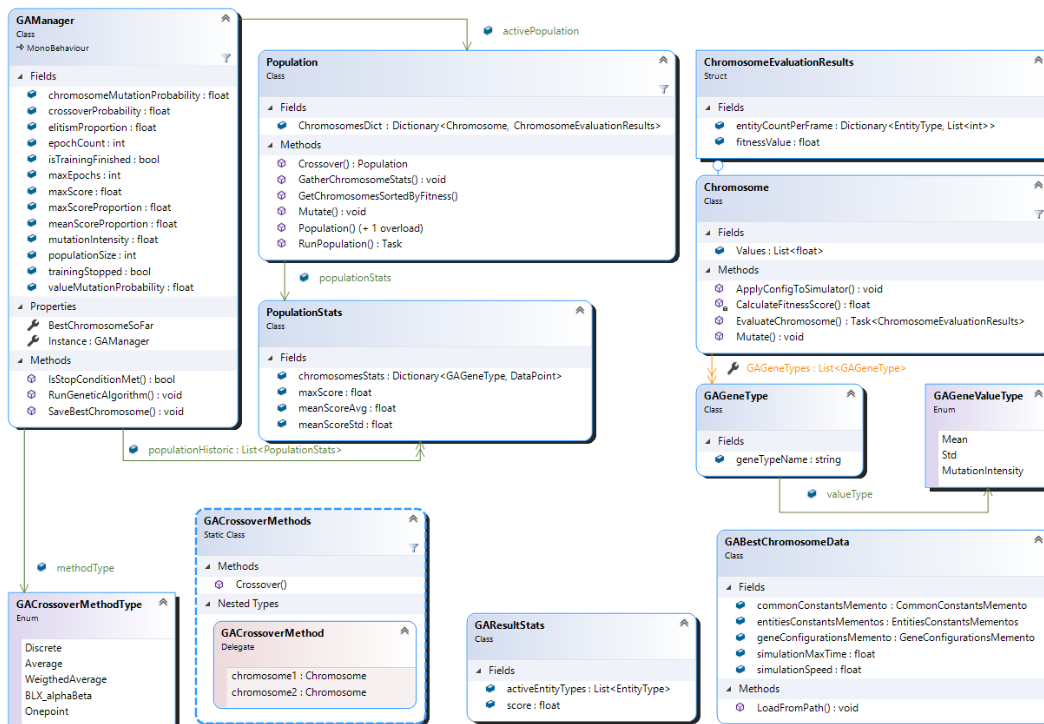


Figure B.1: Class diagram of the *Genetic Algorithm* subsystem.

