University of Glasgow | School of Computing Science

Blobs on a Plane

George Popa

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — March 21, 2016

Abstract

The project involves building an evolutionary simulator, dealing with observations of mimicked natural evolutions. It is intended as a teaching aid, with the user being able to intervene and change parameters in the simulation. Creatures called ”blobs” evolve and adapt to the environment, allowing the user to study group behaviours and population dynamics.

A on-line deployment is available: `https://s3-eu-west-1.amazonaws.com/ui-v1/webtest.html`

Contents

Chapter 1

Introduction

1.1 Motivation

While multiple systems implementing genetic algorithms are available, outlined in Section 2.1, they tend to be obsolete, no longer supported, or generally intended as frameworks providing only the algorithms to be used in optimization. Moreover, either due to underlying complexity or intended scope, these were unsuitable as teaching tools. In general, the focus also tends towards each individual in particular, rather than the population as a whole.

1.2 Overview

The project is intended as a light-weight educational utility, in order to provide insight into the dynamics of an evolving population under environmental conditions. The purpose of the simulation is to observe how average values of parameters among a population change with respect to events in the environment. From these changes, the overall direction of evolution can be inferred. By also allowing the user to directly interact with the population and its food sources, a higher level of responsiveness is achieved: the simulation adapts to deal with user input.

The creatures being simulated, called "blobs", are modelled to behave similar to bacterial lifeforms. For simplicity, the model is highly abstracted, disregarding the physical layer, such as the shape of an individual, particular cellular properties, or metabolism. Instead, focus is put on the behavioural side, and the genetics.

1.3 Dissertation Outline

The rest of the dissertation presents in detail the process of development of the evolutionary system, comically named "Blobs on a Plane". The outline is as follows:

Chapter 2 deals with some of the previous research carried out in the field, as well as other pre-existing applications.

Chapter 3 details the functional and non-functional requirements, as well as the requirements gathering process.

Chapter 4 discusses various concerns about the design of the system and the user interface, any assumptions, and the design decisions made. Presented are also the initial design ideas and the final design.

Chapter 5 describes the implementation details, choice of framework, some of the evolutionary algorithms used, as well as the project deployment.

Chapter 6 deals primarily with evaluation, from both users and experts, as well as internal testing.

Chapter 7 provides a summary of the project, project management matters, and future work.

Chapter 2

Background

This chapter aims to present the reader with useful explanations of the field-specific terms used in the dissertation as well as provide insight into the significant research previously carried out. This chapter aims to present a general idea of the existing research, both in the field of computing, and in the field of biology . Included are example of pre-existing applications, either implementing similar algorithms, or possessing a comparable purpose.

2.1 Previous Work

Evolutionary computing is the field of artificial intelligence devoted to solving problems by implementing Darwinian principles of evolution. It includes evolutionary algorithms, genetic algorithms, and genetic programming. Starting from the premise of a common biological ancestor, Darwin proposed evolution from the simple to the complex, given enough time. With mutations appearing in a creature's genetic code, he stipulated that beneficial mutations providing an advantage would be passed on to future generations [9]. Accumulating multiple such mutations would lead to a completely new organism, different from the original.

2.1.1 Evolutionary Algorithms

Since initially proposed by Alan Turing in 1950 under the form of "learning machines" [27] akin to biological systems, such simulations became the study of subsequent decades. Selection, key to Darwinian principles, was first simulated by Alex Fraser, in 1957 [10].

When faced with complex optimization problems, performing a linear search, especially through a large solution space usually associated with such problems, may prove ineffective. Evolutionary algorithms became regarded as a method of solving complex optimisation problems in the 1970s [16]. In general, several biological methods, such as sexual or asexual reproduction or genetic mutation, are used in order to evolve the population of a particular system with respect to a certain predetermined fitness function. As such, more "fit" individuals have a higher chance of being selected to pass on their genome to the next generation, as seen in Figure 2.1, thus increasing the fitness factor of the overall population after each iteration. Over a longer period of time spanning from a few thousand, to a million generations, the population will evolve towards a solution. This, however, suffers from diminishing returns [11], as the population tends to meet an impasse in locally optimal points, rather than reach the global maximum.

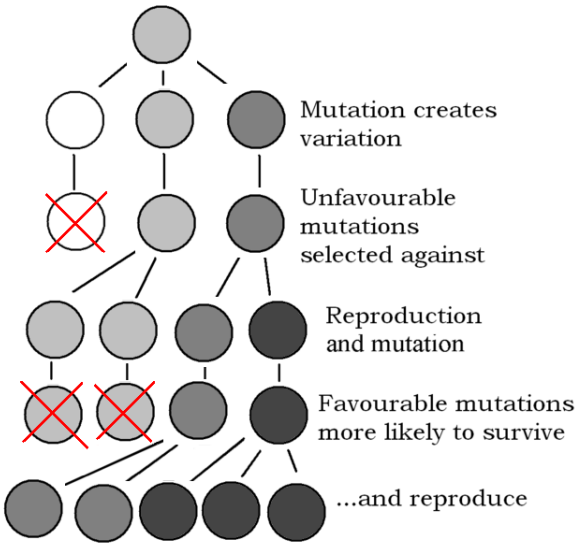

Figure 2.1: A diagram showing a simplified process of natural selection.
Copyright Wikimedia Foundation, under Creative Commons Attribution-Share Alike [3]

2.1.2 Biological Simulation

Building upon the basis of genetic algorithms, biological simulations become possible.

A more prominent paper in the field of artificial evolution is "Evolving Virtual Creatures" by Karl Sims [26]. The paper details an evolutionary system which focuses evolving a population of creatures consisting of three dimensional blocks with respect to various movement-specific tasks (such as walking, running, jumping, climbing, and swimming). An example of such creature is represented in Figure 2.2. Noteworthy is the fact that both the creature's morphology and it's "brain" are evolved. While the initial randomly created generation was incapable

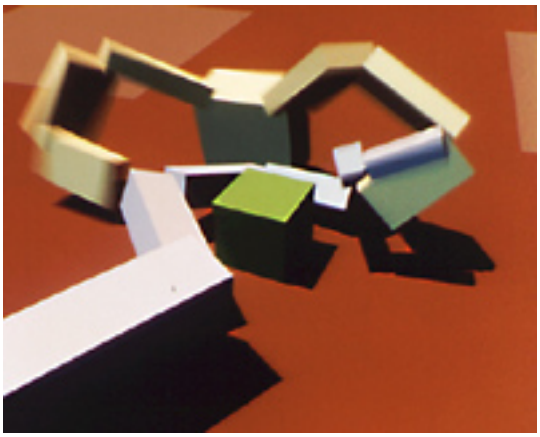

Figure 2.2: A screenshot from the simulator build by Karl Sims "Evolved Virtual Creatures"

of performing any task, subsequent generations showed significant improvement. Even after a relatively low number of generations, some creatures were becoming adapted to their environment. In the swimming example, some creatures were successful in developing fin-like appendages, used for locomotion, while other replicated the waving motion of water-snakes. The scope of the experiment, however is quite limited, as creatures are observed individually, with a rather basic fitness function, and not in competition with each other.

A more recent paper, Dan Lessin's et al. "Open-ended behavioral complexity for evolved virtual creatures" [18],

deals with the open-ended nature of evolution, previously unexplored in Karl Sims's paper. The paper proposes the evolution of a creature's brains as well as body, thus allowing it to better adapt to its environment. While the evolution of the body is fairly similar to the method described by Karl Sims, the control part of a creature is evolved differently. This involves encapsulating a creature's previously learned skills. An example of such a creature, together with its subroutines can seen in Figure 2.3. The benefits presented are persistent memory,

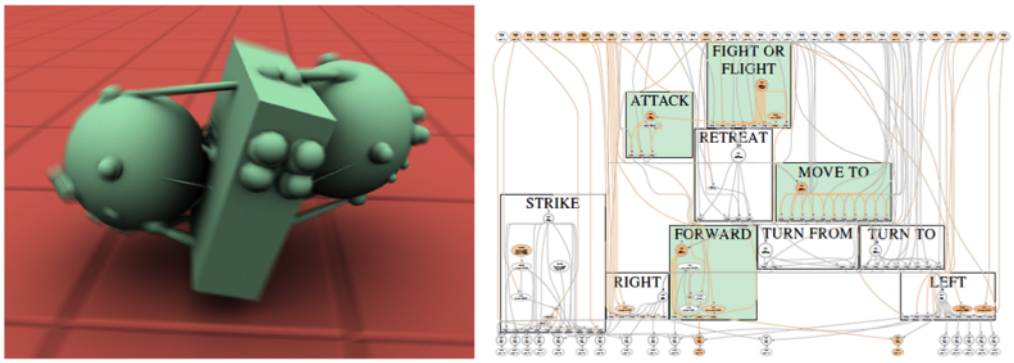

Figure 2.3: An example of a creature's physiognomy and control circuit

allowing a creature to store a certain behaviour once it has been learned, and overall simplicity, allowing the whole procedure to be called as a single node, instead of actively recreating the movement. By allowing newer nodes to be modified more frequently than older ones, the algorithm simulates the forming of habits. By changing the fitness function, creatures can adapt to different environments, or learn more complex behaviours, without "forgetting" their previous training. This, in turn, provides a more accurate representation of nature.

2.1.3 Virtual Worlds

While research in the field of Genetic Algorithms is extensive, it deals mostly either with problems of optimization, or with the best few individuals of a population. This is not the case when trying to simulate the entire population, as being the very best only gives a marginally higher chance of passing genes to the next generation. Simply being "good enough" to eat sufficient food in order to be able to live and reproduce is adequate. This is especially important when dealing with a heterogeneous environment, where food density might vary. In this scenario, simply being in an area with more food could be considered more beneficial than becoming the best individual. A common occurrence in such simulations is the gathering of individuals around food clusters, with those to do so dying off.

Darwinbots

A similar application in appearance is Darwinbots [6]. Darwinbots presents itself as an artificial life simulator, where creatures, called "bots" can compete for food in a simulated two dimensional environment. The most successful "bots" will live on to pass their genes to subsequent generations. Darwinbots, however, specialises in simulating Von Neumann machines.

Darwinbots specialises in providing a platform for interactivity between multiple species of organisms with wildly varying behavioural patterns. Should it prove beneficial to do so, creatures may combine different purposes to form more complex, specialised, multi-cellular organisms, with constituent parts focusing independently on locomotion, sensory input, digestion, and combat.

The creatures implement a piece of code represented as a syntax tree as their inherent algorithm. Using tree crossover operations described in "A Field Guide to Genetic Programming" [22], these algorithms, which essentially

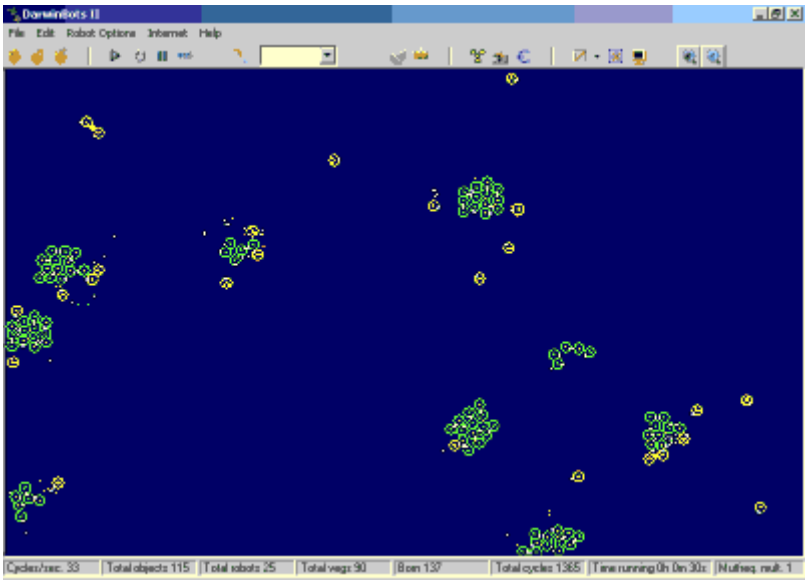

Figure 2.4: A screenshot from a Darwinbots simulation

form the control circuit of a particular individual, are evolved. Due to the nature of mutation in tree cross-over individuals of the same species can evolve in different directions, should both be more optimal than the current incarnation, due to divergent evolution.

