CMP-7009A Advanced Programming Concepts and Techniques

Project Report - 20 January 2021

Predator and Prey Ecosystem Simulation

Group members:
Ruari Armstrong, Zhongye Chen, and Łukasz Dziwiszewski

School of Computing Sciences, University of East Anglia

Version 1.0

Abstract

The aim of this project was to develop an ecosystem simulation using Unity engine. The development process started by researching literature covering topics dedicated to building ecosystems, as well as analysing already existing software. Gathered information was then used to create a MoSCoW prioritized list that helped the team focus on the most crucial at the time tasks, consequently increasing chances of delivering a complex software that could be used to predict behaviour of organisms. Following principles of Agile development, work was split into weekly timeboxes, with each being preceded and followed by a team meeting. Amongst the techniques and solutions used to develop the discussed software were: Gaussian Distribution, Box-Muller transform, Cellular Automata, A* Algorithm, Box Blur. In order to increase the simulation's performance, Unity DOTS and ECS were used. Animals' behaviour is controlled by a complex FSM utilising an Enums with Flag attributes. Controlling of the simulation is done through an intuitive UI that allows the user to change settings, modify the parameters, save and load said parameters and data, move the camera around, as well as control the speed of the simulation. Results can be observed on a graph updated in real-time. A set of unit, integration, user, and black-box tests were conducted, helping the team eliminate bugs and increase the software's performance and quality. The development process ended with the team managing to achieve all of the MoSCoW 'Must' goals, as well as most of the 'Should' and 'Could' ones.

Chapter 1

Introduction

1.1 Problem

This project aims to use a 3D game engine, Unity, to develop an ecosystem simulation software. The user needs to be able to observe and gather information on how various species behave in different conditions, as well as when interacting with each other. The use of the developed system would allow for a better understanding of the observed animals, possibly aiding researchers in facing real-life problems.

1.2 MoSCoW

A MoSCoW prioritized list has been used in the development process of this project. It is a well-known technique used to determine which elements of the software the team should focus their efforts on. The principle behind it is to split the previously gathered requirements into four groups: Must, Should, Could, and Won't. 'Must' represents features that are mandatory and need to be included in the final product; 'Should' describes those that are not vital but would significantly increase the quality of the software if ignored; 'Could' will not impact the quality drastically; 'Won't' are not going to be included in this particular development cycle [ProductPlan, 2020].

The MoSCoW prioritized list created by the team can be seen in Table 1.1. Its contents were agreed on after conducting research and analysis of existing work in this field.

Based on the information found it was decided that for the ecosystem simulation to perform its most basic function, at least 2 species following a simple behaviour model would be needed. Said behaviour would include moving around the simulation field, looking for and consuming one source of nutrition, as well as aging and dying. The purpose of including simple UI elements in the 'Must' category was to provide the user with the controls needed to operate the simulation.

'Should' contains features that expand on the aforementioned elements, aiming to add more realism and complexity to the software. It includes another nutrition source, additional behavioural patterns, as well as further improvements to the way the system can be operated.

'Could' mainly includes features that were thought to be worth implementing only after ensuring most of the goals set in other categories were met. Animals' ability to mutate throughout the generations can be used as an example, with the feature undeniably adding more complexity to the simulation, but being impossible to implement without a working mating system.

The machine-learning algorithm in the last category was agreed to be too difficult to implement considering the amount of time available for this project.

MoSCoW Category	Requirements
Must	• Vegetation randomly spawned on the map • Herbivores walking randomly and eating grass • Carnivores eating herbivores when hungry • Creatures dying of old age or hunger • Fast-forward system • Path-finding algorithm • UI with an option to load the map and basic parameters, as well as showing current number of animals
Should	• Bodies of water and thirst system • Herbivores running away from danger • Ability to mate (animals divided into genders and mating/breeding stat) • Loading the map from a text file • Adding more individuals mid-session • Grass growing realistically (possibly using cellular automata) • Stats changing with age
Could	• Mutations • More species • More interesting and diverse maps (grass, sand, mud, lakes, rivers, hills, mountains, etc.) • UI with an option to modify species parameters • Flock algorithm • Simple weather system • Procedurally generated maps
Won't	• Machine learning algorithms

Table 1.1: MoSCoW prioritized list

1.3 Report structure

The remainder of this report is structured as follows:

- **Chapter 2** - includes the team's review of the ecosystem literature, as well as an analysis of the existing software.
- **Chapter 3** - describes various methods used in this project, including: gaussian distribution, cellular automata, A* algorithm, box blur. Also contains snippets of the code used.
- **Chapter 4** - contains the description of how certain elements of the system were implemented.
- **Chapter 5** - offers an overview of the testing that was done to ensure that both the front-end and back-end of the product perform at a sufficient level.
- **Chapter 6** - concludes the work done and puts it in perspective with the MoSCoW analysis conducted at the start of the development cycle. It also discusses what future work could be done and how the product could be further improved and developed.

Chapter 2

Background

2.1 Introduction

The aim of this background review was to analyse the existing literature and scientific resources surrounding the topic of developing ecosystem simulations, as well as to take a closer look at already existing solutions. While conducting the research the focus was placed on finding out what elements are crucial for the end-product to sufficiently imitate realistic scenarios, but also to understand what are the characteristics of the existing software: what are the similarities and differences.

2.2 Findings

2.2.1 Literature review

As stated by [Belew, 2018], one of the factors that have to be considered when designing a system is choice of the right model. One of the models is an individual-based model. It is argued that it can help formulate general theories of ecology [Gras et al., 2009]. However, it is also said that the difficulty of using such model is high, compared to simulating physics individuals possess properties that atoms do not [Grimm and Railsback, 2006]. When implementing individual-based model, it should be remembered to allow the organisms to recognise each other, and to differentiate various objects around them [Ward et al., 2001]. The same paper discusses a possibility of using complex sensory systems along with neural networks in an attempt to develop evolutionary and schooling behaviour amongst the individuals.

As for the simulation environment, 2-dimensional grid seems to be a popular choice [Minar et al., 1996]. It is suggested best practice to use an object-oriented approach along with capabilities of encapsulation and inheritance in order to create agents living in the simulation space; each of them should be represented by a separate object.

2.2.2 Software review

One of the existing projects that was analysed in order to develop the ecosystem simulation discussed in this report was EcoSim [EcoSimIBM and ScottRyanD, 2018]. In their approach, the authors have created a system where two types of individuals interact with each other: prey and predators. Animals possess acquired and inherited traits that change as they age and act; for instance, young and old individuals have their Strength trait set to a lower value. During their lifetime, prey regain energy and sate hunger by consuming grass, whereas predators achieve that by hunting for prey or scavenging their remains. Moreover, each action taken by the individual spends a certain amount of energy, with the amount spent being dependant on their traits, as well as empirically-determined physiological scaling rates. Other than looking for source of nutrition, animals are able to evade predators, socialize, move to the closest cell with an individual of the same species, explore, rest, reproduce. When giving birth, offspring receives genomes that are a combination

of its parents genomes with some possible mutations. Amongst many other elements present in this software, there are: sensory mechanisms; behaviour patterns dictated by fear; advanced sensory options available to predators; unsuccessful reproductive sessions between incompatible individuals; direct and indirect interactions between individuals. Initial values for state variables are read from a text file, and the data collected during the run is stored in an HDF5 file.

Comparable elements can be found in the project created by [Lague, 2019]. The simulation designed by this author focuses on an interaction between two species, with the animals of choice being foxes and rabbits. Other than wandering around, animals are able to look for food, water; flee from danger; and reproduce. Similarly to the previously described system, this one also features sets of traits that change with age, as well as a possibility for offspring to inherit traits from their parents. The system also features a desirability value used when looking for a potential mate.

While the simulation developed by [EcoSimIBM and ScottRyanD, 2018] is too complex and would require the team to work for much longer than possible given time constraints, it was decided that it would be possible to build a system suitable for the purposes of this project by combining findings observed in both of the aforementioned simulations.

Chapter 3

Methodology

3.1 Gaussian Distribution

A normal distribution was used to simulate the litter size of animals in order to better meet the ecological environment changes. The normal distribution has the distribution of continuous random variables with two parameters μ and σ^2. The first parameter μ is the mean value of the random variable that obeys the normal distribution, and the second parameter σ^2 is the variance of this random variable, so the normal distribution is denoted as N(μ, σ^2)[Eugene et al., 2002].

The normal distribution is the most widely known and used of all distributions. Because the normal distribution approximates many natural phenomena so well, it has developed into a standard of reference for many probability problems [Doak et al., 1998].