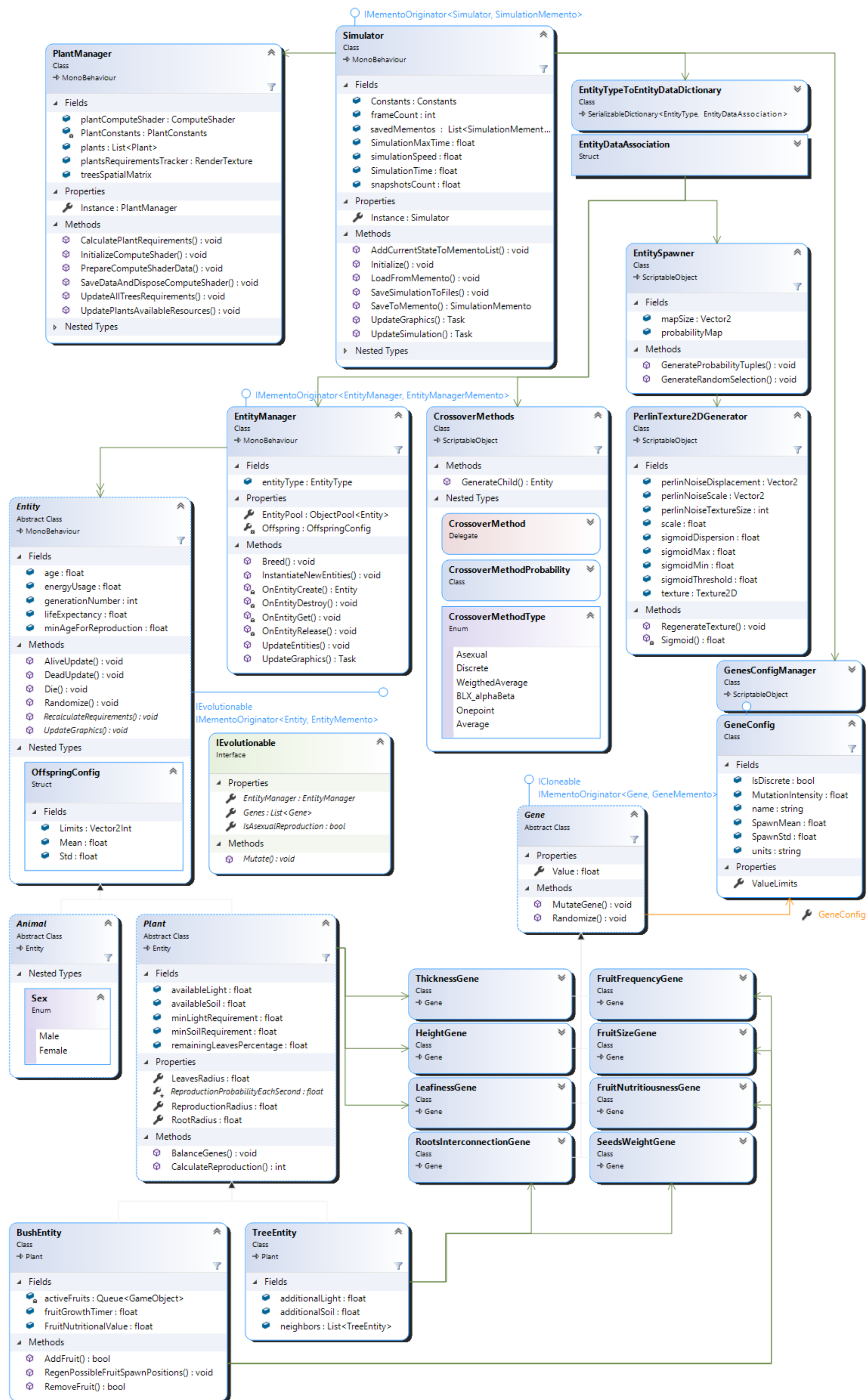


Figure B.2: Class diagram of the *Ecosystem Simulator* subsystem

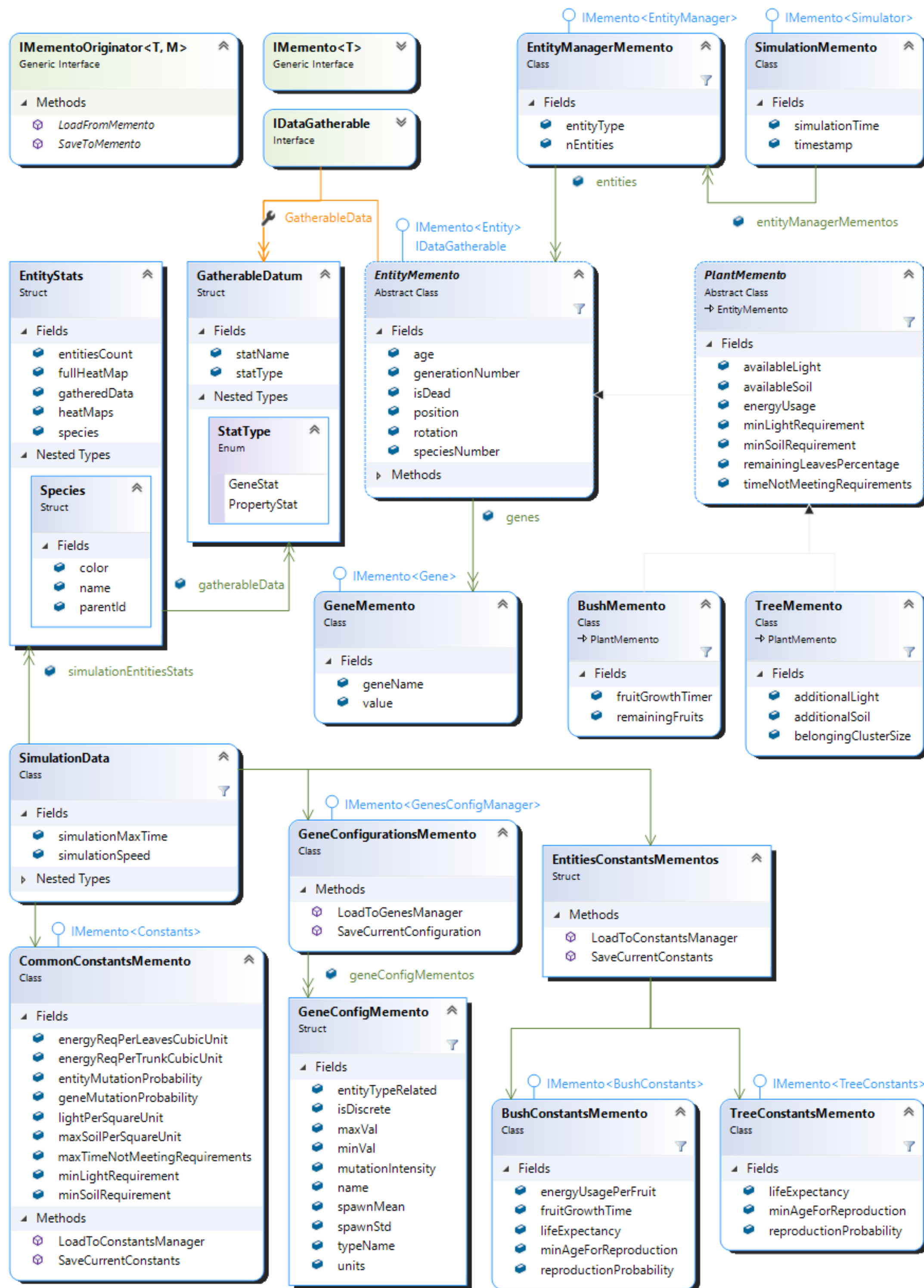


Figure B.3: This figure shows the interfaces used for serialization and statistics retrieval, along with the classes that implement them.

C
Exploration of Evolving Clusters

In the simulator, the survival of an entity can be viewed as a complex multidimensional function. When a species evolves, what they are doing is traversing the search space of their survival function towards a better solution. To better understand how populations may evolve, we can take a look at what different transformations for evolving clusters mean in the context of species recognition. The article *"Characterizing Diffusion Dynamics of Disease Clustering: A Modified Space–Time DBSCAN Algorithm"* [20] shows a classification of evolving clusters specific to the diffusion of diseases. A slightly different classification of clusters is used for the species recognition problem. As Figure C.1 shows, four kinds of cluster evolution are considered. The two most basic transformations are clusters that move and clusters that change size.

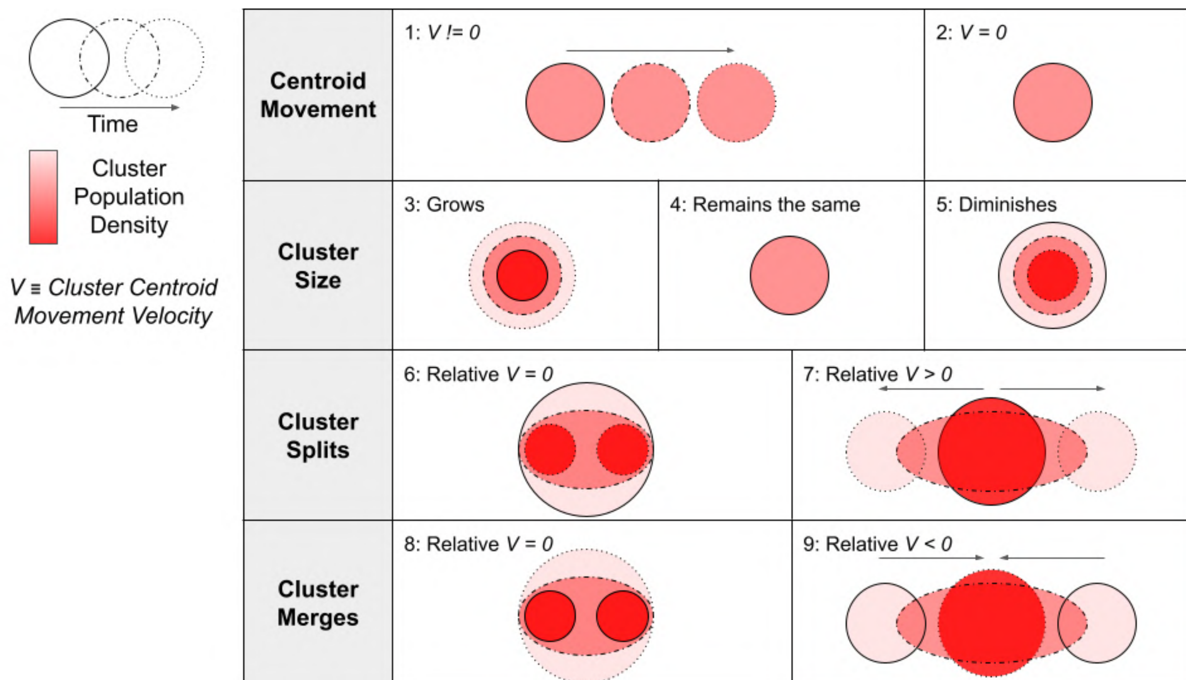


Figure C.1: This table shows the possible forms of evolution a cluster can suffer over time.

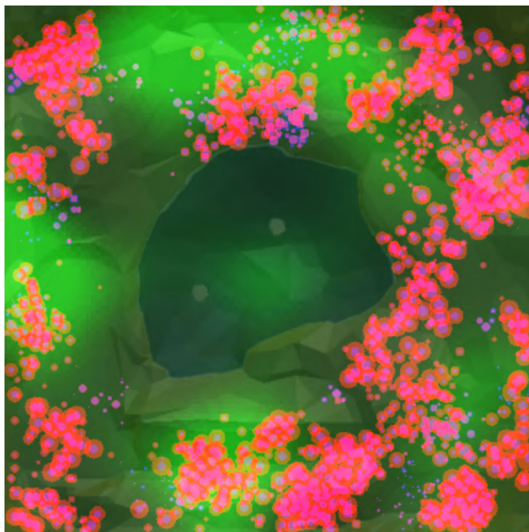
On the one hand, a moving cluster implies that the population contained in said cluster drifts towards a better solution in the survival function. If the cluster centroid does not move, then that means that a local minimum has been found in the survival function. Thanks to the nature of genetic algorithms, the optimization of the survival function does not necessarily end at a local minimum, for mutation can provide a leap in the search space and start a new evolution towards a better solution.
On the other hand, the size of a stable cluster represents how restricted the optimal solution is. To understand what "big" and "small" really mean, we have to give them context. In the simulator, genes are restricted to a range of values, so the survival function has a measurable n-dimensional volume. This means that the size of a cluster can be defined as the percentage of the search space it overlaps. With this, a smaller cluster means that the conditions for survival are harsh, while a bigger one represents an ecosystem in which it is easier to survive, as a wider range of genetic configurations are viable.
Moreover, measuring sizes and distances as a percentage of the search space helps the analysis of clusters with the curse of dimensionality problem, as distances do not get exponentially larger with more dimensions.