Goopies

A more simple approach is taken by Paul T. Oliver in his project called "Goopies" [21]. The creatures live in a bounded, circular world, together with food pellets and obstacles. An example of such a world can be seen in Figure 2.5. A creature's behaviour is determined by its intrinsic neural network. Creature physiognomy exists only for the purpose of conveying information about certain parameters of its control circuit. Unlike Darwinbots, at the end of one generation, the best individuals with respect to a fitness function are selected. Crossover is performed on their genes and a new population is created.

In this simulate world, creatures do not actively interact with each other like in Darwinbots. The impact made by an individual on the others is the food he consumes and his corpse. This has generated a peculiar behaviour where "Goopies" would linger close to others close to their death, similar to carrion-eater behaviour in animals.

Conway's Game of Life

Introduced by John Conway in 1970 [12], the eponymous game is a two-dimensional implementation of a cellular automation. According to a predefined rule, the subsequent states are created from an initial state. Building on von Neumann ideas of a machine capable of self-replication, Conway attempted to simplify the concepts. Due to the fact that it implemented a universal Turing Machine [2], this sparked interest for the game since its inception.

It is particularly simple to observe patterns among the population. Notable among the categories of patterns are "still lifes", "oscillators", and "spaceships". The first denote a series of arrangements of cells that, if left alone, do not change in any way [13]. Oscillators are a set of repeating patterns, characterised by a period. "Spaceships" are patterns that can translate themselves across the grid. Two gliders, seen in Figure 2.6, can be shot at a particular angle at a static two-by-two block in order to move it. This can act as memory for a more complex system implementing a finite-state machine.

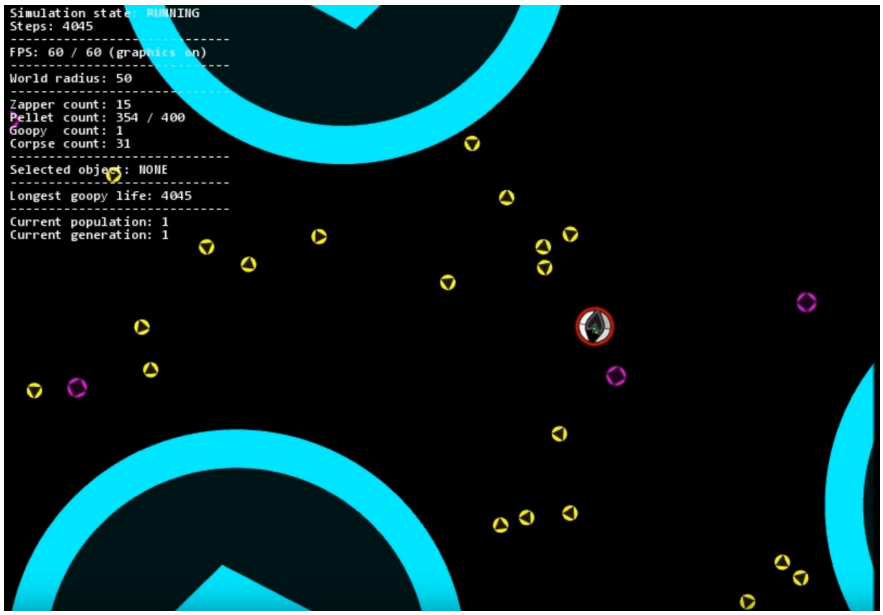

Figure 2.5: A “goopie” searching for food

Figure 2.6: A cellular formation that has the property to generate “gliders”

2.2 Summary

Presented above are different examples of evolutionary computing, ranging from genetic algorithms used for optimization problems, to entire simulated virtual worlds. The “Blobs on a Plane” project is based on such algorithms, and in turn, attempts to provide a simplified simulation where population dynamics can be observed in relation to environmental parameters. The next chapter will present the requirements for the project.

Chapter 3

Requirements

3.1 Requirements Gathering Process

This section describes the requirements gathering process, as well as enumerates and details the requirements themselves. Since the project is self-defined, there were no requirements provided. Based on the initial project specification, a list of requirements was drafted. During the background research this list changed both shape and scope. It reached its final form after a video conference with Simon Hickinbotham [15], a researcher in Evolving Systems. Simon's expertise proved vital in aligning the simulated behaviour to be more in line with real-life bacteria.

3.2 Functional / Non-Functional

The final functional requirements are presented in MoSCoW format [8]. This was selected in order to complement the Agile style of development and to provide clear priorities. As such, critical deliverables are presented as "Must", important features are labelled as "Should", desirable requirements as "Could", and least-critical items as "Would".

3.2.1 Functional Requirements

Must Have

- **Visualisation**
  The system should provide the user with a visualisation of the simulation.
- **Parameter sliders**
  The user should be able to control parameters of the environment.
- **Statistics**
  The user should be provided with details about the population.
- **"Blob" behaviour**
  The system should be capable to simulate food finding behaviour.
- **Evolve blobs**
  The system should evolve the blobs in some way.

Should Have

- **Direct user interaction**
  The user should be able to actively interact with the simulation in real time. (Add food sources, add extra "blobs")
- **Reproduction**
  The system should be able to simulate some form of reproduction; either sexual or asexual.

Could Have

- **Camera controls**
  The user should be able to move the camera around, as well as zoom in or out.
- **Evolutionary parameters**
  The user should be able to interact with the parameters of the evolution itself.

Would Have

- **Tree crossover**
  The evolution should include tree crossover for blob behaviour.
- **"Blob" characteristics**
  The user should be able to tweak the characteristics of individual "blobs".

3.2.2 Non-Functional Requirements

Since the system's intended purpose is that of a teaching tool, the non-functional requirements are concerned mainly with usability and interactivity. While performance is not an immediate concern, the systems should be able to deal with a rapidly increasing population.

- **Intuitiveness**
  The user interface should be easy to use and intuitive.
- **Interactivity**
  The user should be able to interact with the simulation.
- **Performance**
  The visualisation should be able to handle a sizeable population.
- **Cross-Platform**
  The system should be able to run on a multitude of platforms.

3.3 Summary

This chapter presented the requirements for the project. The next chapter will explain how the system was designed in order to match these requirements.

Chapter 4

Design

4.1 Overview

The system simulates a population of individuals, termed "blobs", whose sole purpose is continued existence. This is achieved thorugh continuous feeding, followed eventually by reproduction. The latter is done asexually; when each individual reaches a sufficient energy threshold encoded in its internal state. Energy can be gained by eating food pellets scattered randomly around the world. The blobs eat food by colliding with it, and find it by performing a random walk, in this case, a simplified version of a Lévy walk[20].

The system is designed with interactivity in mind, allowing the user to alter various parameters of the environment, while observing how the population evolves in order to adapt to the change. A user could interact more directly with the simulation by adding blobs or placing food. A sudden influx in either will generate a period of instability in the population which eventually reverts back to a stable state. Information about the population is presented via a graph, showing total number and average characteristics of the individuals. More specific information about a blob in particular is displayed via a pop-up message, should a creature be clicked on.

4.1.1 Definitions

In order to provide a better understanding, some terms are defined below:

- Blob — A single creature in the simulation;
- Food — A nutritional pellet, providing Energy to whomever eats it;
- Population — The set containing all blobs: at a certain time;
- Energy — A blob's measure of how healthy it is. This triggers death at low values;
- Patience — The time measured in number of ticks until a blob leaves a certain area.
- Reproduction — The act of division of a blob when an energy threshold is reached;
- Genome — A simulated DNA string in which the parameters of a blob are encoded.
- Mutation — The act of a single bit in the genome becoming inverted at random.

4.1.2 Goals and Considerations

As the main purpose of the system is that of a teaching aid, it was designed with the primary goal of interactivity in mind. This resulted in a need for a visualisation as an effective way of conveying information. A visual component would present the state of the system, and at the same time, it would allow the user to directly interact with the simulation in real-time. A further goal was portability; due to the variety of platforms available, allowing the simulation to run on as many as possible became of paramount importance. As such, the final version of the system is built using the Unity Game Engine[5].

4.2 Design Process

Before any implementation work was carried out, the system specifications were laid out. Based on the pre-established requirements (Chapter 3), a plan of the system was drafted. An early draft can be seen in Figure 4.1. This plan was further refined into the existing architecture, currently containing two logical components: the visualisation, and the simulation logic. The characteristics of an individual blob were also discussed, together with a set of basic behaviours, as seen in Figure 4.2.

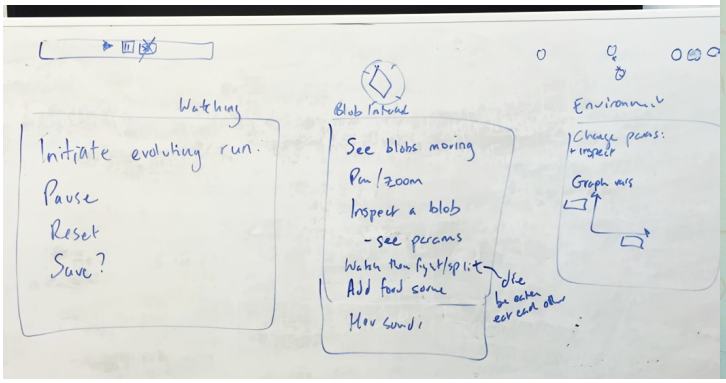

Figure 4.1: The base functions of the system, split into logical components

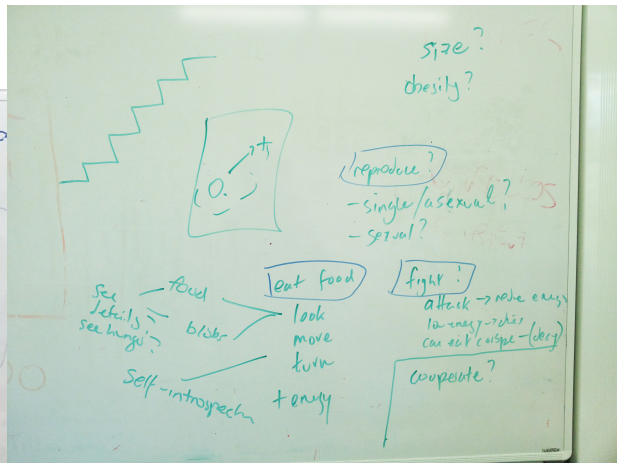

Figure 4.2: Initial blob characteristics and behaviour

4.2.1 Simulation Type

The system is an agent-based model (ABM), as it focuses on simulating individual blobs. ABMs are often used in biology as the mapping from the simulated entities to their real-world equivalents is straightforward. The paper "Agent-based models in translational systems biology"[7] highlights the advantages of ABMs for biological systems. Noteworthy are the characteristic representation of space and strongly-enforced rules, as the origins of ABMs are cellular automatons, and the stochastic nature of the individuals inside the simulation. Despite the simplicity of the simulation, the emergent property is still verified, as complex population dynamics can be observed from uncomplicated individuals.