3.1.1 Box-Muller transform

Box-Muller transformation is a random number sampling method used to generate Standard Normal distribution random numbers [Box, 1958]. Two independent random numbers U1 and U2 come from a uniform distribution (ranging from 0 to 1) and the below transformation is then applied to obtain two new independent random numbers; generated using Gaussian distribution, with the mean being 0, and the standard deviation being 1.

$$Z_0 = R\cos(\Theta) = \sqrt{-2\ln U_1}\cos(2\pi U_2) \tag{3.1}$$

$$Z_1 = R\sin(\Theta) = \sqrt{-2\ln U_1}\sin(2\pi U_2) \tag{3.2}$$

3.2 Cellular Automata

Map generator uses the Cellular Automata in order to create structures similar to caves or natural environments. Randomly generated maps are more in-line with natural environments, and can be used to analyse the impact that the environment can have on the simulation [Lague, 2018].

A Cellular Automaton is a model of a system of 'cell' objects with the following characteristics [Shiffman et al., 2012]:

- Each unit has several units adjacent to it; which are usually defined as this unit's area.
- Looping through all units in the grid is based on their area, and the order of looping is generally from left to right, from top to bottom.

An example of Cellular Automata can be seen in Figure 6.1.

3.3 A* Algorithm

A* is a path searching algorithm first published in 1968, and is an extension of Dijkstra's algorithm [Hart et al., 1968]. A* uses heuristics, functions to rank alternatives in search algorithms, to achieve better performance than Dijkstra.

3.3.1 Description

A* calculates a lowest-cost path tree from the start node to the target node. A* uses a function $f(n)$ that returns the estimated total cost of a path using the node n. The total cost of a path to a node is calculated using $f(n) = g(n) + h(n)$ where $f(n)$ (fCost) is the total estimated cost, $g(n)$ (gCost) is the cost so far to reach node n, and $h(n)$ (hCost) which is the estimated cost from n to the goal node using an admissible heuristic function. An admissible function is where the cost estimated to reach the goal cannot be overestimated, in that the cost is not higher than the lowest possible cost from the current node [Norvig and Intelligence, 2002].

$h(n)$ is admissible if, $\forall n$, $h(n) \leq h^*(n)$ where $h^*(n)$ is the optimal cost to reach a goal from n.

Algorithm 1 A* algorithm

Require: Node: start, Node: goal, function: heuristic

```
1:  openSet = start;
2:  closedSet = empty;
3:  start.gScore = 0;
4:  start.fScore = heuristic(start);
5:  while openSet is not empty do
6:    current = node in openSet with lowest fScore;
7:    if current = goal then
8:      return reconstruct_path(closedSet, current);
9:    end if
10:   openSet.Remove(current);
11:   for all neighbour of current do
12:     tentative_gScore = current.gScore + distance(current, neighbour);
13:     if tentative_gScore < neighbour.gScore then
14:       closedSet[neighbour] = current;
15:       neighbour.gScore = tentative_gScore;
16:       neighbour.fScore = neighbour.gScore + heuristic(neighbour);
17:       if neighbour not in openSet then
18:         openSet.Add(neighbour);
19:       end if
20:     end if
21:   end for
22: end while
23: return failure;
```

Algorithm 2 A* Reconstruct Path function

Require: Map(Node): closedSet, Node: current

```
1: total_path = current;
2: while current in cameFrom.Keys do
3:   current = cameFrom[current];
4:   total_path.prepend(current);
5: end while
6: return total_path;
```

3.4 Box Blur:Grid

A box blur is a simple linear filter with a square kernel where all kernel coefficients are equal. It is the computationally quickest blur algorithm but lacks the smoothness of other blurs. Applying a box blur multiple times can approximate a Gaussian blur [Lukin, 2007].

3.4.1 Description

The naïve box blur algorithm iterates through each position in the array and averages all the neighbouring positions up to a given radius away from the original position. The Naïve approach, seen in Algorithm 3, has a time complexity $O(n) = O(Nr^2)$ where N is size of the array, r is radius of the blur.

A common improvement over Naïve is to split the averaging into two passes, a vertical and horizontal pass over the array. The second pass takes the results from the first pass, this removes the nested inner loop [Jarosz, 2001]. The Vertical and Horizontal Pass approach, seen in Algorithm 4, has a time complexity $O(n) = O(N2r)$ where N is size of the array, r is radius of the blur.

A further optimisation of the box blur is to treat the box blur as a 2D moving average. The previous array position's calculation value is taken, the leftmost position from the previous average is removed and the rightmost position of the current position is added. The result is then averaged for the current position's value [Amrit, 2013]. The Accumulation approach, seen in Algorithm 5, is independent of radius and has a time complexity $O(n) = O(N)$ where N is size of the array.

Algorithm 3 Naïve Box blur algorithm

```
1: for all row, in array do
2:   for all col, in array do
3:     count = 0;
4:     weightsum = 0;
5:     for all pixel j, in the radius in x direction do
6:       for all pixel k, in the radius in y direction do
7:         weightsum += weight(j,k);
8:         count ++;
9:       end for
10:    end for
11:    array[row,col] = weightsum/count;
12:  end for
13: end for
```

Algorithm 4 Vertical and Horizontal Pass Box blur algorithm

```
1: for all row, in array do
2:   for all col, in array do
3:     count = 0;
4:     weightsum = 0;
5:     for all pixel p, in the radius in x direction do
6:       weightsum += weight(p);
7:       count ++;
8:     end for
9:     array[row,col] = weightsum/count;
10:  end for
11: end for
12: for all row, in array do
13:  for all col, in array do
14:    count = 0;
15:    weightsum = 0;
16:    for all pixel p, in the radius in y direction do
17:      weightsum += weight(p);
18:      count ++;
19:    end for
20:    array[row,col] = weightsum/count;
21:  end for
22: end for
```

Algorithm 5 Accumulation Box blur algorithm

```
1: for all row, in array do
2:   count = 0;
3:   weightsum = 0;
4:   for all pixel p, in the radius of 1st pixel in x direction do
5:     weightsum += weight(p);
6:     count ++;
7:   end for
8:   array[row,col] = weightsum/count;
9:   for all remaining pixels p, in row do
10:    array[row,col] = weight(p) - weight(p-radius) + weight(p+radius);
11:  end for
12: end for
```

Chapter 4

Implementation

4.1 Unity DOTS and ECS

It was decided that the simulation would utilise Unity's high-performance, multi-threaded Data-Oriented Technology Stack (DOTS) [Unity, 2020]. This was chosen due to the potential for huge performance gains when running many animal instances through the Entity Component System (ECS), as opposed to Unity's traditional object-oriented MonoBehaviours.

The ECS is the core of DOTS and comprises three elements:

Entities represent the individual 'things' in the simulation such as flora, fauna, and tiles. Entities have no behaviour or data, but identify what pieces of data belong together to make up a 'thing'.

Components represent the data in the simulation such as movement data, size data, or state data. Entities index collections of components.

Systems are the logic or behaviours that transform component data such as updating the world position of all entities with a movement data component by the movement speed multiplied by the delta time since the last update.

4.1.1 State System

A Finite-State Machine model (FSM) was used to design the behaviour model of the entities present in the simulation. The characteristics of the FSM are that it consists of a limited numbers of states, with an option to transition from one state to another when a certain input is received. In this program, a high level of hunger is an input transitioning an animal to the 'Hungry' state. The initial design of the Finite State Machine used in this project can be seen in Appendix 6.2.

In order to implement such a system, C# Enum Flag Attributes were used and can be seen in Appendix 6.3. With the complexity of the developed simulation, each entity can be in a state consisting of several 'sub-states'; for example, a fox can experience being both hungry and thirsty at the same time, meaning it would not ignore sources of food or water when coming across them. Achieving such complex states is possible by using Enum Flags along with bitwise operators, as this allows the enumeration variable to store multiple values. For instance, a bitwise operator '&' can determine if a prioritized 'Fleeing' is on the list of currently triggered 'sub-states', and in case it is, the animal will enter the routine of running away from danger. Similarly, '|', '~', and '^' are used to set, unset, and toggle the flags respectively.

What 'sub-states' are currently toggled on determines the overall state of the entity, and as the result its current behaviour. Moreover, some 'sub-states' take priority over others, further improving the accuracy of the system; for example, 'GivingBirth' takes

priority over 'Fleeing', which in turn is more important than 'Drinking' or 'Eating', whilst 'Dead' is the most important of them all.

4.1.2 A* Pathfinding

Pathfinding was used for the fauna in the simulation to calculate the path to be taken to their target. The process of pathfinding was split into separate subsystems.

A Grid script runs at start up; this creates a 2D array of *GridNode*s which represent the nodes that the pathfinding algorithm uses. Each *GridNode* stores information if the node is walkable, world position, and movement penalty for travelling that node. To smooth the movement penalties an accumulation Box blur is applied with a radius of 3 to the grid. An example of the grid array overlaid on the map can be seen in Figure 6.4.

A path request system was implemented to add a *PathFindingRequestData* component to entities which have a target and are not currently following a path. The component stores the start and end position of the path to be calculated.