More complex behaviors can appear when considering several clusters at the same time. If two viable survival solutions are close to each other, one big cluster can be split into two smaller clusters. When this situation takes place, we will consider that a new subspecies has appeared. On the contrary, two clusters merging in the simulator will mean that two different species (or subspecies) have evolved into the same solution. In this case, the cluster that was further away from the new centroid is discarded, and the nearest one will absorb its population.

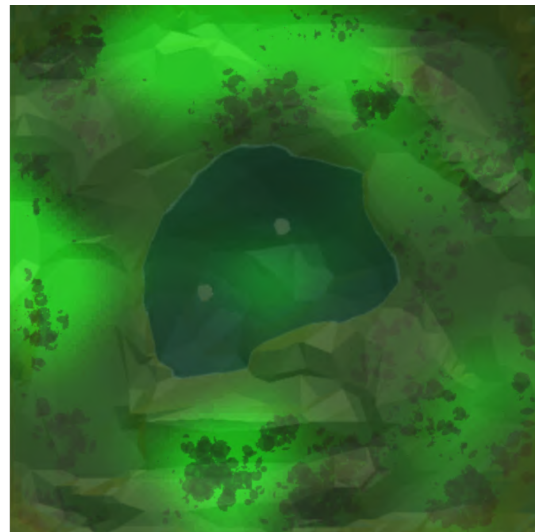
We have to take into account that not only a species is evolving towards the best solution, but the whole ecosystem is. That means that the survival function for each entity type is constantly changing, and the evolution of clusters may not follow a continuous transformation like moving in a constant direction towards a minimum.

D

Plants Requirements' Tracker Texture

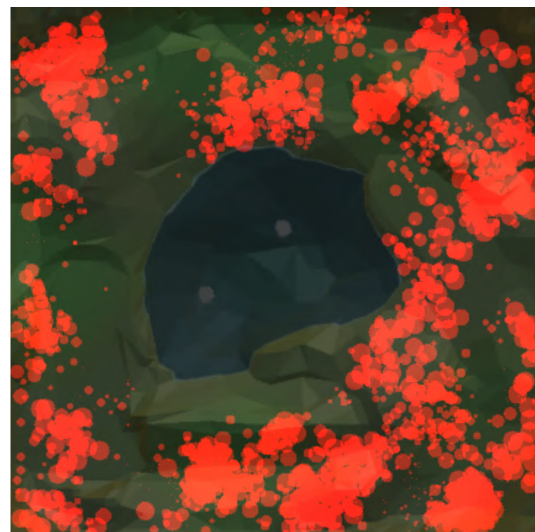


(a) Mixture of all the color channels.



(b) The green channel shows the soil available per pixel, per plant.

(c) The blue channel is used to paint the plants' leaves radii.



(d) The red channel is used to paint the plants' root radii.

Figure D.1: This figure shows the resulting texture for calculating the available soil and light for each plant. The texture is overlaid with a top-down view of the map in the background for reference.

E

User Guide

This appendix contains an explanation for every screen present in the Ecosystem Evolution Simulator application. This resource is an extension of the design document to show the looks and interactions of the application, but a general knowledge of the application is required to follow the guide. Therefore, to better understand the exact purpose of the following fields, tools, and windows head to the requirements elicited in Section 3.1.

The *Main Menu* window shown in Figure E.1, is the first window show when the application is run. It has four buttons:

1.– Button to access the *New Simulation Configuration* windows to configure and run a new simulation.

2.– Button to access the *Load Simulation File* window, where the user can load saved simulations to check their results and analyze the evolution and properties of each entity type.

3.– Button to access the *GA Configuration* window, where the user can configure an experiment to try to avoid extinction in a simulation with a specific set of settings.

4.– Button to exit the application.

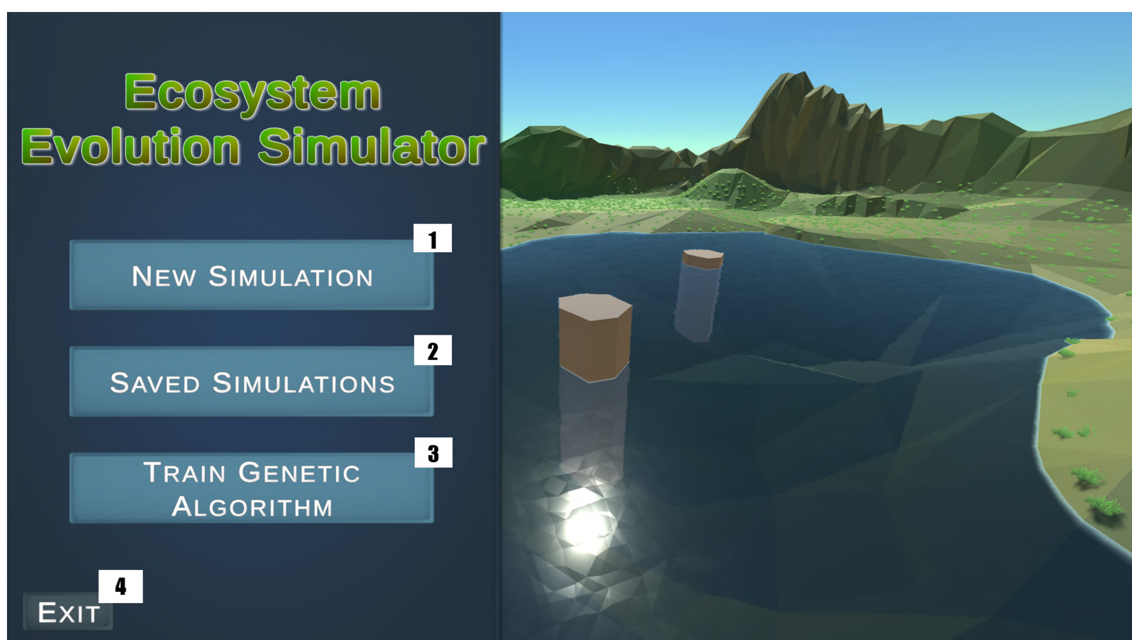


Figure E.1: Main menu for the application

Once the user clicks on button 1, the *New Simulation Configuration* windows shown in Figure E.2 will appear, displaying the following panels and buttons:

1.– Button to go back to the *Main Menu* window.

2.– Panel used to configure the basic operation of the simulator. From top to bottom:

Sim. Name Name of the save file for the simulation. If a simulation with the given name already exists, the field will be highlighted in red, and a new simulation may not be started.

Sim. Duration Duration of the simulation measured in simulation time.

Sim. Speed Measure of how much time passes for each step in the simulation. In a single step, the simulator will calculate the evolution of each entity present in the scene. The smaller this speed is, the more precise the results will be, but the simulation will take more time to compute.

Updates Number of frames that the simulator will calculate. Depends on the duration and the speed of the simulation.

eta. (30 fps) Coarse expectation for the execution time the simulator is going to last, with an average of 30 FPS.

Snap. amount Number of snapshots that the simulator will save to analyze the results. The greater the number of snapshots, the heavier the save file will be, but the results will be more accurate.

Snap. interval This field is complementary to the previous one, and determines how often (in simulation time) a snapshot will be captured.

World scale Used to determine which percentage of the world will be used. It is useful to have smaller simulations that will run faster, which is specially important in the GA experiments.

3.– Panel that determines the spawn distribution of each gene for each entity type. The min and max headers represent the gene value limits. The mean and standard deviation are used to generate samples in a normal distribution. The mutation intensity determines how strongly a mutation will affect the gene during a simulation.