Similar to most agent-based models, the system is discrete event simulation[25]. The simulation advances only on update ticks which can happen with varying frequency, between 1 and 60 ticks per second. The speed of the ticks is controlled by the user. From a spatial perspective, however, the system is continuous, as a blob can occupy any position on the plane, unrestricted by any imposed grid; this was done in order to provide a greater freedom of movement.

4.2.2 Framework selection

During the initial stages of the projects, several development platforms and frameworks were explored. Among the initial choices were Windows Forms Application, and a more general DirectX based, .NET implementation. These, however, proved inadequate, either due to performance issues in the former, or portability issues which both suffered from. While this line of implementation was stopped, elements of the interface design were kept.

A later implementation was attempted in Mono[4], an open-source .NET framework built with platform portability in mind. This solved the concern of cross-platform, but did not provide a viable solution for the visualisation, as the existing portable GUI tool-kits for Mono proved difficult to work with. As such, Mono was dropped as an implementation choice.

The final incarnation of the project, is implemented in the Unity Game Engine. Due to its intended nature as a game engine, the performance issues previously encountered with other tool-kits were not present. Unlike Mono, Unity also offered a built-in way of designing the user interface. In order to target a larger audience, the project could be compiled to run on any of the three major platforms (Windows, MacOS, Linux), as well as mobile platforms, and web browsers, via the Unity Web Player. Together with the Asset Store allowing the import of plug-ins, Unity became the framework of choice. A screenshot of the final project open in the Unity Editor can be seen in Figure 4.3.

A comparison of the previously presented frameworks can be found in Table 4.1.

	Performance	Portability	Ease-of-use	Language	Community support
Windows Forms	×	×	✓	C++, C#	×
.NET	✓	×	✓	C++, C#	✓
Mono	✓	✓	×	C++, C#	✓
Unity	✓	✓	✓	C#	✓

Table 4.1: Comparison between explored frameworks

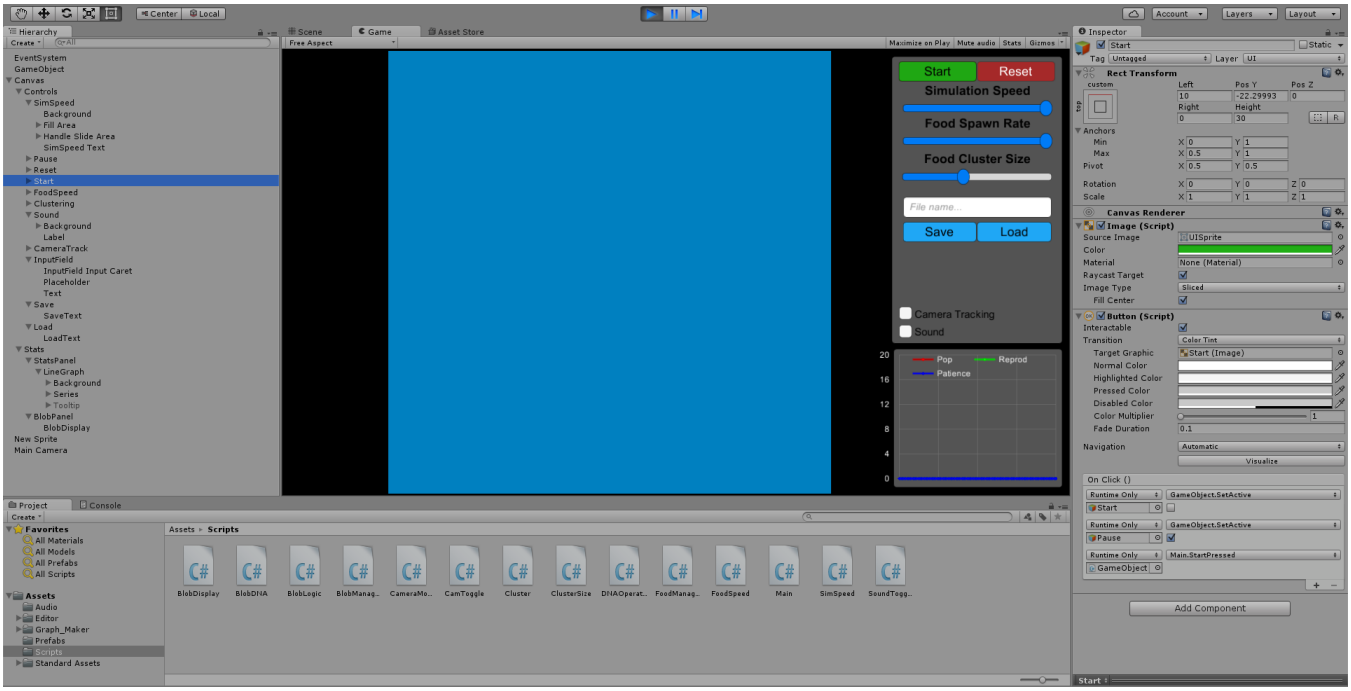

Figure 4.3: The Unity Editor running the system

4.3 Architecture

The underlying architectural pattern of the system is the Model-View-Controller (MVC) design pattern. Initially introduced in the late 1970s, MVC was described by its creator in "The original MVC reports"[24] as a solution for users handling complex data sets. A diagram of how MVC is applied to the system is presented in Figure 4.4. The arrows represent channels of interaction: whether direct manipulation by an user through the user interface, data paths in the system, or visual feedback on the screen. The user interacts with the controller pictured in yellow, which modifies the data in the model, depicted in blue for blob data and green for food-related data. This in turn is reflected in the view, coloured in purple, presented back to the user. MVC is used in order to provide separation between the user interface, the visualization, and the simulation logic. Such separation allows changes to either of the three components with relative ease and minimises the impact on the others. An informal diagram detailing the interaction between the classes directly involved in the simulation can be seen in Figure 4.5.

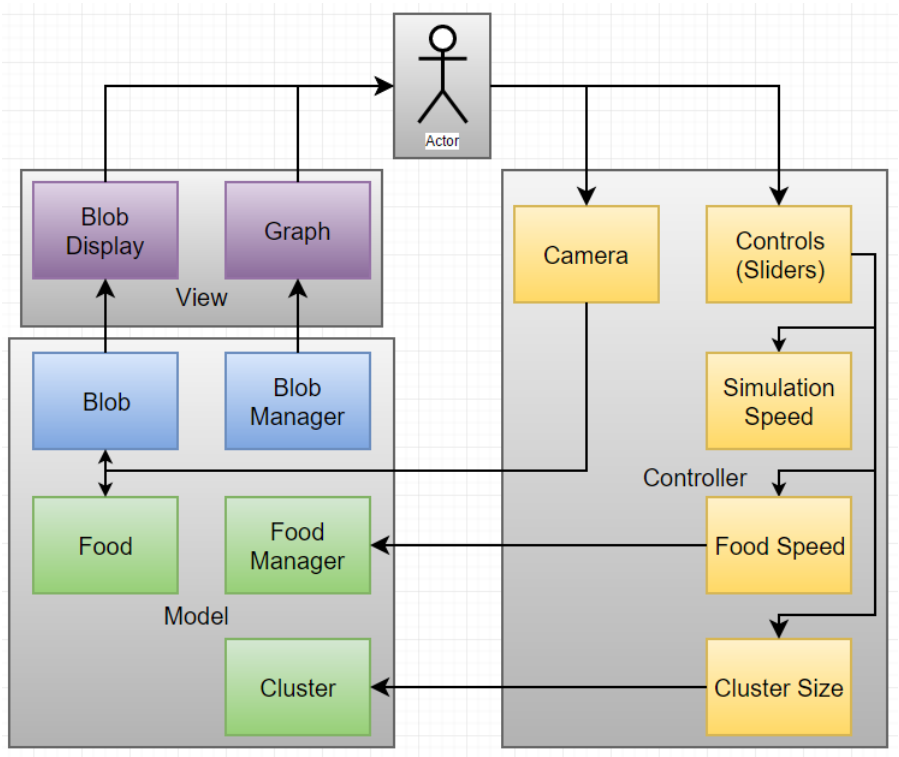

Figure 4.4: The design of the system as an MVC

4.4 Interface

The design of the interface was an iterative process, starting from whiteboard and paper prototypes, continuing with partial, non-functional implementations, and arriving at a usable interface, built in Unity. The user evaluation proved key, as some of the feedback received (Section 6.3) was used in the redesigns. This incremental process complemented the Agile development method employed. A detailed view on how Agile was used in order to build the system can be found in Section 7.2.

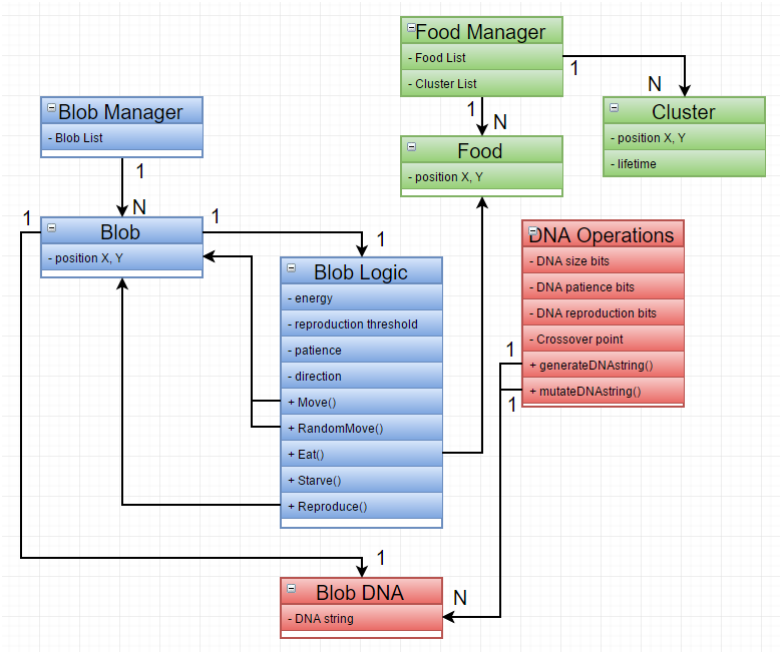

Figure 4.5: Informal class diagram

4.4.1 Interface iterations

While the initial interface sketches were crude, they quickly converged towards a more stable, intermediate design. Some of the initial prototypes can be seen in Figure 4.6, as well as Figure 4.7.