The path finding system was created to run on entities with *PathFindingRequestData*, *PathFollowData* components, and a *PathPositionData* dynamic buffer. A *FindPath* job is executed, which implements the A* algorithm to find a path between the start and end position. Once found the path node positions are added to the dynamic buffer, the *PathFollowData* index is set to the length of the buffer -1 and the *PathFindingRequestData* component is removed.

The final system for pathfinding is the path follow system which iterates through each index in the dynamic buffer in reverse order. The entity is moved to each world position before decrementing the index. When index -1 is reached the path has been followed to completion and therefore the entity is at the target position.

4.1.3 Looking Entity System And Targeting System

The *LookingEntity* system is the field of vision of the simulated animal. Animals need to find an appropriate nutrition source based on their Hunger and Thirst levels. However, they can see objects only within a certain range. Setting physical properties for each object comes from the Unity.Physics class. The unit mask[Monkey, 2018] is used as a filter to get the entities they can collide with.

In the case of the ECS system, in order to manage collisions, it is necessary to build a physical world first. In general, the *BuildPhysicsWorld* system is built in the second half of each frame because Physics World requires a lot of memory support.

Unity.Physics has a function called *OverlapAabb* [Randall, 2017] that can return an array of entities containing the collision attribute. The system then identifies the nearest food, water, predator, and mate.

The *Targeting* system tells the entity the location of their destination based on the animal's state and the nearest entity obtained from the *LookingEntity* system.

4.2 UI

4.2.1 Menu

To aid with the creation of the main menu for the simulation an extension asset was used, Full Menu System[Tutor, 2020]. This provided a framework to create a fully interactable menu including window pop-ups, save-able user settings, and starting and exiting the simulation, which can be seen in Figure 6.5. The menu system uses a state machine for switching and displaying menu slides with interactable menu items calling public methods within the *MenuController* script.

The menu system was improved to add more options: altering the default properties for the flora and fauna in the simulation, as seen in Figure 6.6; generating maps using Cellular Automata, as seen in Figure 6.7 and loading maps in Figure 6.8. The sliders and

input fields edit the static fields of the respective *Defaults* class. When an entity of the fauna or flora is created the starting property values are read from these *Defaults* class.

4.2.2 Loading/Saving using XML

XML (Extensible Markup Language) is a markup language used to mark electronic files to make them structured. It can be used to mark data and define data types. It is a source language that allows users to define their markup language. The process of saving the entity's default values can be seen in Figure 6.9. An example of a saved XML file can be seen in Figures 6.10, Figure 6.11, Figure 6.12 and Figure 6.13. An example of loading an XML file can be seen in Figure 6.14.

The *StandaloneFileBrowser*[Gökçe, 2018] class was used to implement a native file browser for the loading and saving of the map and entity properties. Users can give them a name and choose a save location.

4.2.3 Movable Camera

The camera allows the user to zoom-in and zoom-out, and its movements are limited appropriately. For example, when the camera is placed in its highest position, the option to zoom-out is not available anymore. As the camera zooms-in, the user can view entities more clearly. The camera's movement speed is adjusted accordingly.

4.2.4 Time Control System

The time control system gives users the ability to accelerate the simulation, and to observe the evolution of the ecosystem more quickly. Users are also able to slow down time.

4.2.5 Graph

The graph is used to follow the quantity of rabbits, foxes, and grass. It can be dynamically updated in real-time. The X-axis is in-game seconds, the Y-axis is the population. The graph provides an option to observe the general trend which allows users to roughly analyze the population trend over a long period of time.

If the user wants to observe a defined time-frame, the graph provides a second method to allow them to do so. For instance, when the user's input is 120, the graph will display the last two minutes of changes.

The starting maximum value of Y is five times the maximum value of rabbit, fox, or grass. When the population exceeds 80% of the maximum Y value, the maximum value of Y-axis increases accordingly, ensuring that the user can see the change in the value of Y. If the initial value of Y is set to a fixed value, small changes cannot be observed.

Population data can be saved to CSV(Comma-separated value) file that facilitate future analysis. An example of a graph can be seen in Figure 6.15.

Chapter 5

Testing and Results

5.1 Unit Testing

The Unity's unit testing is mainly divided into Play model and Editor model, and they both run in Unity Test Runner which is a tool to test code. The Unity Test Runner uses Unity's integration of the NUnit library, which is an open-source unit testing library for .NET languages. An example of the Unity Test Runner can be seen in Figure 6.16. The object-oriented part of this project is mostly User Interface, so unit test of the Editor model was a better model choice. The time control system and camera control system in this project were tested using NUnit.Framework provided by .NET Framework. Two examples of Unit Testing Code can be seen in Figures 6.17 and 6.18.

5.2 Integration Testing

As part of the weekly round up, integration testing of the systems and MonoBehaviours were performed before merging the various branches that had been changed during the sprint. Primarily Big Bang Integration [Leung and White, 1990] was used due to each member testing their own branch independently beforehand. This reduced the need for developing drivers and stubs to be used to test systems incrementally. If more systems were to be added to the finished simulation Incremental integration testing would be more appropriate.

5.3 User Testing

User testing is a process of testing the product with real users, with the intention of gathering information as to what potential future users think about the product, as well as to see how the product is being interacted with. One of the key components of designing usability tests is knowing what users are thinking [Wildman, 1995]; having that in mind people involved in the testing process were asked to think aloud.

However, because of the limitations imposed on the team by the COVID-19 restrictions, it was decided that members of the team would act both as developers and users. Following that decision, testing was done after new features were finished, which was conducted by the other two team members that had not worked on its implementation.

Throughout multiple testing sessions a lot of valuable feedback was gathered, and many changes were implemented. For example, the initial version of the graph would allow the user to see progression of the graph either from the last 60 or 300 seconds; testing amongst the team showed that such implementation was unintuitive and too restrictive, this limitation was removed.

5.4 Black-Box Testing

To ensure the program performed as intended at a high level multiple runs were performed altering the initial properties to simulate various scenarios.

A scenario of normal conditions with odds skewed in the fox's favour was performed, as seen in Figure 6.19. The rabbit population grew as a result of an abundance of grass, then steeply declined as a result of being eaten and as the grass population reduced to 0. The fox population increased from mating then died off as the rabbit population declined.

Simulating an infinite abundance of food (no hunger increase) with no predators showed a population explosion of exponential growth amongst the rabbits which can be seen in Figure 6.20.

Chapter 6

Conclusion and Future Work

All of the goals set in the 'Must' category of the MoSCoW list have been achieved, as well as 6 out of 7 goals placed in the 'Should' category. Moreover, the project has been finished with nearly half of the 'Could' features being implemented too. The end product not only fulfills the basic requirements needed for it to be used as an ecosystem simulation, but also consists of a set of additional functionalities that make it a desirable tool for conducting more complex and realistic research. Considering those factors, it can be concluded that the project has been finished successfully.

However, whilst nearly all of the most important goals have been achieved, if more time had been allocated to this project, the implemented features, as well as the overall performance of the system could be improved. Furthermore, even though entities have varying attribute values depending on their age, growing up has been simplified to only 3 stages ('Young', 'Adult', 'Old'), and the number of affected attributes also limited to only 3 ('HungerIncrease', 'MovementSpeed', 'Size'). To better reflect real-life behaviour, stages should be replaced with gradual growth having effect on a wider range of attributes. Additionally, it was agreed that UI elements could be presented differently, allowing for an easier and more intuitive use of the system.

In order to improve performance, and allow for more entities to be present on the stage at once, it was decided halfway through the development that a switch to Unity's data-oriented Entity Component System would be beneficial. Although the conversion brought in desirable improvements, the shortage of time prevented the authors of this project from better understanding the specifics of this experimental architecture, failing to fully utilise its capabilities. Consequently, some of the ECS systems require more time to complete their tasks than they should if the system was better optimised.

As some of the goals set in the MoSCoW list have not been met, further development would also focus on adding those missing features.

Bibliography

[Amrit, 2013] Amrit, A. (2013). Understanding box blur. `http://amritamaz.net/blog/understanding-box-blur`.

[Belew, 2018] Belew, R. K. (2018). *Adaptive individuals in evolving populations: models and algorithms*. Routledge.

[Box, 1958] Box, G. E. (1958). A note on the generation of random normal deviates. *Ann. Math. Stat.*, 29:610–611.

[Doak et al., 1998] Doak, D. F., Bigger, D., Harding, E., Marvier, M., O'malley, R., and Thomson, D. (1998). The statistical inevitability of stability-diversity relationships in community ecology. *The American Naturalist*, 151(3):264–276.

[EcoSimIBM and ScottRyanD, 2018] EcoSimIBM and ScottRyanD (2018). Ecosim, an individual-based evolving predator-prey ecosystem simulation. `https://github.com/EcoSimIBM/EcoSim-Default`.