4.– Panel that states the constant values used by different systems in the simulator. From top to bottom:

Entity Mut. Prob. Probability that determines when a descendant will mutate its genes.

Gene Mut. Prob. Probability of a gene mutation when an entity is selected to suffer mutations.

Light/unit^2 Determines how much energy can be gathered through the leaves of a plant in an area of a square unit.

Max. Soil/unit^2 Determines how much energy can be gathered through the roots of a plant in an area of a square unit where all the pixels in the soil map are pure white.

Energy Req./trunk unit^3 Determines how much energy costs to sustain a cubic unit of trunk.

Energy Req./leaves unit^3 Determines how much energy costs to sustain a cubic unit of leaves.

Min. Light Req. Determines what percentage of the energy required by the leaves comes from light.

Min. Soil Req. Determines what percentage of the energy required by the trunk comes from the soil.

Max. time not meeting Reqs. Determines how long the grace period lasts in simulation time. The grace period is the additional time an entity has to accomplish its vital requirements before dying.

5.– Button to display the panel 7.

6.– Button to go to the next window to configure the entities of the simulation.

7.– Panel that lists all the results obtained through the Genetic Algorithm experiments. Once a result is selected, the corresponding configuration will automatically fill all required fields present in the *New Simulation Configuration* windows. Each field shows: the date it was obtained, the final score it obtained, and the entities that were involved in the experiment.

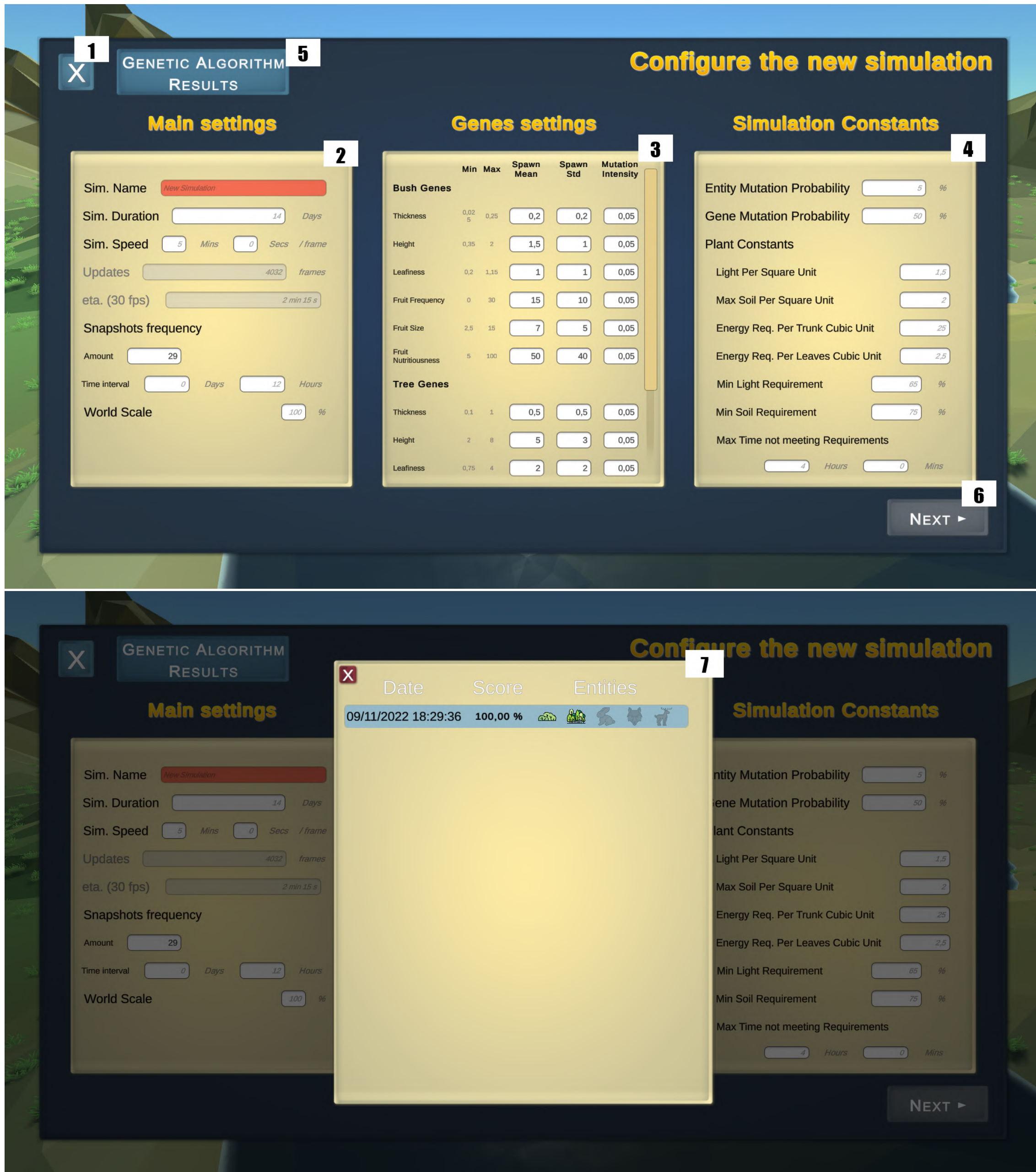


Figure E.2: Windows used to configure a new simulation in the application.

The configuration window shown in Figure E.4 is necessary to decide which entity types will appear in the simulation, their constants, and their spawn coordinates:

1.– Buttons and toggles used to select which entities will be present in the simulation. The toggles are used to select/deselect them, while the buttons will make the respective entity constants appear on panel 2 and their spawn positions in panel 3.

2.– This panel shows the entity constants that belongs to the selected button in 1. Both trees and bushes share the same 3 constants, but bushes have 2 additional ones related to their fruits. From top to bottom:

Life Expectancy How long an entity will live in simulation time before dying of natural causes.

Min. age for reproduction Determines at which age this entity will be able to start reproducing.

Reproduction prob. Determines how often a plant is expected to drop a seed and reproduce. In the case of bushes, it will decide when it will drop a fruit to reproduce.

Energy usage per fruit Determines the additional energy cost that a bush will experience for every single mature fruit it has.

Fruit growth time Determines how often a fruit will spawn for a bush in simulation time.

3.– The panel is used to visualize the spawn positions (with orange dots) and spawn map (black and white) that is configured by panel 4. The reload button will generate a new set of orange dots following the configured spawn map as a probability distribution. The slider will determine the transparency of the spawn map over the map itself, as portrayed in Figure E.3.

4.– Panel used to configure the spawn map for the entity type. From top to bottom:

Num. of entities Determines the number of spawn positions that will be sampled when the reload button in 3 is clicked.

Tex. displacement The coordinates from where the Perlin Noise sample will be extracted.

Tex. scale Determines how much zoom is applied to the sample of Perlin Noise.

Min./Max. value Both determine in the range $[0, 1]$ which are the limit values of the spawn map. A minimum value greater than 0 will imply that any position as a possibility of being chosen as a spawn position. If the maximum value is set lower to the minimum value, then the colored blobs will be inverted.

Dispersion Determines how blurry the blobs are at their bounds.

Threshold Determines how much interconnection there is between blobs.

5.– Button used to go to the previous configuration window, as shown in Figure E.2.

6.– Button used to start running the simulation.

