Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Building an Artificial Life Simulator

Author:
Jia Qi Poon

Supervisor:
Dr. Edward Johns

Second Marker:
Professor Andrew Davison

June 22, 2023

Abstract

Artificial life simulators are complex pieces of software that leverage evolutionary algorithms in an attempt to explore how life works by modelling biological processes. While many simulators have been built over the years, most of them are desktop-based and not easily accessible to users.

We build a browser-based life simulator that features complex simulation mechanics that are user customisable. Organisms can have neural networks that evolve across generations, leading to some interesting behaviours. The webpage loads in one second and has a Google Lighthouse performance score of 95, which corresponds to a 'Good' rating. It also supports up to 200 organisms at once while running at 30x real-time speed.

We also make use of the simulator to investigate if multi-agent collaborative behaviour can emerge naturally using genetic algorithms. While collaborative behaviour was not observed, organisms were noted to clump together and learned to better move towards food over many generations. We then theorise reasons for such observations.

Acknowledgements

I would first like to thank my supervisor, Dr. Edward Johns, for his patience and continued support throughout this project. Your feedback and insights have inspired many elements of this simulator.

I would also like to thank my friends who have accompanied me on this journey throughout my years at Imperial. Your encouragements, advice and presence will be dearly missed. Special thanks to Siting Zhao for her unwavering friendship and support during tough times.

Lastly, I would like to thank my family for their constant encouragement over the past few years. You have influenced me to always give my best in the things that I do and taught me to be a better person.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Artificial Life is a field of study that tries to understand how life works through abstracting and modelling biological processes (3). Although primarily used to study biological processes, the concepts and ideas learnt from artificial life have been applied to other fields, from robot control (4), to traffic simulators (5), to scheduling problems (6).

While many artificial life simulators such as EvoSim (7), REvoSim (8) and Artificial Life Environment (ALIEN) (9) exist, most of them are desktop-based applications, which require the user to download the program and set it up. This means that they are far less accessible and more troublesome to use as compared to a browser-based simulation.

This project solves this problem by offering a browser-based artificial life simulator that is well featured, easy to use and available online at jqpoon.com/life.

The artificial life simulator models organisms living in an environment that evolve their physical attributes and brains over time using genetic algorithms. Through simple sliders, simulator parameters like the mutation rate and food spawn rate can be controlled by users investigating different scenarios. Users can also build and customise their own organisms using the organism builder.

A simple and intuitive user interface enables users to explore their own scenarios and draw their own conclusions. They can also make use of the in-built tutorial to learn about the different features of the simulator.

Overall, this project is different by allowing users to be able to draw their own conclusions through a simple to use but powerful web-application. Using this simulator, a key research question is explored: Can multi-agent collaborative behaviour emerge naturally using genetic algorithms?

1.1 Ethical Considerations

1.1.1 Applications of Project

As this project is a life simulator, it is a tool that can be potentially used for nefarious purposes. For instance, although exceedingly unlikely, it can be used to conduct research on viruses and how they can evolve and spread.

This project purely focuses on the basic evolution of organisms. There are no specific applications (neither civilian nor military) for this project, as it is a foun-

dational machine-learning research project.

1.1.2 Use of Third Party Libraries

In any modern software product, third party libraries are almost always used to aid in the development process. This allows the developer to focus on building their application instead of reinventing the wheel. Like many others, this project relies on the use of a few third party libraries to support development.

Third Party Library	Use	License
Phaser 3	Game framework used to build simulator	MIT License (10)
Rex Plugins	UI Components for Phaser	MIT License (11)
Webpack	Compile Javascript/Typescript files	MIT License (12)
Jest	Javascript testing framework	MIT License (13)
Bootstrap	Frontend library	MIT License (14)
Chart.js	Build graph visualisations	MIT License (15)

According to St Laurent, a lawyer who is well versed in software licensing issues, the MIT License is a straightforward license that allows the licensee to make use of the software with almost no limitations. This includes the "exclusive right to commercially exploit the work and to develop derivative works from the work" (16). The licensee need only provide a copy of the license in "copies of the software" or in "substantial portions" of it. As all of the third party libraries explicitly used in this project are governed by the MIT License, there are few, if any, licensing concerns.