[Eugene et al., 2002] Eugene, N., Lee, C., and Famoye, F. (2002). Beta-normal distribution and its applications. *Communications in Statistics-Theory and methods*, 31(4):497–512.

[Gras et al., 2009] Gras, R., Devaurs, D., Wozniak, A., and Aspinall, A. (2009). An individual-based evolving predator-prey ecosystem simulation using a fuzzy cognitive map as the behavior model. *Artificial life*, 15(4):423–463.

[Grimm and Railsback, 2006] Grimm, V. and Railsback, S. F. (2006). Agent-based models in ecology: Patterns and alternative theories of adaptive behaviour. In *Agent-Based Computational Modelling*, pages 139–152. Physica-Verlag.

[Gökçe, 2018] Gökçe, G. (2018). Standalone file browser. `https://github.com/gkngkc/UnityStandaloneFileBrowser`.

[Hart et al., 1968] Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107.

[Jarosz, 2001] Jarosz, W. (2001). Fast image convolutions. In *SIGGRAPH Workshop*.

[Lague, 2018] Lague, S. (2018). Procedural-cave-generation. `https://github.com/SebLague/Procedural-Cave-Generation`.

[Lague, 2019] Lague, S. (2019). Ecosystem-2. `https://github.com/SebLague/Ecosystem-2/tree/master`.

[Leung and White, 1990] Leung, H. K. N. and White, L. (1990). A study of integration testing and software regression at the integration level. In *Proceedings. Conference on Software Maintenance 1990*, pages 290–301.

[Lukin, 2007] Lukin, A. (2007). Tips & tricks: Fast image filtering algorithms. *jiS*, 12(1):2.

[Małecki et al., 2019] Małecki, K., Jankowski, J., and Szkwarkowski, M. (2019). Modelling the impact of transit media on information spreading in an urban space using cellular automata. *Symmetry*, 11(3):428.

[Minar et al., 1996] Minar, N., Burkhart, R., Langton, C., and Askenazi, M. (1996). The swarm simulation system: A toolkit for building multi-agent simulations. *Santa Fe Institute Working Paper*, 96-06-042.

[Monkey, 2018] Monkey, C. (2018). Layers, layer mask, bitmask, bitwise operators, raycasts. `https://unitycodemonkey.com/video.php?v=uDYE3RFMNzk`.

[Norvig and Intelligence, 2002] Norvig, P. R. and Intelligence, S. A. (2002). *A modern approach*. Prentice Hall.

[ProductPlan, 2020] ProductPlan (2020). What is moscow prioritization?: Overview of the moscow method.

[Randall, 2017] Randall, J. (2017). Introductory guide to aabb tree collision detection.

[Shiffman et al., 2012] Shiffman, D., Fry, S., and Marsh, Z. (2012). *The nature of code*. D. Shiffman.

[Tutor, 2020] Tutor, S. (2020). Full menu system. `https://assetstore.unity.com/packages/tools/gui/full-menu-system-free-158919`.

[Unity, 2020] Unity (2020). Unity dots. `https://unity.com/dots`.

[Ward et al., 2001] Ward, C., Gobet, F., and Kendall, G. (2001). Evolving collective behavior in an artificial ecology. *Artificial life*, 7:191–209.

[Wildman, 1995] Wildman, D. (1995). Getting the most from paired-user testing. *interactions*, 2(3):21–27.

Contributions

40% Ruari Ruari is the member that has contributed the most. Parts of the projects that he was responsible for or heavily involved in include but are not limited to: the path-finding system, along with the introduction of walkable/unwalkable tiles; map loading; converting the core of the existing object-oriented project into the new ECS-based version; main menu; GitHub repository management; proof-reading the final version of the report.

His knowledge of GitHub, and an ability to easily explain complex concepts to the rest of the team have been greatly appreciated. The transition to the ECS-based project was much easier thanks to his efforts.

30% Will Will has been responsible for implementing such features of the system as: time-control; determining what objects are present around the entity; generating new maps using Cellular Automata; plotting graphs showing population progression; saving and loading XML containing current settings; fleeing from danger.

30% Łukasz Łukasz contributed to the group by his detailed project design and planning in the initial stages of the project and by researching the most appropriate engine to use. During development he designed and implemented: the State system, mating systems including spawning new entities with properties inherited from the parent, and hunger and thirst systems. Proof reading and editing of the report was also done by Łukasz.

Appendix A

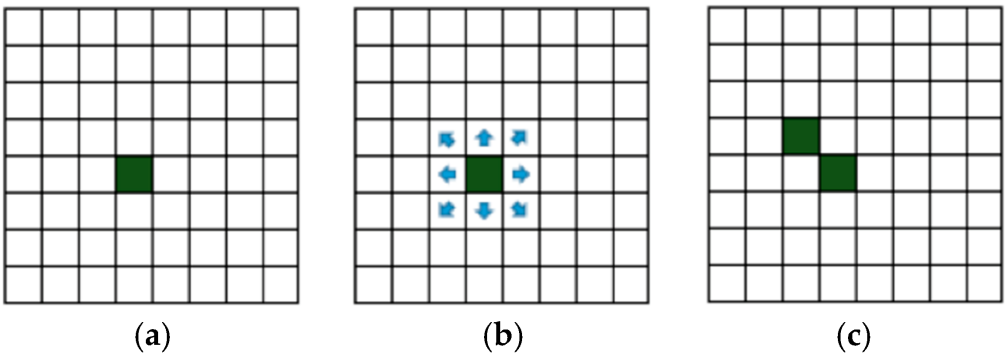

Figure 6.1: A example to explain Cellular Automaton[Małecki et al., 2019]

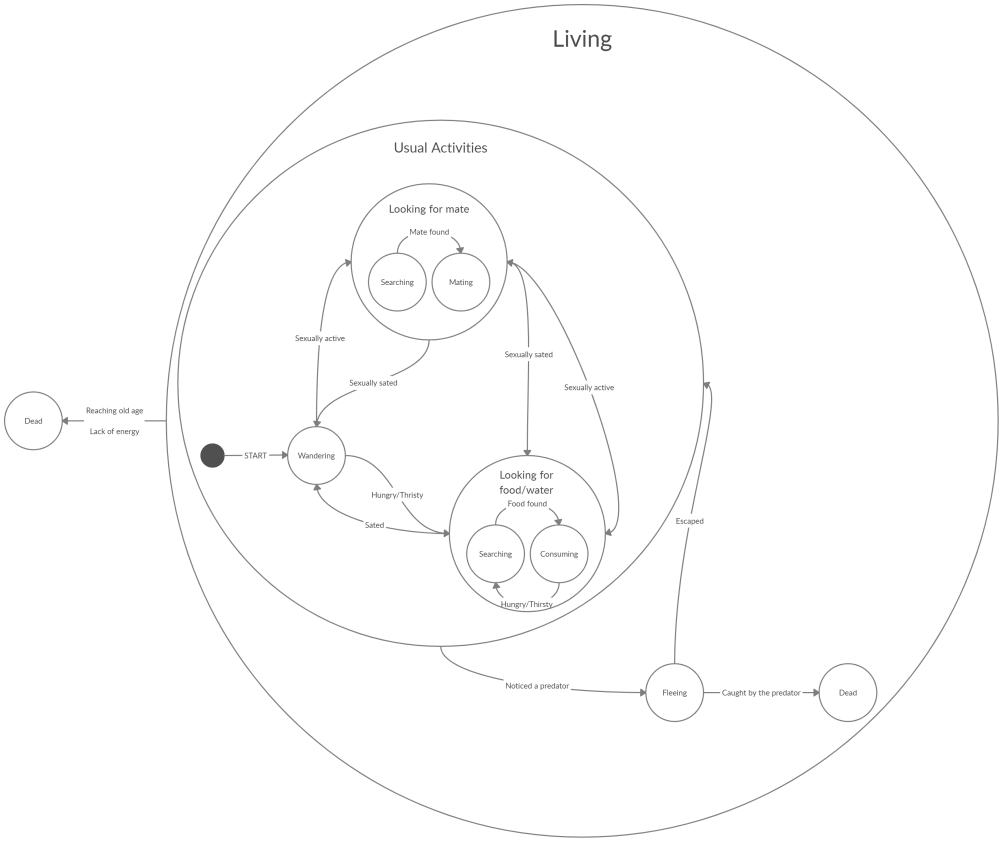

Figure 6.2: A State Transition Diagram representing the initial design of the Finite State Machine used in the simulation.

```
[Flags]
99+ references
public enum FlagStates
{
    None = 0,                // 000000000000
    Wandering = 1,           // 000000000001
    Hungry = 2,              // 000000000010
    Thirsty = 4,             // 000000000100
    Eating = 8,              // 000000001000
    Drinking = 16,           // 000000010000
    SexuallyActive = 32,     // 000000100000
    Mating = 64,             // 000001000000
    Fleeing = 128,           // 000010000000
    Dead = 256,              // 000100000000
    Pregnant = 512,          // 001000000000
    GivingBirth = 1024,      // 010000000000
}
```