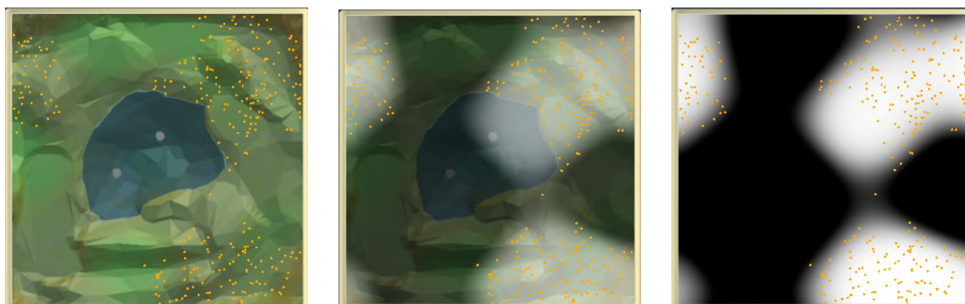


Figure E.3: Visualization of the effect of the spawn map slider.

Figure E.4: Windows used to configure the entities of a new simulation in the application.

The *Simulation Execution* window shown in Figure E.5 has some panels that presents the user the statistics of the progress of the simulation that is running.

1.– Panel that shows the basic statistics of the progress of the simulation that is running. From top to bottom:

Screenshots Number of how many snapshots have been taken over the total. The last snapshot is saved once the exit button is pressed.

Frames Number of how many frames have been computed over the total

Current time The current simulation time that the simulator sits in.

Max. Time The maximum simulation time that the simulator will run for.

Expected duration The expected execution time remaining (measured in real time), calculated as a function of the FPSs of the previous 10 frames computed.

Frame duration Average of the execution time of the previous 10 frames.

Frames per second Average FPS for the previous 10 frames.

2.– Panel used to toggle on and off the color channels of the texture shown in panel 3. Each toggle tick has a color that corresponds to the color they control. Color red displays the root radius of each plant. Color green displays the proportional soil available per pixel for a single plant, averaged by how many plants have access to said pixel. The blue channel displays the crown radius of each plant. Rendering this texture has no additional cost, as it is used internally to calculate the plants' requirements.

3.– Panel that displays the map of the world. It can have overlaid the plants tracker texture or the position of each entity, or both.

4.– Progress bar that represents the percentage of completion of the simulation, along with a percentage number, based on how many frames have been computed.

5.– Play/pause buttons, used to stop or resume the computation of the frame.

6.– Button used to save the snapshots, calculate the statistics, and exit the simulator. It can be pressed even if the simulation has not finished, as an early stopping method.

7.– Panel with a toggle for each entity type, used to enable/disable displaying the position of entities with a marker on panel 3. Rendering the position of entities has an additional computational cost, and it is recommended to disable it for faster executions. However, it is useful to have a glance at how the entities are behaving and how many of each type are left.

8.– Button used to access the world in the state of last frame that has being computed. It will only be visible when the simulator is paused.

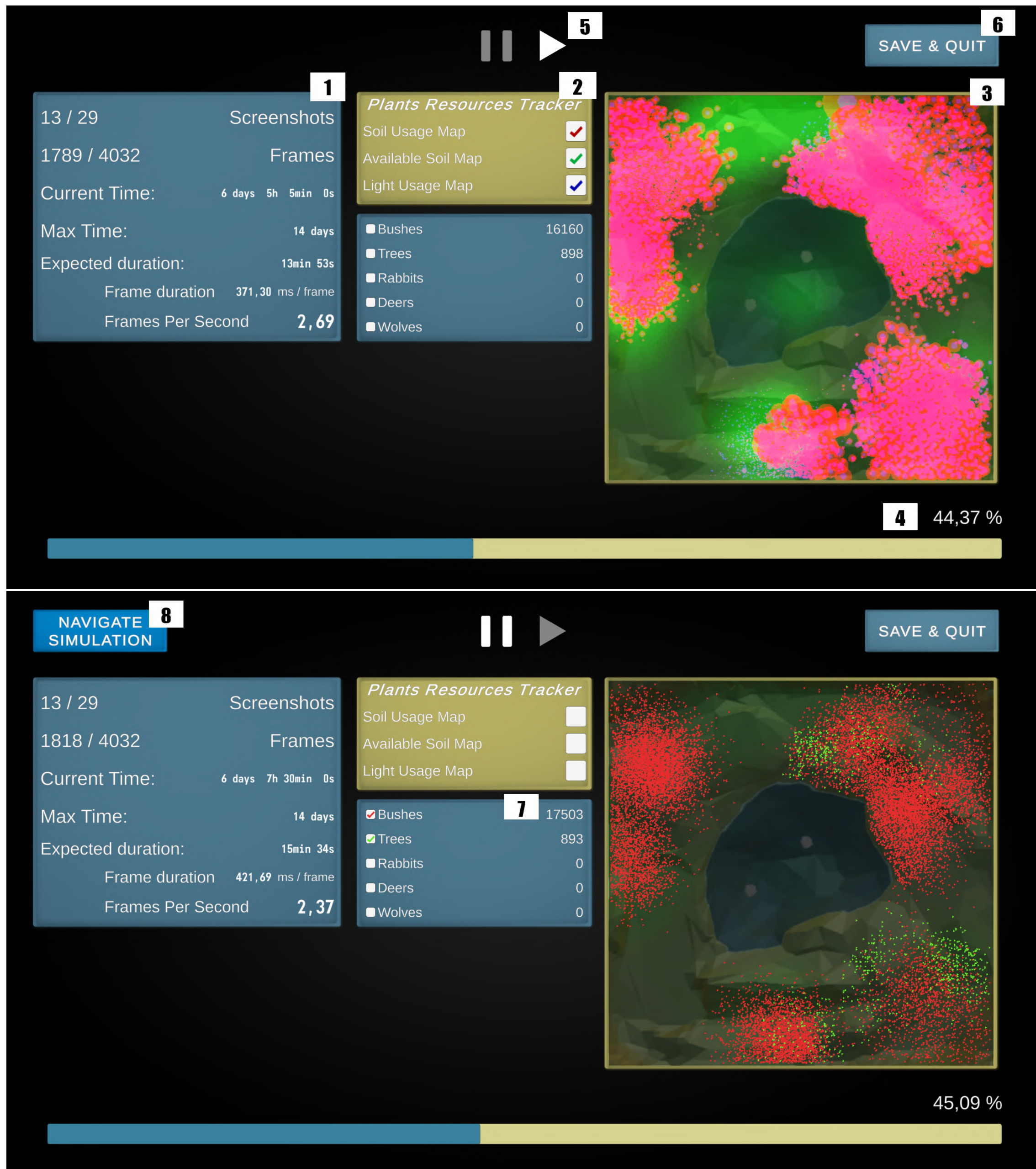


Figure E.5: Window used to show the progress of a running simulation.

The user can access its saved simulations through the *Load Simulation File* window. In this window, each panel has a collection of information used to identify the contents of the save file.

1.– The 'X' button is used to go back to the *Main Menu* window. The reload button will reload the panels. This might be necessary if the user manually handles the files from disk, or if a thumbnail is not loaded and a refresh is required. The trash can button toggles on and off the delete mode. When the delete mode is active (noticeable by a red heartbeat effect over the whole screen), the file panels that are clicked will be deleted from disk instead of loaded.

2.– Default look for a file panel. It displays the name and date when the simulation was saved, along with a thumbnail. If no thumbnail has been saved, a default image that shows an empty scene will be shown.

3.– A file panel that is being hovered with the mouse will display additional information, regarding the duration of the simulation in simulation time, the number of saved snapshots, and the entities involved in the simulation. If a file panel is clicked, the application will load the respective save file and redirect the user to the *Simulation Results* window.