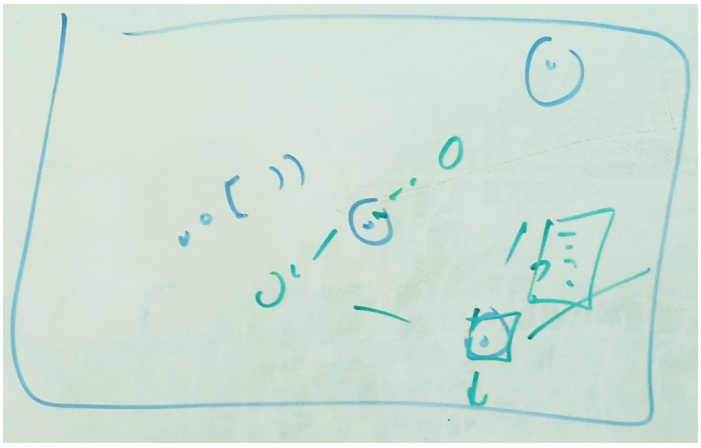

Figure 4.6: The base functions of the system, split into logical components

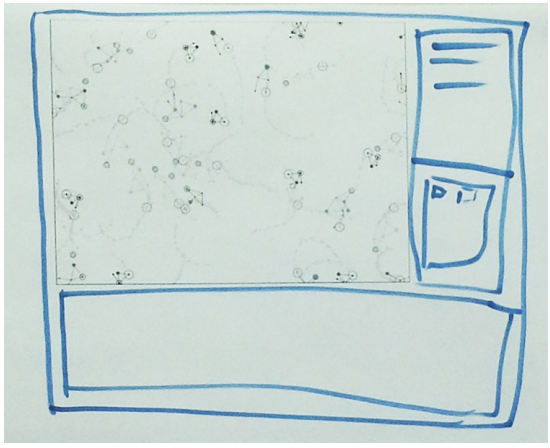

Figure 4.7: Initial blob characteristics and behaviour

Following the initial designs, a computer-assisted prototype was created. Even though this was non-functional, it served as a baseline for any subsequent iterations. A mock implementation can be seen in Figure 4.8. This includes the environmental controls via the sliders, as well displays containing information about a selected blob and the population in general.

After the transition to Unity, the interface was remade using the build-in Unity UI tools. Information about the population is also displayed via a rudimentary histogram in the top-left corner of the screen. An intermediary design can be seen in Figure 4.9.

Together with the line graph and the blob selection tool-tip, implantation of the camera changes led to the finalised

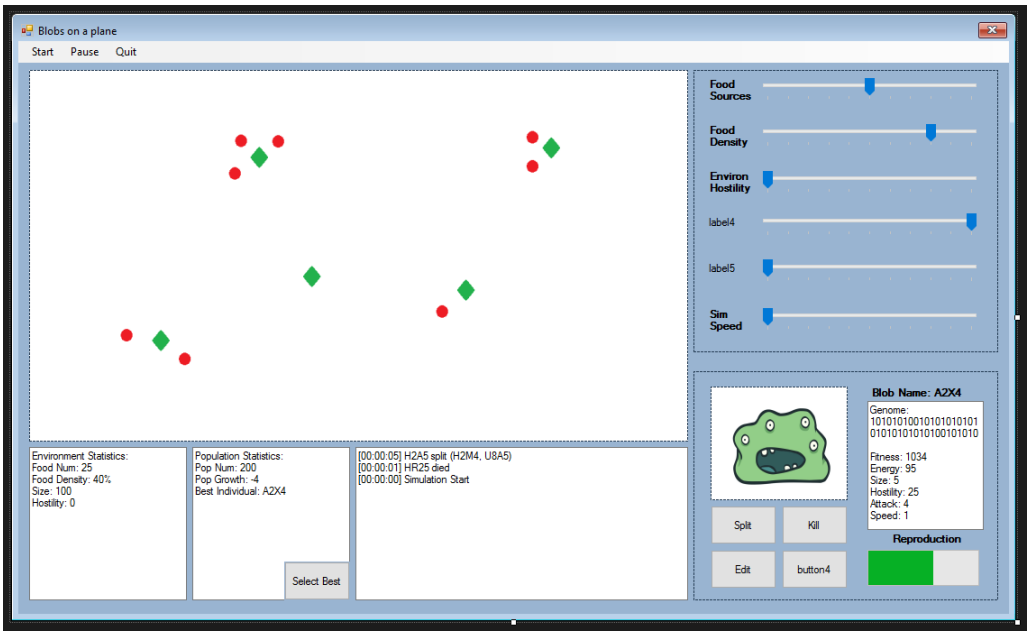

Figure 4.8: User Interface and Visualisation build in Windows Forms Application

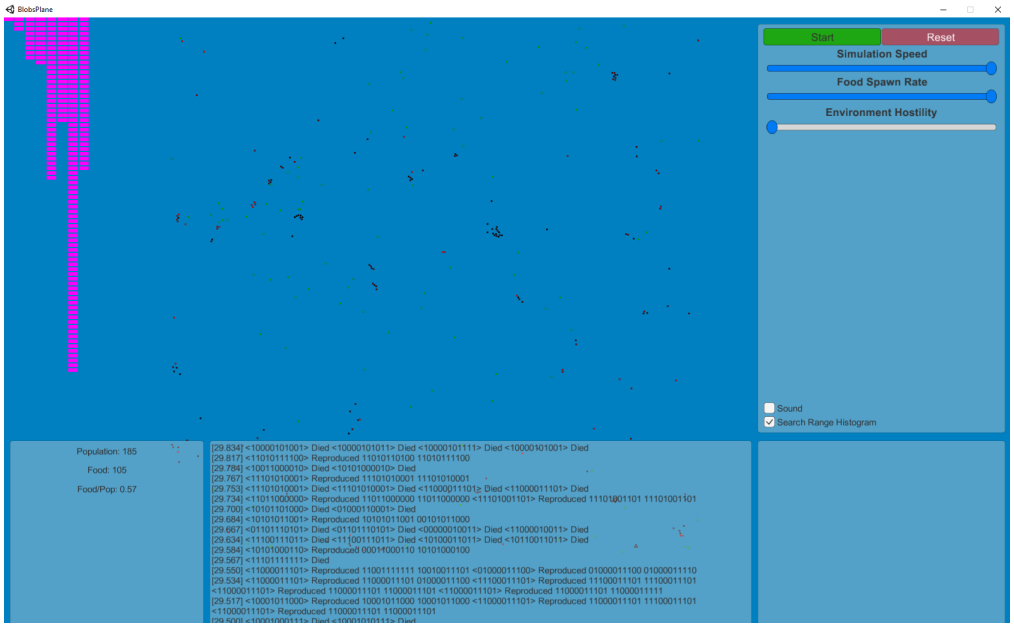

Figure 4.9: One of the intermediary designs

design, as seen in Figure 4.10

4.4.2 Statistical Graph

In the bottom-right of the application, a graph displaying information about the current population can be seen. An example is presented in Figure 4.11. The data series available are: number of individuals, the average reproduction threshold amongst the population, and the average time until a blob leaves an area in search of food. Albeit few, these three parameters accurately represent trends within the population following significant environmental events, such as those described in Section 6.2. The graph is created with the help of a library available through the Unity Asset Store as a purchasable Asset[1].

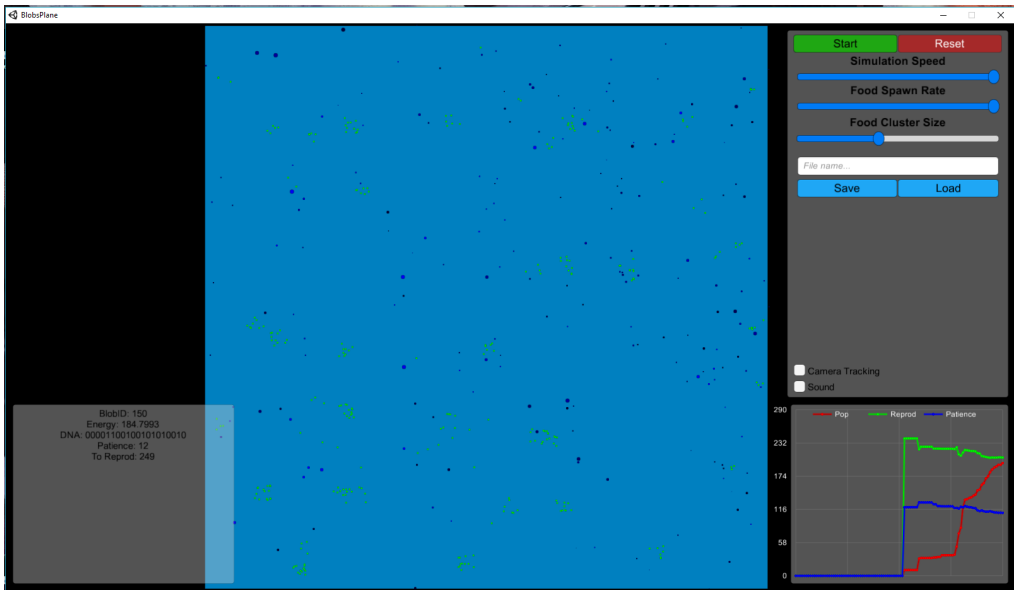

Figure 4.10: Final design of the application

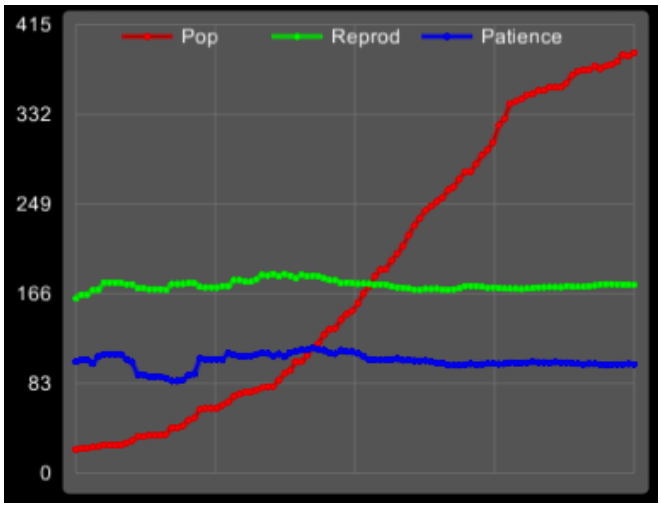

Figure 4.11: A screenshot of the line graph

4.5 Evolution Logic

In the presence of food, blobs will just float in the general area. Once food levels decrease, the blobs simulate a type of random walk, named Lévy walk. This is also used by bacteria, and is thought to be an optimal strategy for searching sparsely populated random environments[28], such as food distributed on a plane. To note is that the number of food pellets available cannot exceed a predefined maximum, in order to keep the population within manageable levels.

As the blobs gain energy by eating, they become visually larger, until they reach their reproduction threshold. This causes a blob to begin asexual reproduction, effectively splitting into two children. Energy is lost over time; should at any point a blob's energy reach zero, it would immediately die. A blob can be formally described as a finite-state machine, as seen in Figure 4.12.

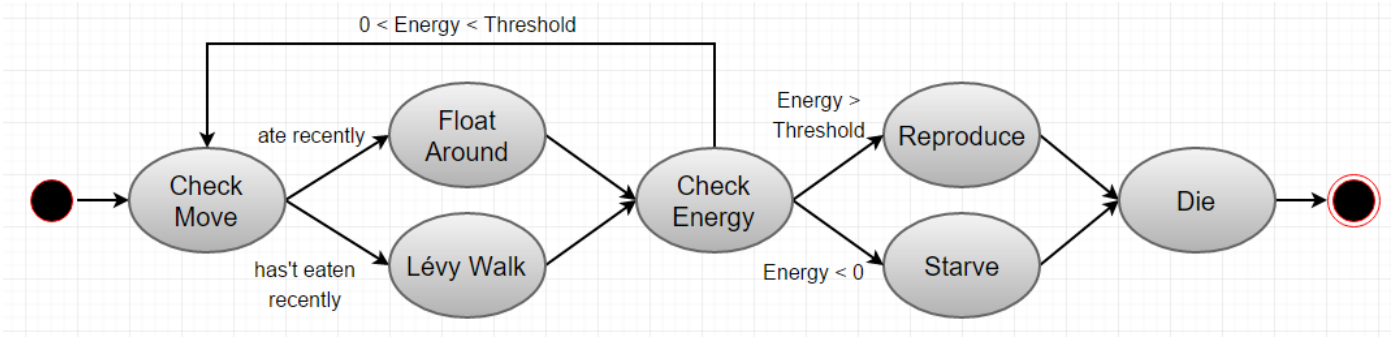

Figure 4.12: A finite-state machine detailing a blob's behaviour

4.5.1 Reproduction and DNA

Reproduction was chosen to be asexual in order to simplify a blob's behaviour to its basics. Sexual reproduction would involve either the need to search for a partner, or the inherent randomness of blobs colliding and exchanging genes. This would add an unnecessary layer of complexity to the simulation. As such, a blob simply divides into two children in a manner similar to cytokinesis[23]. Each child carries the DNA of its parent, with a slight variation dependant on whether a mutation occurred. Encoded in a blob's DNA are its two main characteristics: the reproduction threshold and the time needed until it starts searching for a new food source.