Figure 6.3: An Enum with Flag Attribute storing information about an entity's sub-states

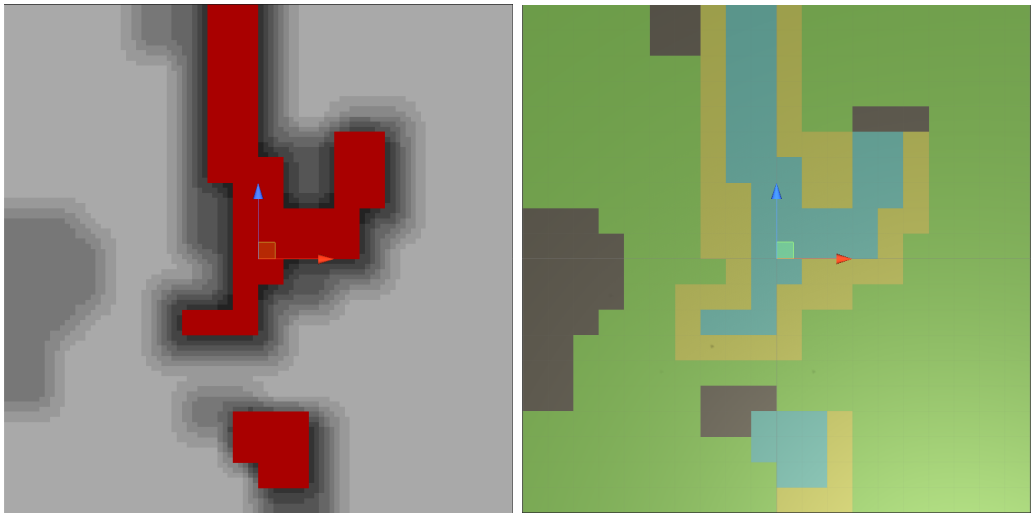

Figure 6.4: Left: grid node overlay, Right: tiles that make up the terrain.

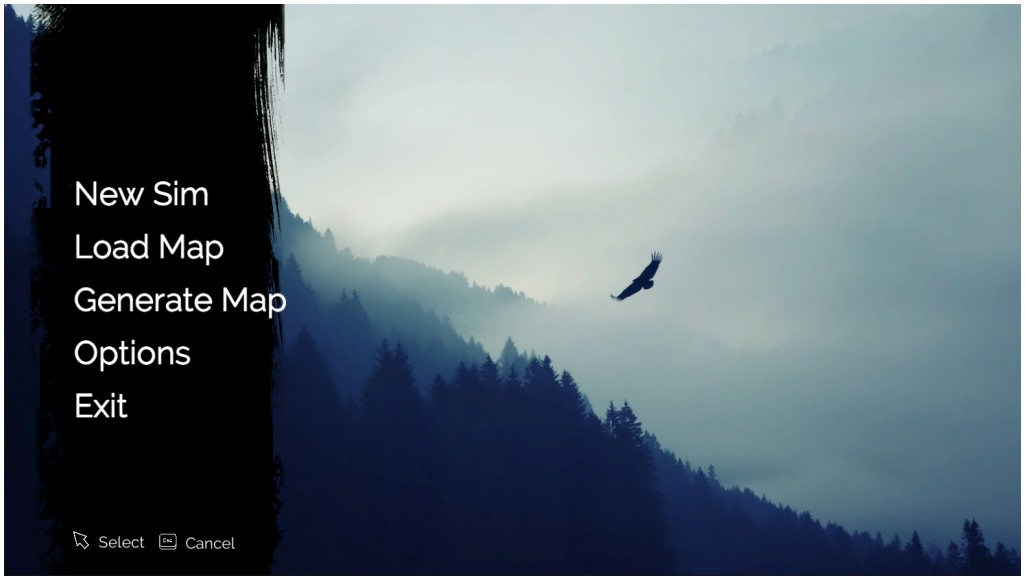

Figure 6.5: Main menu screen.

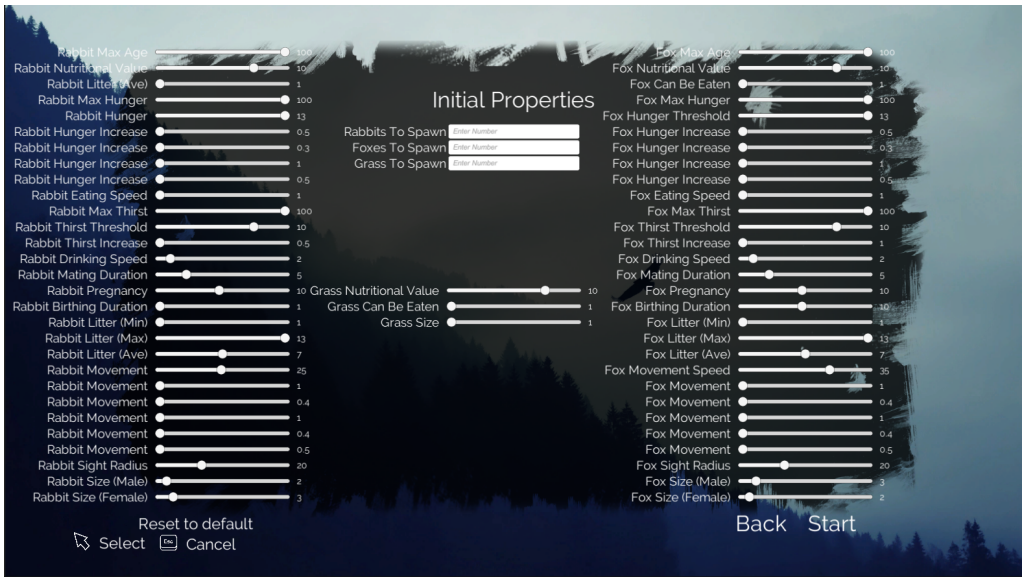

Figure 6.6: Menu screen to change the initial properties of the fauna and flora, and the initial number of each entity to spawn.

Figure 6.7: Map generation menu screen.

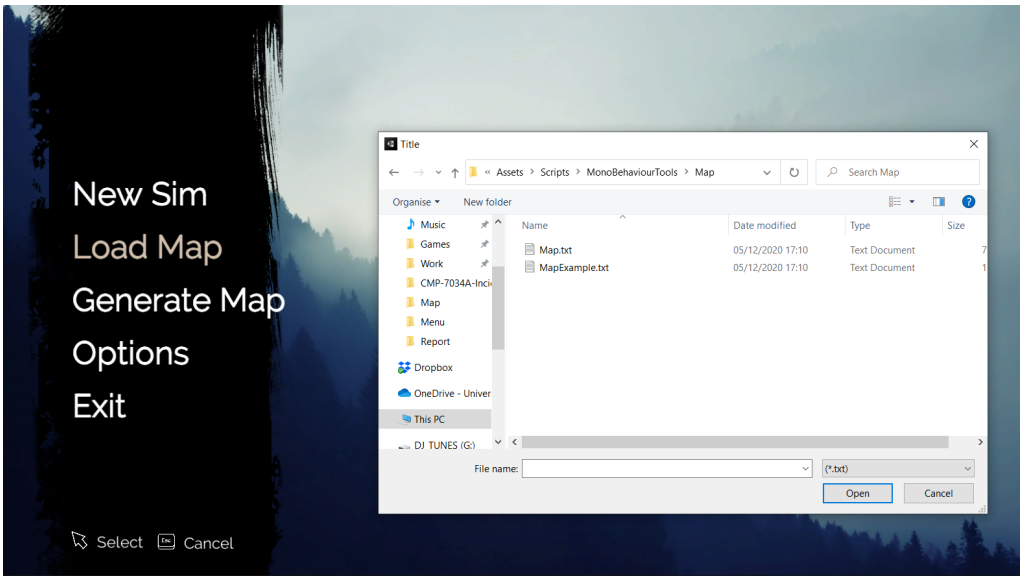

Figure 6.8: Loading map file browser.