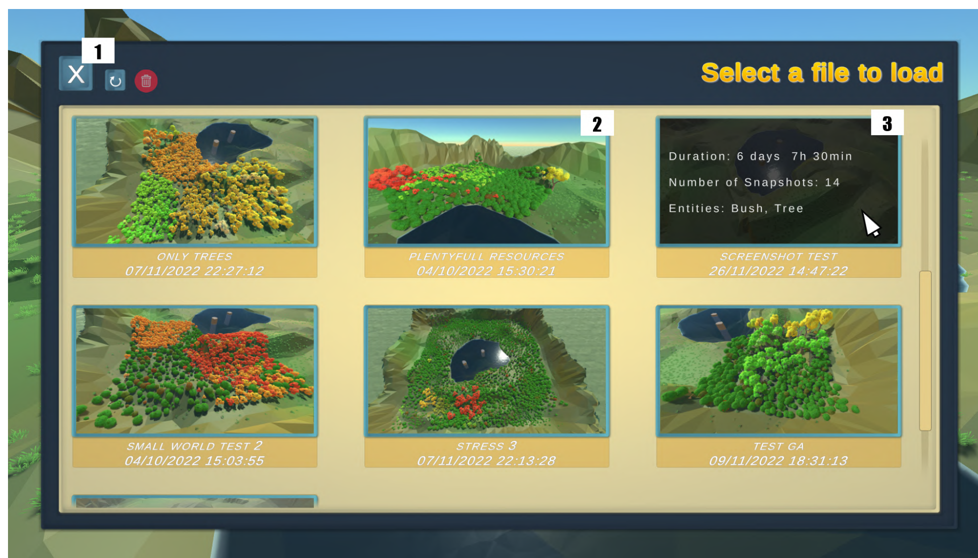


Figure E.6: Window used to load a saved simulation.

The loading screen shown in Figure E.7 will appear when a heavy task is being carried out. It displays a message in the center of the screen explaining said task. It also shows an animation.

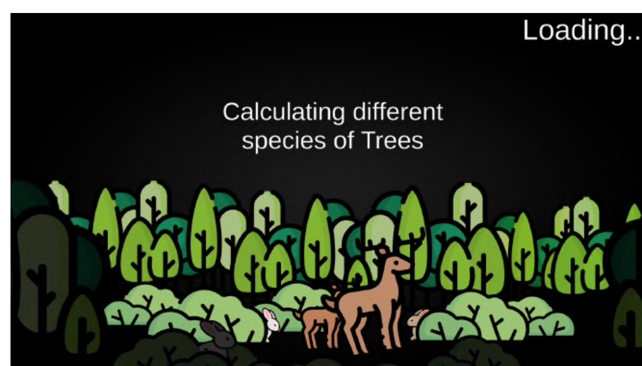


Figure E.7: Loading window used to show the user that a heavy task is being carried out.

The *Simulation Results* window displays all the statistics generated and gathered from a simulation. It provides two panels to compare entities one to one. The Figure E.8 can be explained as follows:

1.– Button used to go back to the *Load Simulation File* window.

2.– Snapshot slider, used to select a snapshot to display its statistics.

3.– Button used to navigate the simulation by loading the snapshot selected with 2.

4.– Entity type selector buttons, used to select which entity types will display their statistics in each panel. The entities that were not present in the simulation are disabled.

5.– Panel button used to display the snapshot heat map. Each dot in the map represents an entity. A brighter pixel in the map implies a denser cluster of entities. A gray dot implies that there is an entity that has not been categorized in any species.

6.– Legend, dynamically changes depending on which statistic is being displayed in the center panel. It can display the species root names with their associated colors, or other relevant information like the color gradient for global heat maps or minimum and maximum values for plotted genes.

7.– The first field displays the number of species and subspecies that are present in the selected snapshot.

8.– Panel button used to display the global heat map, in which every entity of every snapshot is accounted. Reddish areas imply a denser concentration of entities in the simulation over time.

9.– Panel button used to display the evolution of the values of a specific gene of the population as a species tree. Each line represents the average value of a species, and the shaded area the standard deviation. When no species has been detected, a grey line represents the whole population of entities of the specific entity type. The minimum and maximum values the genes can take are also plotted as black horizontal lines. The plot is generated dynamically, adjusting the number of ticks and the vertical axis name, units, and scale.

10.– Panel button used to display the evolution of the values of a specific property. Each entity type has its specific properties. This plot is generated in the same way as the previous one.

11.– Dropdown used to select either which gene or which property is displayed in the panel.

While navigating a simulation, the user will see the scenery with some UI, as portrayed in Figure E.9. Here, the user may control the camera with either the arrow keys or 'wasd' to move around, and the mouse to rotate. By pressing the 'Shift' key the movement will be boosted. The GUI listed in Figure E.9 can be toggled on or off, and has the following usage:

1.– Button used to go back to the previous window.

2.– Help panel that explains some additional controls and features of the navigation mode.

3.– Mini-map to show where the camera is located and looking. It also counts on four default camera positions that, if clicked, will teleport the camera to said positions.

4.– The camera icon is a button that will capture the scene (without the GUI) and use it as a thumbnail for the save file. The slider determines the Field of View (FOV) of the camera in degrees.

5.– Panel used to see the value of the genes or properties of a specific entity. Each of these fields is also presented in the Simulation Results window. The entity that is selected is highlighted with a rainbow outline. If no entity is selected this panel will be hidden. To deselect an entity, press the 'F' key while not looking an any entity.

6.– Dropdown used to select between displaying either genes or properties in panel 5.