Chapter 5

Implementation

This chapter will explain how the design was translated into a concrete system implementation. The chapter is split with respect to it's main logical components: the simulation logic, Section 5.1; the logic behind placing and handling food, Section 5.2;

5.1 Main Logic

Following the Model-View-Controller architectural pattern, the system implements multiple independent update loops in order to separate the simulation logic from the frame rate of the visualisation. Each of the individual update functions are called by the Unity engine at a frequency of 60 Hz. In order to provide the user with the option to slow down the simulation, a counter is implemented and checked in all updating classes. A global delay is set and acquired by each class individually to be used to synchronise their update; this is done in order to ensure consistency. The aforementioned counter, t, is measured in milliseconds and it records the time since the last frame was completed. This is given by "deltaTime" and converted to milliseconds. An example of this synchronisation can be found below. The update will only be carried out if sufficient time has passed since the previous update.

```
void Update()
{
    [...]
    t += Time.deltaTime * 1000;
    if (t > delay)
    {
        [...]
        t = 0f;
    }
}
```

Every Unity update tick, that is 60 times per second, the statistical graph of population statistics, explained in Section 4.4.2, is updated. This is separate from the simulation update time in order to provide a better resolution should the simulation be slowed down by the user. As such, the graph can present a constant stream of data between farther apart simulation updates, allowing for a better observation of small changes in the data series. From the perspective of the MVC, the graph can be considered part of the view, therefore it is updated separately from the model.

Data in the graph is extracted from the BlobManager class that contains a list of all the blobs in the simulation, each with their own subcomponents: BlobLogic and BlobDNA. The information presented is the blob population

number, obtained simply by getting the size of the list; the reproduction threshold and patience are averaged across the entire population, giving the arithmetic mean. As this type of average is influenced by outliers, it was selected to render variations in the population's characteristics more obvious. The graph has three internal data series named: population_data, population_reprod, and population_patience. Each of the values acquired above are added to the end of its respective list. The first element is deleted in order to create a real-time display. An example for the reproduction threshold can be found below.

```
public void UpdateLineGraph()
{
    [...]
    float avgreprod = 0f;
    foreach (GameObject b in BlobManager.blobs)
    {
        if (b != null)
        {
            avgreprod += b.GetComponent<BlobLogic>().getReprod();
        }
    }
    if (BlobManager.blobs.Count != 0)
    {
        avgreprod /= BlobManager.blobs.Count;
    }
    else
    {
        avgreprod = 0;
    }
    population_reprod.pointValues.Add(new Vector2(0, avgreprod));
    population_reprod.pointValues.Remove(population_reprod.pointValues[0]);
    [...]
}
```

5.1.1 Simulator State

The simulation has three states, tracked by two internal variables. The default state is the initial one: an empty plane. Once the user presses the Start button, the simulation passes into the running state. The variables are updated accordingly; both "gameState", which differentiates between run and pause, and "started", which tracks the first click of the Start button. The pause button simply sets the variable tracking the running state back to the Paused state. Should, at any point, the Reset button be pressed, both blobs and food pellets are deleted and the simulation is returned to the initial state. Below can be found the detail of the functions implemented by the control buttons.

```
public void StartPressed()
{
    gameState = GameState.Started;
}

public void PausePressed()
{
    gameState = GameState.Paused;
}

public void ResetPressed()
{
    gameState = GameState.Paused;
    started = false;
```

```
    ResetLineGraph();
    BlobManager.blobs.Clear();
    FoodManager.foods.Clear();

    timestamp = 0; BlobLogic.ID = 0;

    var toClear = GameObject.FindGameObjectsWithTag("Blob");
    for (int i = 0; i < toClear.Length; i++)
        Destroy(toClear[i]);

    toClear = GameObject.FindGameObjectsWithTag("Food");
    for (int i = 0; i < toClear.Length; i++)
        Destroy(toClear[i]);
}
```

5.1.2 Save/Load

The current state of the system can also be stored on disk, to be imported back into the simulation at a later date. This is done via the use of three dump .txt files: one for the random seed, one for the blobs and their parameters (ID, energy, position, direction of movement, and DNA), and one for the food pellets, for which only the position is stored. To note is that the same file loaded into simulations running on two different machines may behave differently due to the inherent stochastic nature of the algorithms used. The simulation state is also output as a JSON file in order to support a more standardised output. JSON was chosen because it is language-independent and can be imported in a multitude of external applications.

5.2 Food Logic

The food pellets are stored inside a list in the FoodManager class; this also contains parameters such as the amount of energy a pellet provides. In order to impose an artificial cap to the population, the maximum food is limited to 500 units.

5.2.1 Clustering

Rather than being randomly scattered on the plane, food is grouped into clusters. A cluster is a (coordinate, time-stamp) pair indicating where food may spawn. In order to ensure variation in the food density, clusters are uniformly distributed. A number of clusters are created and scattered on the plane, each with its own lifetime, varying between 60 and 120 simulation ticks. On every simulation tick, the cluster returns its state, either true or false, depending on the time elapsed since the cluster was created. The FoodManager then checks whether clusters have expired and replaces the inactive ones.

```
for (int i = 0; i < foodclusternum; i++)
{
    if (clusters[i].UpdateTimer() == false)
    {
        clusters.Remove(clusters[i]);
        clusters.Add(new Cluster());
    }
}
```

5.2.2 Food Placement

If the current number of food pellets is smaller than the maximum, food is allowed to spawn, with respect to the same delay mechanic previously explained in Section 5.1. Once a pre-existing food cluster is acquired, a food pellet is created using Unity's Instantiate method and placed at a random position around the centre of the cluster. This method takes as parameters: a GameObject (in this case, the base object), a three-dimensional Vector pointing towards a position in space, and a Quaternion indicating rotation. A Quaternion can be interpreted mathematically as the quotient of two vectors, as defined by its inventor, William Rowan Hamilton [14]. The food pellet is then added to the list contained in the manager class. Below, the algorithm for spawning can be observed.

```
t += Time.deltaTime * 1000;
if (t > spawnspeed && foods.Count < maxFood)
{
    // get random cluster
    int cluster = Main.random.Next(0, foodclusternum);
    // aquire X and Y of cluster
    // add random to make it off centre
    float xpos = clusters[cluster].getx() + (((float)Main.random.NextDouble() -
        0.5f) * foodSpawnDiameter);
    float ypos = clusters[cluster].gety() + (((float)Main.random.NextDouble() -
        0.5f) * foodSpawnDiameter);
    // create food pellet
    GameObject clone = GameObject.Instantiate(food, new Vector3(xpos, ypos), new
        Quaternion(0, 0, 0, 0)) as GameObject;
    foods.Add(clone);
    t = 0f;
}
```

5.3 Blob Logic

Blobs are the primary actors inside the simulation. They are stored in a list in the BlobManager class. A blob GameObject encapsulates two other classes, specifically BlobLogic and BlobDNA. While the latter is only a wrapper class which contains the DNA string of a particular individual, the former contains most of the interaction logic and the parameters. The function run when a blob GameObject is instantiated is seen below.

```
void Start ()
{
    // load Unity components
    rb = blob.GetComponent<Rigidbody>();
    audioSource = blob.GetComponent<AudioSource>();

    DNAaux = 0; // current position in DNA string
    // levytime = first DNALEVYTIME bits
    this.levytime =
        (System.Convert.ToInt32(blob.GetComponent<BlobDNA>().getDNA().Substring(
        DNAaux, DNAOperations.DNALEVYTIME), 2));
    DNAaux += DNAOperations.DNALEVYTIME;
    // toReproduce = next DNAREPROD bits
    this.toReproduce =
        ((float)System.Convert.ToInt32(blob.GetComponent<BlobDNA>().getDNA().Substring(
        DNAaux, DNAOperations.DNAREPROD), 2)) + 100f;
    DNAaux += DNAOperations.DNAREPROD;
```

```
    this.blobID = ID++;

    // switch audioclip
    audioSource.clip = drop;
    if (Main.hasSound)
    {
        audioSource.Play();
    }

    for (int i = 0; i <= levythreshold; i++)
    {
        ate.Add(Main.globaltimestamp);
    }
}
```

Inside BlobLogic, during its initialisation, a blob's AudioSource and Rigidbody components are acquired. Due to a technical limitation that only allows the mapping on one sound to each object, a swapping technique is used in order to utilise multiple sound files. The Rigidbody stands as the "physical" body of the blob; it is used in movement and collision detection. At the same time, a blob's characteristics are extracted from its DNA string. Finally, a blob is forced to perform a localised search for a number of ticks determined by his genome. After that, it will perform a Lévy walk. The movement of blobs is explained in more detail in Section 5.3.3.

5.3.1 DNA Encoding

The DNA string is a 16-bit number encoding two intrinsic characteristics used in a blob's behavioural algorithm. The upper eight bits encode the "patience", that is the time required to pass since a blob last ate until it stop searching locally for food. The lower eight bits encode the "reproduction threshold", which specifies the energy required for a blob to reproduce. A constant is added to this number in order to prevent destructive mutations. The reproductive process is detailed in Section 5.3.5.