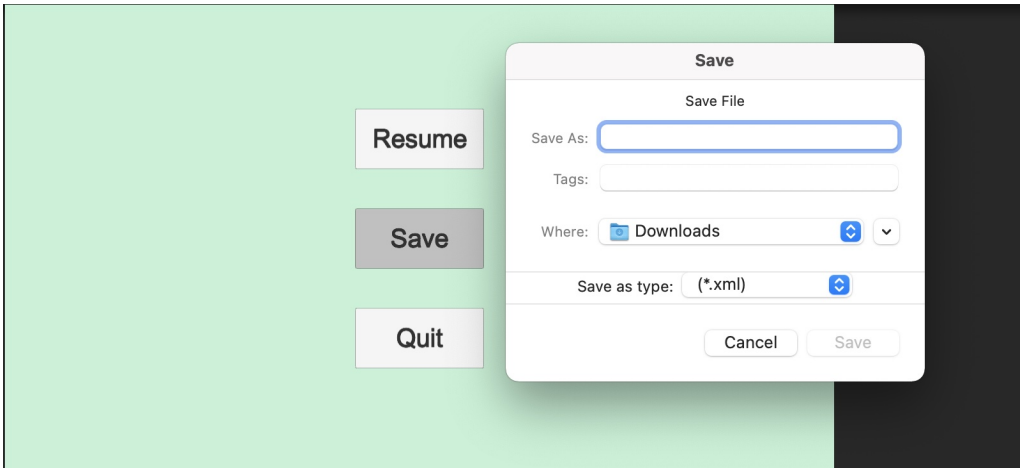

Figure 6.9: Escape Menu And Saving Entity Default Value to XML File

```xml
<XML>
  <Rabbit>
    <AgeData>
      <age>0</age>
      <ageIncrease>1</ageIncrease>
      <ageMax>600</ageMax>
      <ageGroup>Young</ageGroup>
      <adultEntryTimer>20</adultEntryTimer>
      <oldEntryTimer>450</oldEntryTimer>
    </AgeData>
    <edibleData>
      <nutritionalValue>50</nutritionalValue>
      <canBeEaten>True</canBeEaten>
      <nutritionalValueMultiplier>1</nutritionalValueMultiplier>
      <foodType>Meat</foodType>
    </edibleData>
    <hungerData>
      <hunger>0</hunger>
      <hungerMax>100</hungerMax>
      <hungryThreshold>49</hungryThreshold>
      <hungerIncrease>0.1</hungerIncrease>
      <pregnancyHungerIncrease>0.7</pregnancyHungerIncrease>
      <youngHungerIncrease>0.1</youngHungerIncrease>
      <adultHungerIncrease>0.1</adultHungerIncrease>
      <oldHungerIncrease>0.1</oldHungerIncrease>
      <eatingSpeed>1</eatingSpeed>
      <diet>Herbivore</diet>
    </hungerData>
    <ThirstData>
      <thirst>0</thirst>
      <thirstMax>100</thirstMax>
      <thirstyThreshold>49</thirstyThreshold>
      <thirstIncrease>0.1</thirstIncrease>
      <drinkingSpeed>2</drinkingSpeed>
    </ThirstData>
    <mateData>
      <mateStartTime>0</mateStartTime>
      <matingDuration>5</matingDuration>
      <reproductiveUrge>0</reproductiveUrge>
      <reproductiveUrgeIncreaseMale>0.3</reproductiveUrgeIncreaseMale>
      <reproductiveUrgeIncreaseFemale>0</reproductiveUrgeIncreaseFemale>
      <matingThreshold>50</matingThreshold>
    </mateData>
    <pregnancyData>
      <pregnancyStartTime>0</pregnancyStartTime>
      <babiesBorn>0</babiesBorn>
      <birthStartTime>0</birthStartTime>
      <currentLitterSize>0</currentLitterSize>
      <pregnancyLengthModifier>1</pregnancyLengthModifier>
      <pregnancyLength>10</pregnancyLength>
      <birthDuration>1</birthDuration>
      <litterSizeMin>1</litterSizeMin>
      <litterSizeMax>13</litterSizeMax>
      <litterSizeAve>7</litterSizeAve>
    </pregnancyData>
    <movementData>
      <moveSpeed>25</moveSpeed>
      <rotationSpeed>10</rotationSpeed>
      <moveMultiplier>1</moveMultiplier>
      <pregnancyMoveMultiplier>0.5</pregnancyMoveMultiplier>
      <originalMoveMultiplier>1</originalMoveMultiplier>
      <youngMoveMultiplier>0.4</youngMoveMultiplier>
      <adultMoveMultiplier>1</adultMoveMultiplier>
      <oldMoveMultiplier>0.4</oldMoveMultiplier>
    </movementData>
```

Figure 6.10: XML1

```
    <sizeData>
      <sizeMultiplier>1</sizeMultiplier>
      <scaleMale>2</scaleMale>
      <scaleFemale>3</scaleFemale>
      <youngSizeMultiplier>0.6</youngSizeMultiplier>
      <adultSizeMultiplier>1</adultSizeMultiplier>
      <oldSizeMultiplier>0.75</oldSizeMultiplier>
    </sizeData>
    <stateData>
      <flagState>Wandering</flagState>
      <previousFlagState>Wandering</previousFlagState>
      <deathReason>Eaten</deathReason>
      <beenEaten>False</beenEaten>
    </stateData>
    <targetData>
      <touchRadius>1</touchRadius>
      <sightRadius>20</sightRadius>
    </targetData>
    <lookingEntityData>
      <shortestToEdibleDistance>Infinity</shortestToEdibleDistance>
      <shortestToWaterDistance>Infinity</shortestToWaterDistance>
      <shortestToPredatorDistance>Infinity</shortestToPredatorDistance>
      <shortestToMateDistance>Infinity</shortestToMateDistance>
    </lookingEntityData>
    <ColliderTypeData>
      <colliderType>Rabbit</colliderType>
    </ColliderTypeData>
  </Rabbit>
  <Fox>
    <ageData>
      <age>0</age>
      <ageIncrease>1</ageIncrease>
      <ageMax>800</ageMax>
      <ageGroup>Young</ageGroup>
      <adultEntryTimer>20</adultEntryTimer>
      <oldEntryTimer>600</oldEntryTimer>
    </ageData>
    <edibleData>
      <nutritionalValue>50</nutritionalValue>
      <canBeEaten>False</canBeEaten>
      <nutritionalValueMultiplier>1</nutritionalValueMultiplier>
      <foodType>Meat</foodType>
    </edibleData>
    <hungerData>
      <hunger>0</hunger>
      <hungerMax>100</hungerMax>
      <hungryThreshold>20</hungryThreshold>
      <hungerIncrease>0.5</hungerIncrease>
      <pregnancyHungerIncrease>0.7</pregnancyHungerIncrease>
      <youngHungerIncrease>0.3</youngHungerIncrease>
      <adultHungerIncrease>1</adultHungerIncrease>
      <oldHungerIncrease>0.5</oldHungerIncrease>
      <eatingSpeed>1</eatingSpeed>
      <diet>Carnivore</diet>
    </hungerData>
    <thirstData>
      <thirst>0</thirst>
      <thirstMax>100</thirstMax>
      <thirstyThreshold>10</thirstyThreshold>
      <thirstIncrease>0.1</thirstIncrease>
      <drinkingSpeed>2</drinkingSpeed>
    </thirstData>
    <mateData>
      <mateStartTime>0</mateStartTime>
      <matingDuration>5</matingDuration>
      <reproductiveUrge>0</reproductiveUrge>
      <reproductiveUrgeIncreaseMale>0.3</reproductiveUrgeIncreaseMale>
      <reproductiveUrgeIncreaseFemale>0</reproductiveUrgeIncreaseFemale>
      <matingThreshold>50</matingThreshold>
    </mateData>
```

Figure 6.11: XML2

```
  <pregnancyData>
    <pregnancyStartTime>0</pregnancyStartTime>
    <babiesBorn>0</babiesBorn>
    <birthStartTime>0</birthStartTime>
    <currentLitterSize>0</currentLitterSize>
    <pregnancyLengthModifier>1</pregnancyLengthModifier>
    <pregnancyLength>10</pregnancyLength>
    <birthDuration>10</birthDuration>
    <litterSizeMin>1</litterSizeMin>
    <litterSizeMax>14</litterSizeMax>
    <litterSizeAve>5</litterSizeAve>
  </pregnancyData>
  <movementData>
    <moveSpeed>35</moveSpeed>
    <rotationSpeed>10</rotationSpeed>
    <moveMultiplier>1</moveMultiplier>
    <pregnancyMoveMultiplier>0.5</pregnancyMoveMultiplier>
    <originalMoveMultiplier>1</originalMoveMultiplier>
    <youngMoveMultiplier>0.4</youngMoveMultiplier>
    <adultMoveMultiplier>1</adultMoveMultiplier>
    <oldMoveMultiplier>0.4</oldMoveMultiplier>
  </movementData>
  <sizeData>
    <sizeMultiplier>1</sizeMultiplier>
    <scaleMale>3</scaleMale>
    <scaleFemale>2</scaleFemale>
    <youngSizeMultiplier>0.6</youngSizeMultiplier>
    <adultSizeMultiplier>1</adultSizeMultiplier>
    <oldSizeMultiplier>0.75</oldSizeMultiplier>
  </sizeData>
  <stateData>
    <flagState>Wandering</flagState>
    <previousFlagState>Wandering</previousFlagState>
    <deathReason>Eaten</deathReason>
    <beenEaten>False</beenEaten>
  </stateData>
  <targetData>
    <touchRadius>1</touchRadius>
    <sightRadius>50</sightRadius>
  </targetData>
  <lookingEntityData>
    <shortestToEdibleDistance>Infinity</shortestToEdibleDistance>
    <shortestToWaterDistance>Infinity</shortestToWaterDistance>
    <shortestToPredatorDistance>Infinity</shortestToPredatorDistance>
    <shortestToMateDistance>Infinity</shortestToMateDistance>
  </lookingEntityData>
  <ColliderTypeData>
    <colliderType>Fox</colliderType>
  </ColliderTypeData>
</Fox>
```