However, each external dependency might have their own dependencies with varying licenses. This project assumes that the license listed by the third party library can be taken at face value, without further investigation. If this project were to be commercialised, more research has to be done to investigate the sub-dependencies and if it remains legal to use these third party libraries.

Chapter 2

Background

2.1 Evolutionary Algorithms

Evolutionary algorithms are algorithms that are used to solve complex problems in computing and include genetic algorithms, genetic programming, evolutionary programming and evolutionary strategies (17).

It is based heavily on the theory of evolution by Darwin. In nature, organisms compete against each other in an environment to survive, and the fittest will win. This means that organisms with characteristics better suited for the environment will reproduce and produce offspring with similar characteristics. The same theory is applied in evolutionary algorithms, where solutions which are better suited for the problem are preferentially chosen to produce the next generation of solutions. By repeating the process many times, a solution that solves a particular problem very well can be obtained.

2.1.1 Genetic Algorithms

Genetic algorithms are a subset of evolutionary algorithms based on the idea of natural selection. It is a 'population based algorithm' with several key features such as encoding scheme, selection, crossover and mutation (18). This section looks at key aspects of genetic algorithm in the context of an entity surviving in an environment.

Encoding Scheme

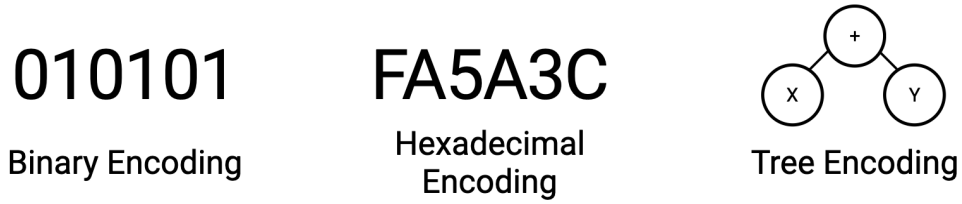


Figure 2.1: Different encoding schemes

Each entity has to be expressed using some kind of encoding. This can also be known as a genotype or genome (19), but they are essentially the same idea.

Together with the environment, the genotype of an entity determines its physical attributes and behaviour.

There are a whole variety of encoding schemes, from binary, to hexadecimal, to octal, to tree encodings (18). Depending on the environment/context, certain encoding schemes may be more suitable.

Binary encoding is the simplest type of encoding, as it just consists of 1s and 0s in a string. It is commonly used as it makes the implementation of crossover and mutation functions simpler and faster to execute. However, it may not be as expressive as other encoding schemes, owing to the fact that there are only two possible values for each character.

Hexadecimal encoding is similar to binary encoding. As its name suggests, it uses hexadecimal characters in a string instead of binary characters. This allows more information to be packed into a denser representation of an entity. It is the encoding scheme used in some simulators explored in Section 2.4, such as EvoSim.

Tree encoding represents genotypes in the form of a tree. It is most often used to represent evolving programs (18).

Selection

Selection refers to the method in which entities are chosen to use for breeding the next generation. In nature, this is determined through the interactions between different entities. Depending on their physical attributes and behaviour, entities that are better suited to the environment they are in will survive for longer than those ill-suited to the environment. These longer surviving entities will then have more opportunities to breed and produce the next generation. This process is also known as natural selection.

In genetic algorithms, there are different types of selection methods, such as roulette wheel selection, Boltzmann selection, tournament selection, rank selection, and more (20). Since this project relies on natural selection to select entities for breeding, there is no need to implement any of these selection methods.

Crossover

Crossover functions refer to the method in which genomes from two parents are mixed to generate the offspring's genome. As such, this can only be done if there is sexual reproduction (i.e. two parents). Furthermore, a crossover probability can be set to determine if a crossover function should be executed. Like selection, there are a wide variety of crossover functions. Some examples include single point crossover, double point crossover and uniform crossover (20).