5.3.2 Update

During a simulation tick, all blobs execute their Update method. As Unity is a game engine, the default Colliders implement physically-modelled collision mechanics; this involves treating blobs as solid physical objects. Due to the fact that the prevention of overlap between blobs can propel them in unwanted directions, the blob's rotation is set to the identity Quaternion and the velocity and angular velocity to the zero vector. In order to enforce the initial conditions of the simulation, the Z-component of a blob's position is also set to 0.

```
rb.rotation = Quaternion.identity;
rb.velocity = Vector3.zero;
rb.angularVelocity = Vector3.zero;
rb.transform.position = new Vector3(rb.transform.position.x,
    rb.transform.position.y, 0);
```

Subsequently, the blob's size is determined by its current energy level, and the blue channel of its colour by its patience. This visual display of state allows the user to rapidly identify a blob's current situation.

```
this.GetComponentInChildren<SpriteRenderer>().color = new Color(0, 0,
    ((float)(levytime + colourfix)) / 255, 1);
float blobsize = (this.energy + 50) / 150;
```

The main control loop for a blob's behaviour is as follows: if the simulation is running and enough time has passed since the previous tick, move and lose some energy; if the current energy if higher than the reproduction threshold, reproduce; if the current energy level is negative, starve. The code can be seen below.

```
if (Main.gameState == Main.GameState.Started)
{
    delay = BlobManager.blobspeed;
    t += Time.deltaTime * 1000;

    if (t > delay)
    {
        Move();
        t = 0f;
        energy -= Main.hostility;
    }

    if (this.energy > toReproduce)
    {
        Reproduce();
    }

    if (this.energy < 0f)
    {
        Starve();
    }
}

void Starve()
{
    Destroy(this.gameObject);
    BlobManager.blobs.Remove(this.gameObject);
    Destroy(this);
}
```

5.3.3 Movement

As the plane is bounded, blobs automatically turn around should they encounter an edge. As opposed to a toroidal grid, blob movements are easier to track on a bounded plane. Since the return angle is random, it is possible for a blob to actually "fall off" the plane.

```
float turn = FoodManager.foodSpawnSize / 2;
if (rb.transform.position.x <= -turn || rb.transform.position.x >= turn ||
   rb.transform.position.y <= -turn || rb.transform.position.y >= turn)
{
    angle = angle - Mathf.PI - Mathf.PI * (float)(Main.random.NextDouble() -
       0.5);
}
```

After the initial check, the method LevyUpdate is called, which updates a list containing timestamps encoding the event of a blob eating food. Only the latest few are stored.

```
void LevyUpdate()
{
    for (int i = 0; i < ate.Count; i++)
        if (ate[i] < Main.globaltimestamp - levytime)
```

```
                ate.Remove(ate[i]);
}
```

If at any point the list is empty, the blob performs a Lévy walk in search for a new area with higher food density. This type of move is performed by choosing a random angle and heading in that direction until food is encountered. A check variable, isLevy, is used to ensure a blob does not start another Lévy walk. If sufficient food has been consumed recently, a blob will just float in the local area; this is implemented in RandomMove.

```
if (ate.Count < levythreshold)
{
    if (isLevy == false) // if not already in Levy
    {
        // pick random direction
        angle = Random.Range(0, 361) * Mathf.PI / 180;
        isLevy = true;
    }
    // move the direction set by "angle"
    rb.transform.position += new Vector3(speed * Mathf.Cos(angle) / 10, speed *
       Mathf.Sin(angle) / 10);
}
else
{
    // should not levy
    isLevy = false;
    RandomMove();
    // float around in the local area
}
```

5.3.4 Eating

Blobs eat by colliding into food. The pellet is deleted, a sound is played, and energy is added to the colliding blob. The blob also records the timestamps of when it has eaten recently.

```
void OnTriggerEnter(Collider other)
{
    if (other.tag == "Food")
    {
        Eat();
        Destroy(other.gameObject);
        FoodManager.foods.Remove(other.gameObject);
        Destroy(other);
    }
}

void Eat()
{
    audioSource.clip = chomp;
    if (Main.hasSound)
    {
        audioSource.Play();
    }
    this.energy += FoodManager.foodEnergy;
    ate.Add(Main.globaltimestamp);
}
```

5.3.5 Reproduction

Blob reproduction is done asexually in order to reduce the behavioural complexity of having to find a partner. When the energy reaches the reproduction threshold, a blob enters a reproductory state. During this period, two child blobs are instantiated with their parent's DNA, and with half of the parent's energy, in order to mimic cellular division. Each child is added to the list of blobs in BlobManager, and the parent is destroyed. The Reproduce method can be found below.

```
void Reproduce()
{
    this.state = 2;
    GameObject child1 = GameObject.Instantiate(blob, blob.transform.position,
       blob.transform.rotation) as GameObject;
    child1.AddComponent<BlobDNA>();
    child1.GetComponent<BlobDNA>().setDNA(DNAOperations.mutate(
       blob.GetComponent<BlobDNA>().getDNA()));
    child1.GetComponent<BlobLogic>().energy = this.energy / 2;
    BlobManager.blobs.Add(child1);

    GameObject child2 = GameObject.Instantiate(blob, blob.transform.position,
       blob.transform.rotation) as GameObject;
    child2.AddComponent<BlobDNA>();
    child2.GetComponent<BlobDNA>().setDNA(DNAOperations.mutate(
       blob.GetComponent<BlobDNA>().getDNA()));
    child2.GetComponent<BlobLogic>().energy = this.energy / 2;
    BlobManager.blobs.Add(child2);

    Destroy(blob.gameObject);
    BlobManager.blobs.Remove(blob.gameObject);
    Destroy(blob);
}
```

5.4 Deployment

The system can be built through the Unity Engine to run on a multitude of platforms ranging from Windows, Linux, and MacOS, as a standalone application; to web browsers through the Unity Web Player plug-in. To note is that the Web Player is unable to run in Google Chrome, as support was dropped with version 39 of Chrome. A web-based deployment is temporarily available on-line through Amazon Web Services (AWS) at `https://s3-eu-west-1.amazonaws.com/ui-v3/webtest.html`.

5.5 Summary

This chapter detailed the implementation of the system, together with snippets of code to assist in the explanation. The next Chapter deals with the evaluation of the system and the feedback received.

Chapter 6

Evaluation

This chapter discusses the various methods used for testing and evaluating the system. This was done in three stages, starting with internal testing, continuing with a series of experiments performed on the system, and finalising with a user evaluation questionnaire taken by both students and researchers in different fields of biology. To note is that, due to the stochastic nature of the simulation, testing the system becomes difficult. Even identical initial conditions will result in visibly different outcomes after a matter of seconds. This can be seen in Figure 6.1. Due to this, testing on a microscopic scale becomes impossible. The testing has to be carried out on a macroscopic level, while ensuing that the same patterns emerge from similar initial conditions.

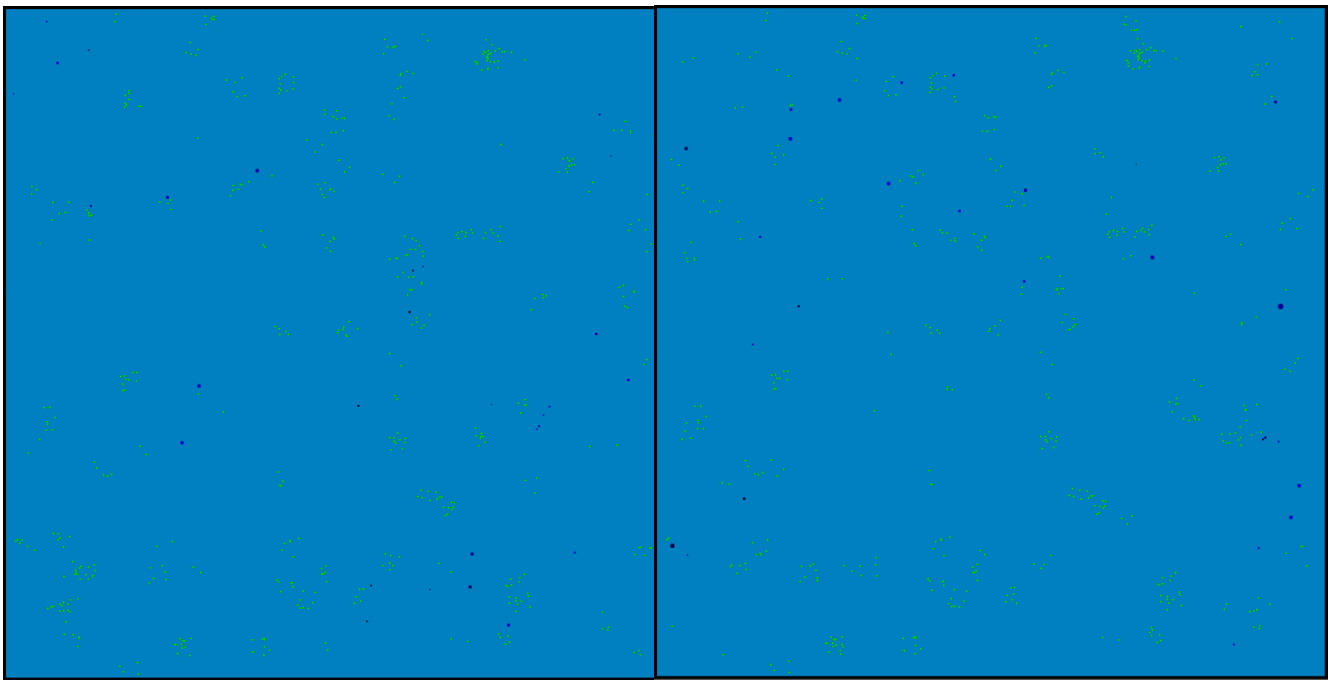

Figure 6.1: Different outcomes from same initial conditions; 20 second run

6.1 Application Testing

As Unity is primarily a game engine, part of the testing performed was visual testing. This involved running the simulation and observing any potential bugs. Due to the Agile development process, exploratory testing was

also employed. With a rapidly evolving system, concrete tests would have proved obsolete after a few iterations. Using exploratory testing, new tests are being devised based on the results of the previous.

The unchanging part of the evolution, that is the algorithms handling the DNA sting required more robust testing in order to ensure functionality. These algorithms have undergone unit testing until deemed satisfactory.

6.2 Experiments

A more high-level examination of the system dealt with the observation of the population under experimental conditions. Repeating the experiment multiple times while detecting the emergence of comparable patterns assured the functionality of the system.

The first step of the experiment was to run the simulation without interference. The expected result was an exponential growth in population until food became insufficient, followed by a small decline in the population number. The result was confirmed, as seen in Figure 6.2.

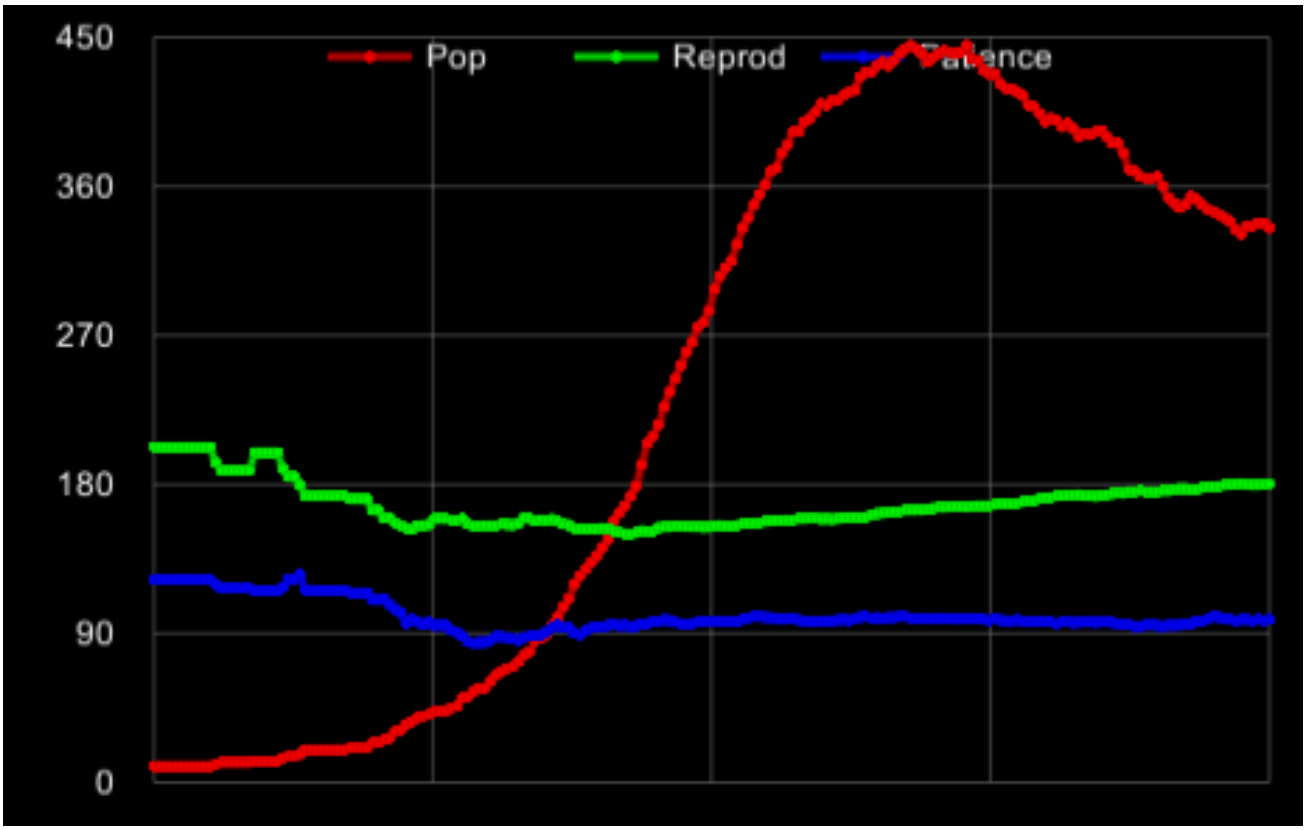

Figure 6.2: Experiment 1: no interference

The next step of the experiment was to confirm the correlation between the availability of food and the population number. As expected, these are directly proportional; the more food available, the more will the population grow. Noteworthy is the tendency of blobs to become more “selfish” as food grows scarcer. With a limited supply of food, the average reproduction threshold increases, causing blobs to require more sustenance until reproduction.