Figure 6.12: XML3

```xml
<Grass>
  <edibleData>
    <nutritionalValue>10</nutritionalValue>
    <canBeEaten>True</canBeEaten>
    <nutritionalValueMultiplier>1</nutritionalValueMultiplier>
    <foodType>Plant</foodType>
  </edibleData>
  <sizeData>
    <sizeMultiplier>1</sizeMultiplier>
    <scale>5</scale>
  </sizeData>
  <stateData>
    <flagState>None</flagState>
    <previousFlagState>None</previousFlagState>
    <deathReason>Eaten</deathReason>
    <beenEaten>False</beenEaten>
  </stateData>
  <ColliderTypeData>
    <colliderType>Grass</colliderType>
  </ColliderTypeData>
</Grass>
```

Figure 6.13: XML4

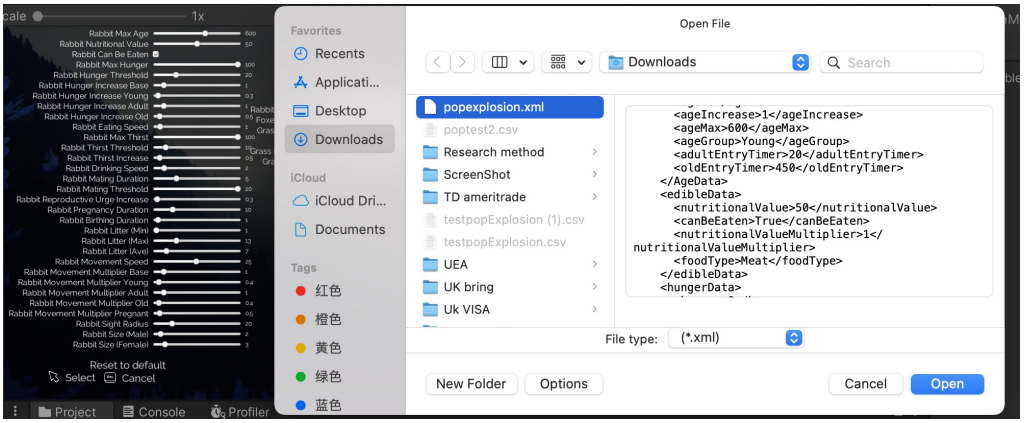

Figure 6.14: A easy way to reset Entity default value based on XML file that saved in previous Simulation using Loading XML file

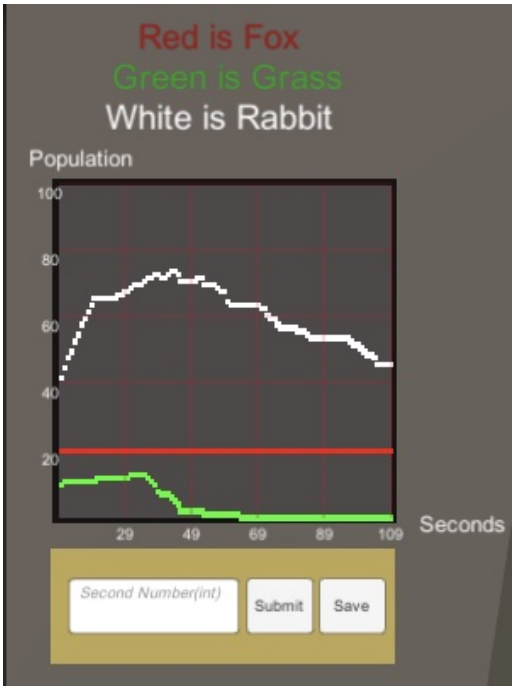

Figure 6.15: Graph

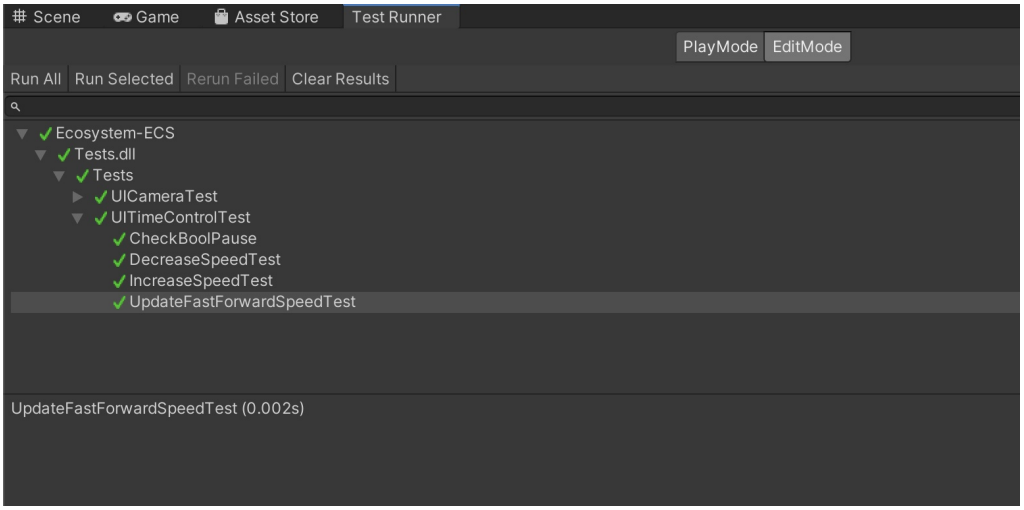

Figure 6.16: The Unity Test Runner tests code in Edit mode

```
using System.Collections;
using MonoBehaviourTools.UI;
using NUnit.Framework;
using UnityEngine;
using UnityEngine.TestTools;

namespace Tests
{
    "chenzhony"
    public class UITimeControlTest
    {
        private UITimeControl uITimeControl;
        private GameObject gameObject;

        [SetUp]
        "chenzhony"
        public void SetUp()
        {
            gameObject = new GameObject();
            uITimeControl = gameObject.AddComponent<UITimeControl>();
        }

        [UnityTest]
        "chenzhony"
        public IEnumerator CheckBoolPause()
        {
            yield return null;

            uITimeControl.Pause();
            Assert.True(uITimeControl.GetPause());

            uITimeControl.Play();
            Assert.False(uITimeControl.GetPause());
        }


        [Test]
        "chenzhony"
        public void IncreaseSpeedTest()
        {
            uITimeControl.fastForwardSpeed = 1f;
            uITimeControl.IncreaseSpeed();
            Assert.AreEqual( expected: 2f, actual: uITimeControl.fastForwardSpeed);

            uITimeControl.fastForwardSpeed = 0.2f;
            uITimeControl.IncreaseSpeed();
            Assert.AreEqual( expected: 0.4f, actual: uITimeControl.fastForwardSpeed);
        }

        [Test]
        "chenzhony"
        public void DecreaseSpeedTest()
        {
            uITimeControl.fastForwardSpeed = 1f;
            uITimeControl.DecreaseSpeed();
            Assert.AreEqual( expected: 0.8f, actual: uITimeControl.fastForwardSpeed);

            uITimeControl.fastForwardSpeed = 0.2f;
            uITimeControl.DecreaseSpeed();
            Assert.AreEqual( expected: 0.2f, actual: uITimeControl.fastForwardSpeed);

            uITimeControl.fastForwardSpeed = 0.4f;
            uITimeControl.DecreaseSpeed();
            Assert.AreEqual( expected: 0.2f, actual: uITimeControl.fastForwardSpeed);
        }

        [Test]
        "chenzhony"
        public void UpdateFastForwardSpeedTest()
        {
            uITimeControl.fastForwardSpeed = 1000f;
            uITimeControl.Play();
            uITimeControl.UpdateFastForwardSpeed();
            Assert.AreEqual( expected: 100f, actual: Time.timeScale);

            uITimeControl.fastForwardSpeed = -10f;
            uITimeControl.UpdateFastForwardSpeed();
            Assert.AreEqual( expected: 0f, actual: Time.timeScale);
        }
    }
}
```