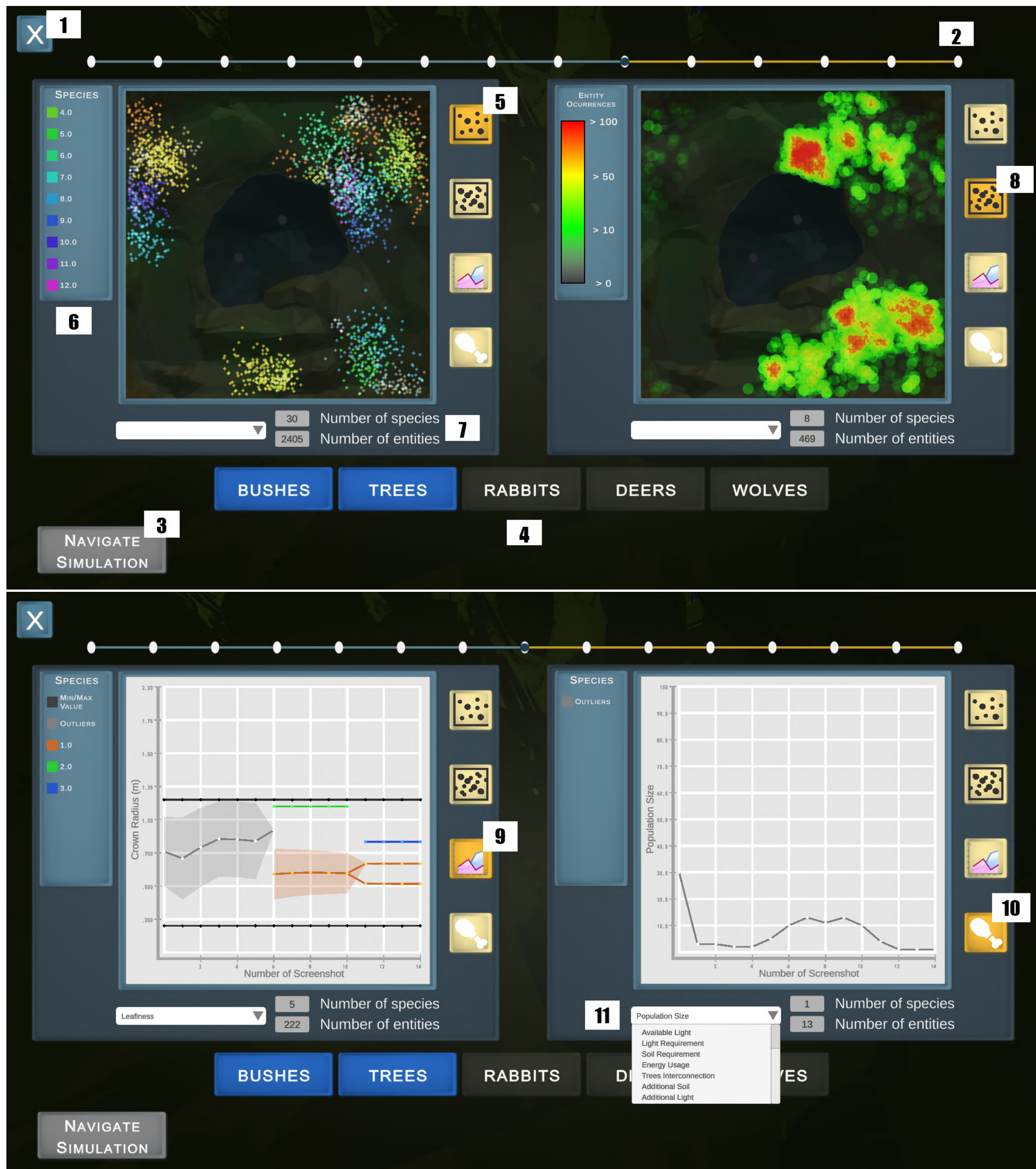


Figure E.8: Window used to view the results of a simulation.

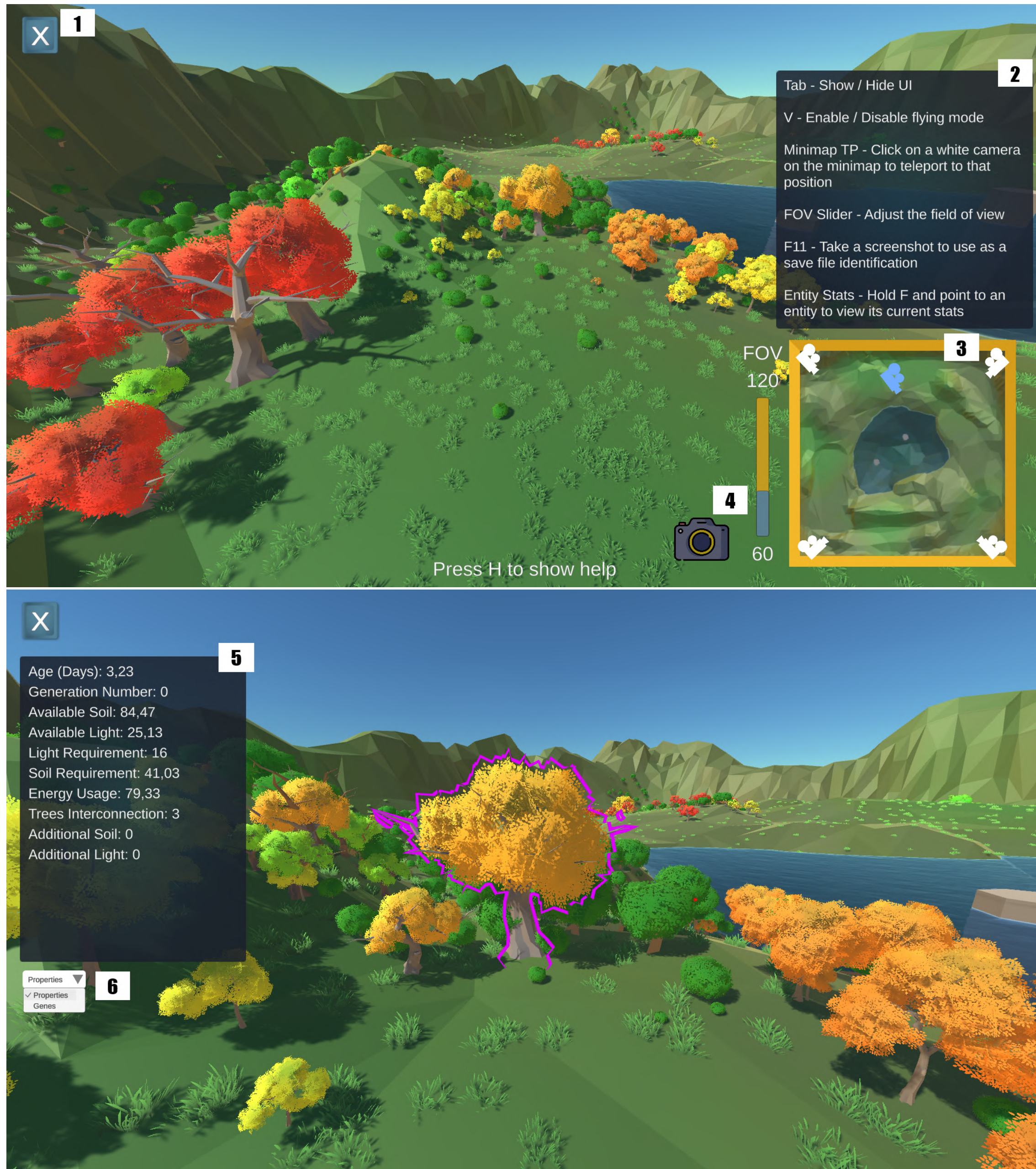


Figure E.9: Scene used to view the world of a simulation in a specific instant.

When configuring an experiment with the Genetic Algorithm, the user will see the *GA Configuration* window, as shown in Figure E.10. All the configurations panels that are not tagged in the figure are already explained in the *New Simulation Configuration* window. However, it is important to note that during an experiment with the GA, no statistics will be gathered, so all the snapshots settings are not present in the *Main Settings* panel. The tagged elements can be explained as follows:

1.– Button to go back to the Main Menu window.

2.– Entity buttons selector, which works in the same way as explained in the *New Simulation Configuration* section. However, in this case the spawning of entities cannot be controlled by the user, and follows a complete random spawn map.

3.– Panel that contains all the configurable options of the Genetic Algorithm. As the panel does not fit in a single screenshot, the remaining fields are shown in Figure E.11. From top to bottom:

Population size Number of chromosomes that will run a simulation for each generation or epoch.

Crossover method Dropdown used to select how chromosomes will generate their descendants. An explanation of each available crossover method is provided in Section 4.2.3.

Crossover prob. Determines the expected percentage of chromosomes in a new population that are descendants of other chromosomes. If the evaluation of this probability is false, then the selected chromosomes will be copied to the new generation.

Mutation prob. Determines the expected percentage of chromosomes that will suffer a mutation when creating a new population.

Elitism proportion Determines which percentage of the population with the highest score will be transferred to the new generation without suffering either crossovers or mutations. A proportion greater than 0 guarantees that at least the best chromosome will be saved by elitism.

Max. Epochs Determines the maximum number of generations that will be computed if none of the rest of stop conditions are met.

Max. score prop. The experiment will finish if the best chromosome has a fitness score equal or higher to this proportion. A proportion greater than 1 will make this stop condition unreachable, hence disabling it.

Mean score prop. The experiment will finish if the average fitness score of the population is equal or above this threshold. A proportion greater than 1 will make this stop condition unreachable, hence disabling it.

4.– Button used to start the experiment with the GA with the provided configuration.

Figure E.10: Window used to configure the GA for an experiment.

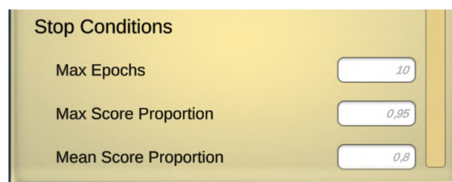


Figure E.11: Configurable stop conditions for the GA.