A similar observation can be made about the size of the food clusters. By altering the size of the clusters, the blobs’ patience is affected. Larger clusters tend to evolve more patient populations, as food is more spread out, therefore increasing the merit of spending time in one particular area. Smaller clusters, however, have the opposite effect, as blobs find no reason to linger in an area after the food has been consumed.

The above two steps indicate a capability of blobs to adapt to their current environment.

The final part of the experiment deals with rapid changes in the environmental conditions with respect to the population. Firstly, a sudden influx in population is tested. This causes an even greater increase, as food is being consumed. However, due to unsustainable numbers, the population is rapidly reduced. Over a longer period of time, the population stabilises again. A graph detailing the artificial increase, natural increase, and sharp decrease can be seen in Figure 6.3

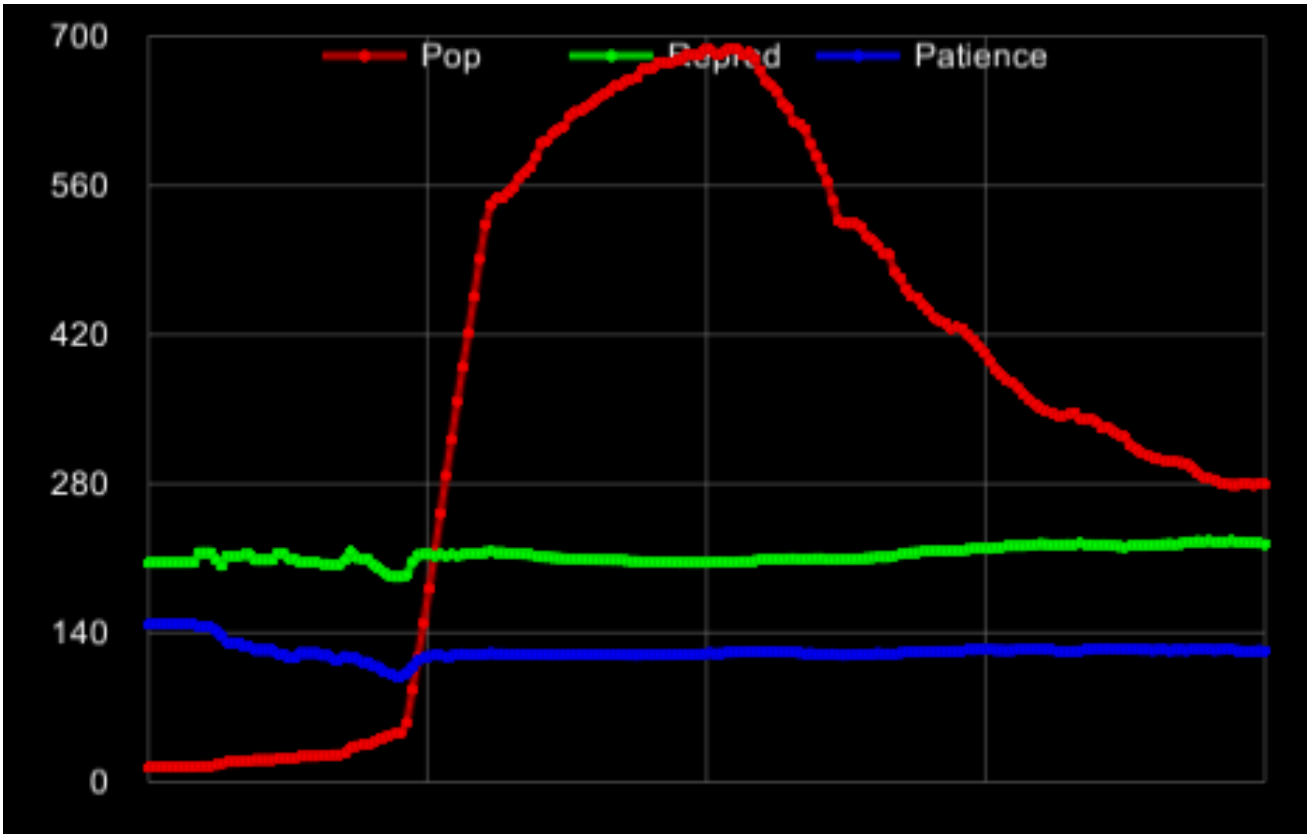

Figure 6.3: Experiment 4: Artificial influx

Another method of influencing the environment is the creation of a hyper-dense food cluster. This would result in a localised population explosion, but a small spike in the overall population number. The only resulting observation was the slight decrease of both reproduction threshold and patience, as local changes can influence the global arithmetic mean.

6.3 User Evaluation

As the project was reaching its final stages, a user evaluation was carried out. This involved the completion of a questionnaire by both fellow students and experts in the field of Biology. The feedback form in its entirety can be found in Appendix B. The purpose of the questionnaire was to measure the usability of the system, together with its capability of conveying useful information about the simulation. The design of the questionnaire was also important, lest the user be confused by any of the questions. A design similar to the System Usability Scale [19] was chosen for the 5-point scale questions regarding usability.

6.3.1 A/B testing

An approach similar to A/B testing, users were presented with three interface versions. Version 3, seen in Figure 6.4, was presented as a control sample. Two other versions, presented in Figures 6.5, and 6.6, containing minor differences, were also presented. Unlike A/B testing, which requires a larger sample size, the results were not compared by the tester; users were asked to select their preferred interface out of the three. The choice was unanimous, all users having selected the control version as their favourite.

Figure 6.4: Version 3, the control version of the interface

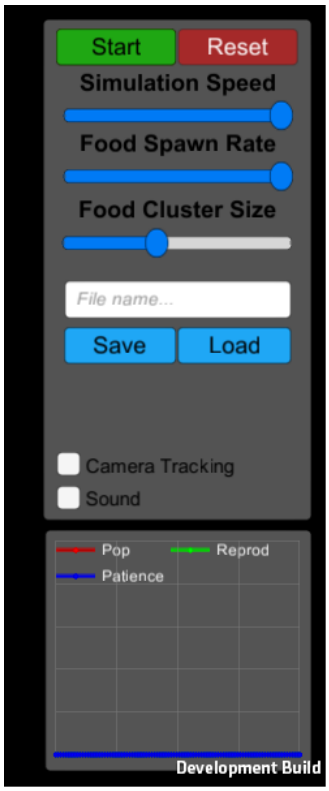

Figure 6.5: Version 1, default interface

Figure 6.6: Version 2, added labels on graph

6.3.2 Feedback Forms

The users were asked to perform the same experimental steps as presented above, in order to enhance their understanding of the simulation. By noting their own observations of the occurrences, the user could test both the functionality and the usability of the system. The user responses to the long answer questions dealing with the observations were in line with the results received from the testing phase.

When asked about the usability of the system, the user responses were mostly positive, as seen in Figure 6.7.

While the interface was labelled as intuitive, some user were displeased with its visual appearance. Figures 6.8 and 6.9 present user responses for the interface-related questions.

Most users were in agreement that the simulation is explicit enough to draw conclusions from, however, it was also felt that more information should be provided. The manner in which information was provided was deemed satisfactory. The user responses for the simulation-related questions can be found in Figures 6.10, 6.11, and 6.12.

Feedback received on the questions concerning the software itself was positive, most users finding the controls

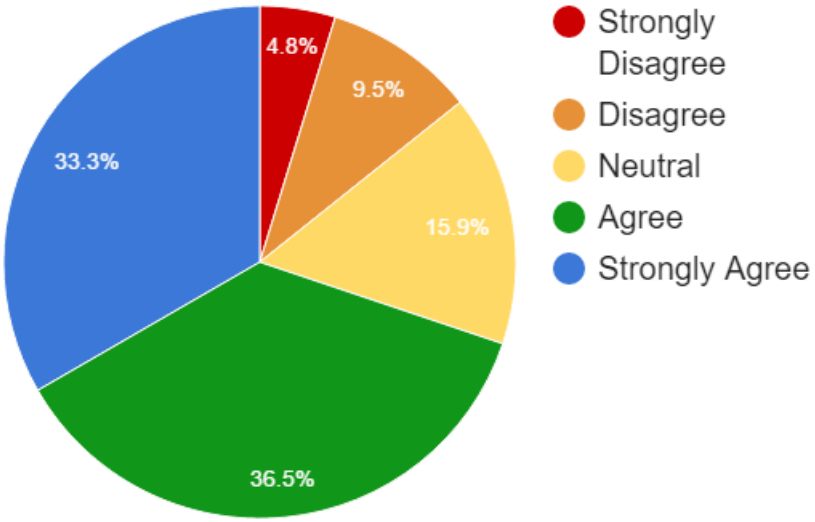

Figure 6.7: An overview of the feedback received

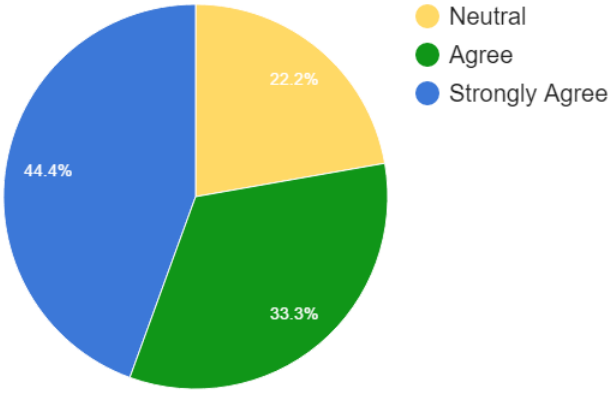

Figure 6.8: Q1: The interface is intuitive

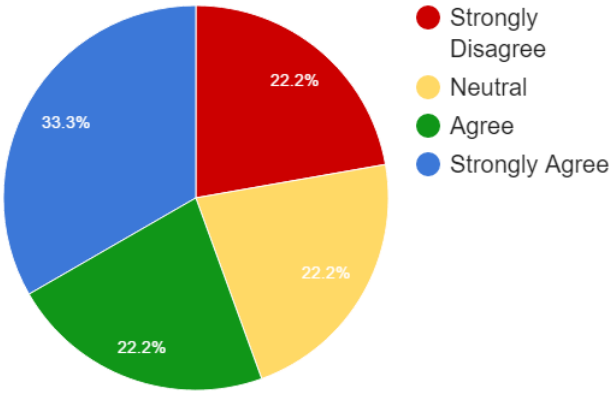

Figure 6.9: Q6: The interface is aesthetically pleasing

easy to understand and use, and the system responsive. The answers can be found in Figures 6.13 and 6.14.