Figure 6.17: UI Time Control Test

```
using MonoBehaviourTools.UI;
using NUnit.Framework;
using UnityEngine;

namespace Tests
{
    "chenzhony"
    public class UICameraTest
    {
        private UICameraFunction cameraFunction;
        private readonly Vector2 mapSize = new Vector2( x: 100,  y: 120);

        [SetUp]
        "chenzhony"
        public void Setup()
        {
            cameraFunction = new UICameraFunction(mapSize);
        }

        [Test]
        "chenzhony"
        public void CheckCameraBorder()
        {
            var yMin :float  = cameraFunction.GetYMin();

            Vector3 pos1 = new Vector3( x: float.PositiveInfinity, yMin,  z: float.PositiveInfinity);
            Vector3 pos2 = new Vector3( x: float.NegativeInfinity, yMin,  z: float.NegativeInfinity);

            Vector3 pos3  = cameraFunction.CheckCameraBorder(pos1, cameraLastPositionY: yMin+5f);
            Vector3 pos4  = cameraFunction.CheckCameraBorder(pos2, cameraLastPositionY: yMin+5f);

            Assert.AreEqual( expected: new Vector2(pos4.x, y: pos3.x),  actual: new Vector2 ( x: -mapSize.x * 5,  y: mapSize.x * 5));
            Assert.AreEqual( expected: new Vector2( x: pos4.z, y: pos3.z),  actual: new Vector2 ( x: -mapSize.y * 5,  y: mapSize.y * 5));

            Vector3 pos5  = cameraFunction.CheckCameraBorder( cameraPosition: new Vector3( x: mapSize.x * 5, y: yMin+4f, z: mapSize.y * 5),yMin);
            float camSpeed = cameraFunction.GETCameraSpeed();
            Assert.AreEqual( expected: new Vector2(pos5.x, y: pos5.z),  actual: new Vector2 ( x: mapSize.x * 5, y: mapSize.y * 5));

            Vector3 pos6  = cameraFunction.CheckCameraBorder( cameraPosition: new Vector3( x: mapSize.x * 5, y: yMin+5f, z: mapSize.y * 5),yMin);
            float camSpeed2 = cameraFunction.GETCameraSpeed();
            Assert.AreEqual( expected: new Vector2(pos5.x, y: pos5.z),  actual: new Vector2 ( x: mapSize.x * 5, y: mapSize.y * 5));
            Assert.Less(pos6.x,mapSize.x*5);
            Assert.Less(pos6.z,mapSize.y*5);
            Assert.Less(camSpeed2, camSpeed);
        }
    }
}
```

Figure 6.18: Movable Camera Test Code

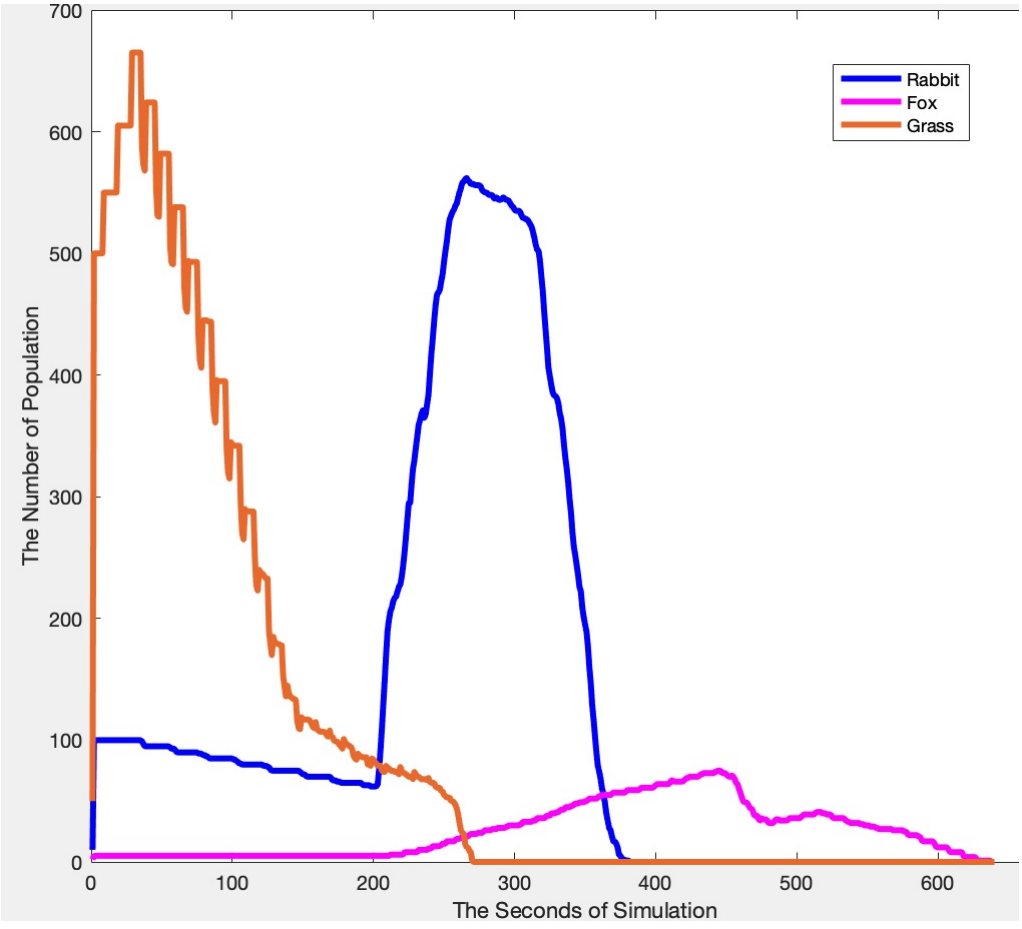

Figure 6.19: Graph of population under simulated 'normal' conditions

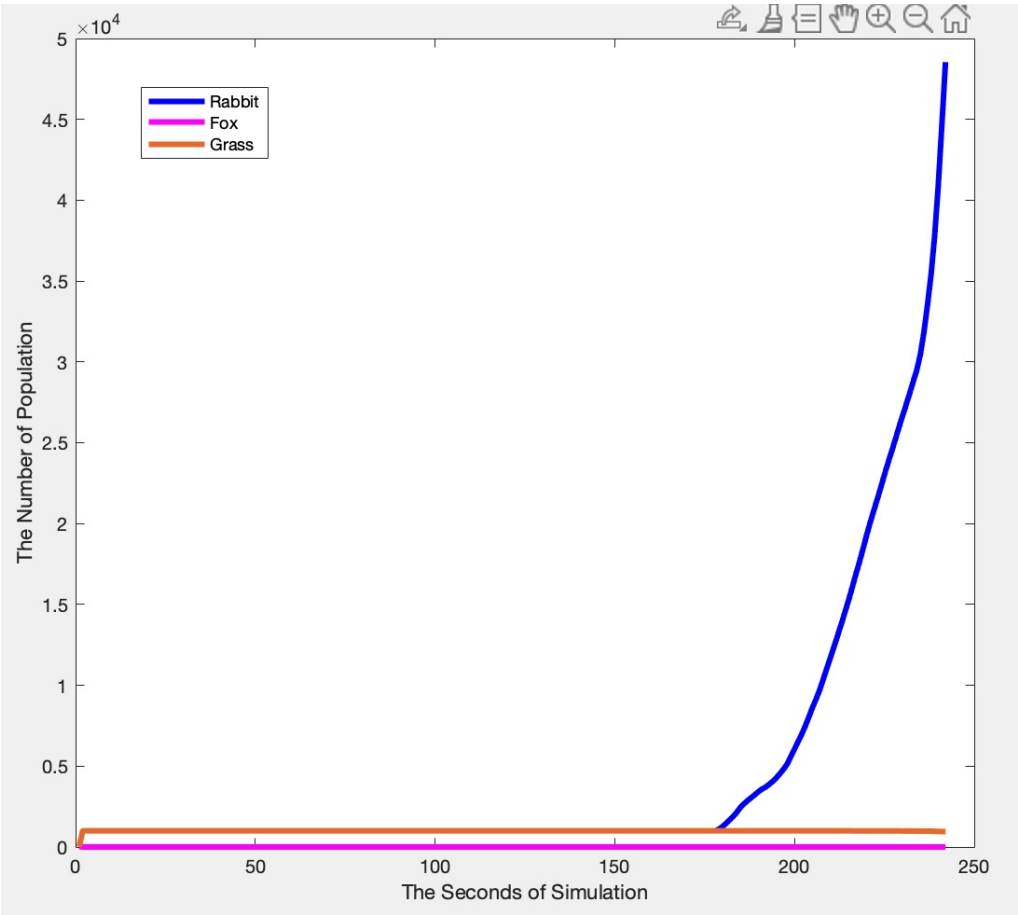

Figure 6.20: Graph of population showing simulated infinite food and no predators