The *GA Execution* window portrayed in Figure E.12 reflects the progress of an experiment with the Genetic Algorithm. It shows the evolution of the score and the average configuration that the chromosomes of the population encode. Its elements can be defined as follows:

1.– Panel that shows the basic information of the progress of the execution of the experiment. From top to bottom:

   Epochs Displays the epoch number that is being computed over the total number of epochs that will run if no other stop condition is met.

   Chromosome Evaluation Displays the number of chromosome that is being computed of the population over the size of the population.

   Current Time Displays the simulation time the current running simulation is on.

   Max Time Displays how long each simulation will run for in simulation time.

   Expected duration The expected execution time remaining (measured in real time), calculated as a function of the FPS of the previous 10 frames computed.

   Frame duration Average of the execution time of the previous 10 frames.

   Frames per second Average FPS for the previous 10 frames.

2.– Panel that represents the legend for the plot 5 to toggle on and off each individual line. The color of the tick indicates which line it belongs to.

3.– Pause/play buttons, used to toggle the calculation of the active simulation.

4.– Button used to save the best chromosome found so far and exit the experiment. It can be clicked before any of the stop conditions is met, as an early stopping method.

5.– Plot that changes dynamically its number of ticks, and the scale of the vertical axis. When displaying information as an average of the population, the lines in this plot have a shaded area that represents the standard deviation of the population of chromosomes around their average. If a threshold is not used in the experiment as an stop condition, when toggled on it displays a shaded area with its respective color over the grid.

6.– Button used to compress the 'x' axis of the plot 5. It toggles the size between the maximum number of epochs and the current number of epochs computed.

7.– Progress bar and number that represent the percentage of completion of the active running simulation.

8.– Dropdown used to select what information to display about the population. It contains a list of every gene that is involved in the experiment, along with the score option.

9.– This panel serves the same purpose as panel 2, but that has a different set of toggles when the score option is not selected in dropdown 8. In this case, the mutation intensity of a gene is not shown as a percentage, but as the value said percentage represents.

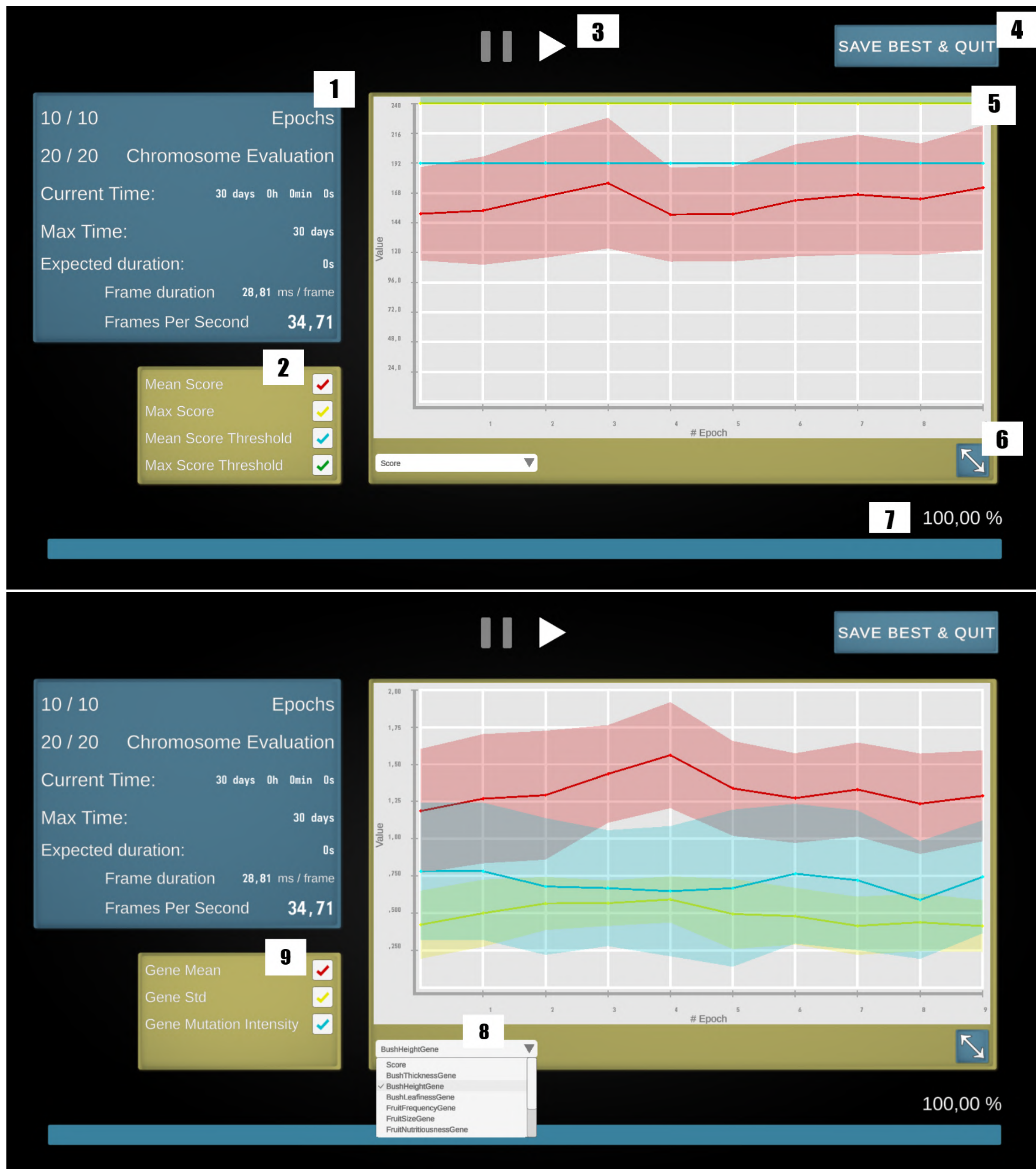


Figure E.12: Window used to show the progress of a running GA.

F

Resources for the Ecosystem Evolution Simulator Application

The following list enumerates other external resources used in the application, among which we can find 3D models, animations, shaders, fonts, icons, source libraries, and extensions of the Unity editor.

- Implementation of HDBSCAN in C# [21].
- Unity editor extension: Serializable Dicts [22].
- Unity Asset Store: Outline effect shader [23].
- Unity Asset Store: Malbers 'Poly Art: Animal Forest Set' (3D models, animations and shaders for animals and the environment) [24].
- Unity Asset Store: 3D low-poly fruits models [25].
- Unity Asset Store: 3D models of trees and bushes [26].
- Basic font used in the GUI [27].
- Monospaced font used for counters [28].
- Icons used in the *Simulation Results* window:
  - Heat Map button icon (Element 5 in Figure E.8) [29].
  - Genes button icon (Element 9 in Figure E.8) [30].
- Icons obtained from creator *Freepik* in flaticon.com [31]:
  - Trash can icon used in the *Load Simulation File* window: https://www.flaticon.com/free-icon/trash-bin_5028066
  - Icons used in the panel to load GA results in the *New Simulation Configuration* window:
    - Bush icon: https://www.flaticon.com/free-icon/bush_2174111
    - Tree icon: https://www.flaticon.com/free-icon/forest_2913483
    - Rabbit icon: https://www.flaticon.com/premium-icon/rabbit_2687160
    - Wolf icon: https://www.flaticon.com/premium-icon/wolf_2315502
    - Deer icon: https://www.flaticon.com/premium-icon/deer_2466079
  - Icons used in the mini-map of the *Simulation Navigation* window:
    - Camera Icon: https://www.flaticon.com/free-icon/camera_1998337
    - Mini-map frame: https://www.flaticon.com/free-icon/frame_595628