6.3.3 Expert Opinion

Throughout its evolution, the project was reviewed by Dr Simon Hickinbotham, who provided vital feedback for the project's development. His recommendations helped develop the biological side of the system. A crucial suggestion with respect to aligning the simulation to real-life bacteria was the implementation of Lévy walks as the blobs' primary movement pattern.

6.4 Summary

This chapter presented the evaluation process for the system from internal testing, to external user testing through a feedback questionnaire. The next chapter provides a summary of the project, as well as discuss project management issues and potential for further development.

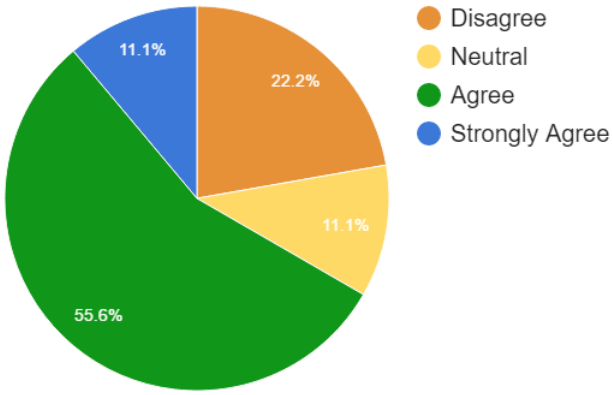

Figure 6.10: Q2: I am able to draw conclusions from the simulation

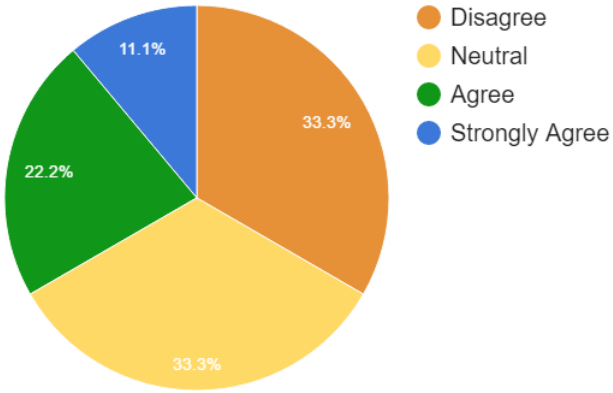

Figure 6.11: Q4: I am given enough information through the interface

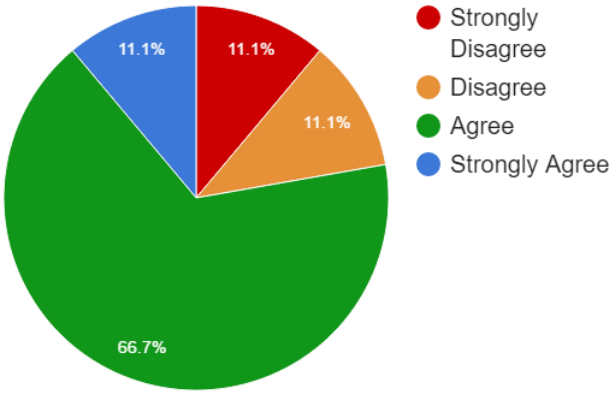

Figure 6.12: Q7: Information is conveyed effectively

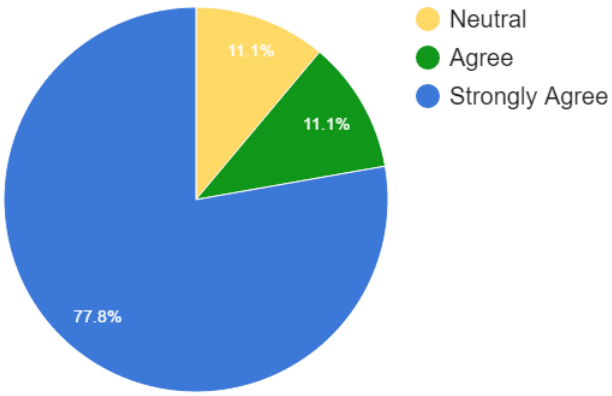

Figure 6.13: Q3: The controls are easy to use

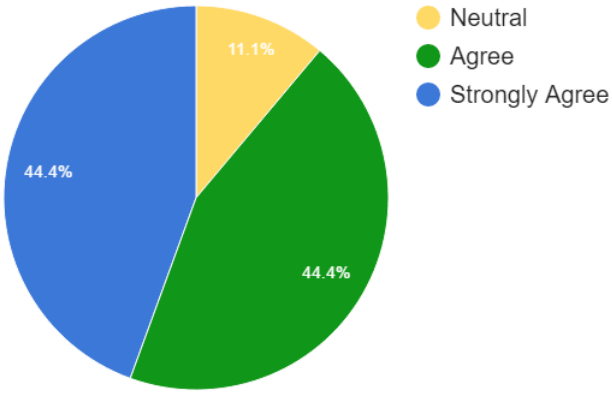

Figure 6.14: Q5: The software is responsive

Chapter 7

Conclusion

7.1 Summary

The main goal of the project was to build an evolutionary simulator with the purpose of a teaching tool; to aid users in understanding the dynamics of an evolving population. Based on the feedback received, this was partially achieved, with the request to provide more information to the users. The web-based deployment allows for lightweight simulation, while the stand-alone system-specific versions offer more performance, together with the saving and loading features.

7.2 Project Management

In order to provide a better approach, the project was split into four main sprints: Requirements, User Interface, Simulation, and Evaluation. The development method was Agile, with weekly iterations. During each supervisor meeting, implemented featured were marked as complete, and a new set of tasks was established for the following iteration. Incomplete features were, based on their importance, abandoned or postponed in favour of more critical ones.

As the development environment was Agile, the importance of source control became paramount. Offering free access to its services, GitHub was deemed suitable for the project. The code can be found in a public repository at `https://github.com/QSilver/UnityBlob`.

7.3 Future Work

Based on the feedback received, the most requested feature was a more in-depth data interface, providing additional information about environmental changes and parameters.

Subsequent developments could include:

- **Increased DNA complexity**
  Blobs only encode two characteristics in their DNA: the reproduction threshold and the patience. Additional parameters would add vast complexity to the simulation.

- **Sexual reproduction for blobs**
  Currently the blobs only divide and mutate their own genome, sexual reproduction would allow for a better transfer of genes through crossover.
- **Behavioural trees**
  At the moment, blob behaviour is a parametrised static algorithm. Using crossover of decision trees, blobs could exchange behavioural patterns directly.
- **External output**
  Since the system can output its current state as a JSON file export, this could be imported by a number of external tools.
- **Horizontal Gene Transfer [17]**
  Based on one of Dr. Simon Hickinbotham's suggestions, blobs could be able to pass on their genes without having to reproduce. Thus, beneficial genes could spread among the population at an increased rate, rapidly leading to more adaptive individuals. Horizontal Gene Transfer is believed to be a significant factor in bacterial drug resistance.

7.4 Reflection

Implementing an evolutionary simulator proved a valuable learning experience. The task required the use of modern tools, such as the Unity Game Engine, paired with solid architectural patterns, such as the Model-View-Controller. Deployment demanded the use of state of the art cloud technology, for which Amazon Web Services were selected. In order to build the simulator, extensive research was required in the fields of evolutionary computing and bacterial cell biology.

Bibliography

[1] Graph maker. https://www.assetstore.unity3d.com/en/#!/content/11782.

[2] Life universal computer. http://www.igblan.free-online.co.uk/igblan/ca/index.html.

[3] Mutation and selection diagram. https://commons.wikimedia.org/wiki/File:Mutation_and_selection_diagram.svg.

[4] Project mono. http://www.mono-project.com/.

[5] Unity game engine. http://unity3d.com/unity.

[6] Darwinbots. http://wiki.darwinbots.com/w/Introduction, 2005.

[7] Gary An, Qi Mi, Joyeeta Dutta-Moscato, and Yoram Vodovotz. Agent-based models in translational systems biology. *Wiley Interdisciplinary Reviews: Systems Biology and Medicine*, 1(2):159–171, 2009.

[8] Kevin Brennan et al. *A Guide to the Business Analysis Body of Knowledger*. Iiba, 2009.

[9] Charles Darwin. *On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*. 1859.

[10] Alex S Fraser. Simulation of genetic systems by automatic digital computers vi. epistasis. *Australian Journal of Biological Sciences*, 13(2):150–162, 1960.

[11] Maria R Fumagalli, Matteo Osella, Philippe Thomen, Francois Heslot, and Marco Cosentino Lagomarsino. Speed of evolution in large asexual populations with diminishing returns. *Journal of theoretical biology*, 365:23–31, 2015.

[12] Martin Gardner. Mathematical games: The fantastic combinations of john conways new solitaire game life. *Scientific American*, 223(4):120–123, 1970.

[13] David Griffeath. *New constructions in cellular automata*. Oxford University Press, 2003.

[14] William Rowan Hamilton. *Elements of quaternions*. Longmans, Green, & Company, 1866.

[15] Simon Hickinbotham. https://www-users.cs.york.ac.uk/sjh/.

[16] John H Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[17] Ravi Jain, Maria C Rivera, and James A Lake. Horizontal gene transfer among genomes: the complexity hypothesis. *Proceedings of the National Academy of Sciences*, 96(7):3801–3806, 1999.

[18] Dan Lessin, Don Fussell, and Risto Miikkulainen. Open-ended behavioral complexity for evolved virtual creatures. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 335–342. ACM, 2013.

[19] James R Lewis and Jeff Sauro. The factor structure of the system usability scale. In *Human centered design*, pages 94–103. Springer, 2009.

[20] Franziska Matthäus, Marko Jagodič, and Jure Dobnikar. E. coli superdiffusion and chemotaxissearch strategy, precision, and motility. *Biophysical journal*, 97(4):946–957, 2009.

[21] Paul T Oliver. Goopies. `http://forum.codecall.net/topic/72637-goopies-evolving-neural-networks-wip/?p=643047`, 2012.

[22] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and available at `http://www.gp-field-guide.org.uk`, 2008.

[23] Raymond Rappaport. *Cytokinesis in animal cells*, volume 32. Cambridge University Press, 2005.

[24] Trygve Mikjel H Reenskaug. The original mvc reports. 1979.

[25] Stewart Robinson. *Simulation: the practice of model development and use*. Palgrave Macmillan, 2014.

[26] Karl Sims. Evolving virtual creatures. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 15–22. ACM, 1994.

[27] Alan M Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950.

[28] Gandimohan M Viswanathan, Sergey V Buldyrev, Shlomo Havlin, MGE Da Luz, EP Raposo, and H Eugene Stanley. Optimizing the success of random searches. *Nature*, 401(6756):911–914, 1999.

Appendix A

User Stories

I want to be able to see the ”blobs” in their environment, so that I am able to study their behaviour.

I want to be able to pan and zoom the display area, so that I am able to better observe the environment.

I want to be able to inspect an individual, so that I am able to predict its behaviour.

I want to be able to pause and restart the simulation, so that I may analyse individuals.

I want to be able to reset the simulation, so that I am able to see different evolutions under the same conditions.

I want to be able to tweak parameters in the environment, so that I am able to influence the evolution.

I want to be able to see statistics about the environment and the simulation, so that I am able to measure the effect of changes.

Appendix B

Feedback Form

asd