In single point crossover, a single point in the parents' genome is chosen. Any genetic material between that point and the end are swapped between parents to produce the resultant offspring genomes. An example of this is illustrated in Figure 2.2.

In double point crossover, two points are chosen in the parents' genome. Genetic material between those two points are then swapped to produce the resultant offsprings' genomes. Figure 2.3 demonstrates an example of this. This idea can be extended to k point crossover, where k number of crossover points are selected. Crossover occurs within those crossover regions, in a way similar to double point crossover.

Instead of treating the parent genome as a continuous stream, in uniform crossover,

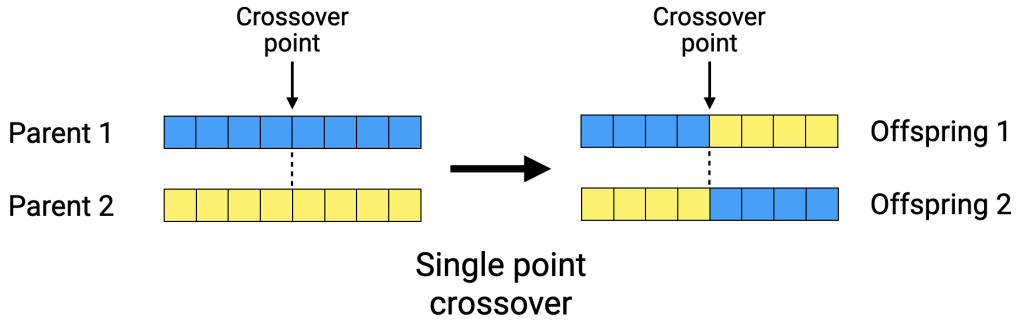


Figure 2.2: Single point crossover example

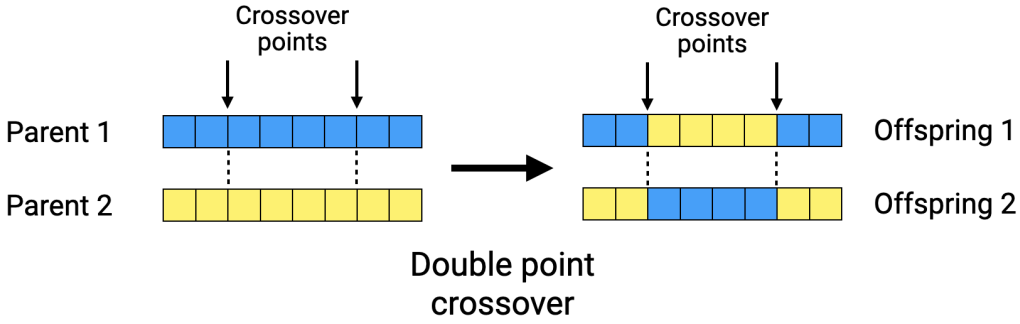


Figure 2.3: Double point crossover example

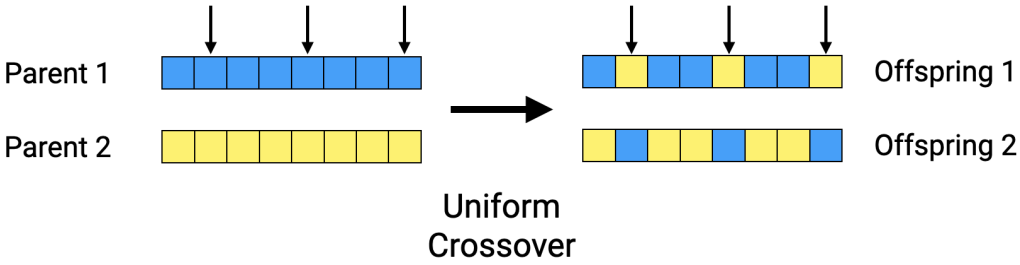


Figure 2.4: Uniform crossover example

each gene is treated separately. Genes are selected at random to be swapped between both parents, as seen in Figure 2.4.

Mutation

Another way to introduce diversity is through mutation. Mutation refers to randomly altering a single gene in an entity's genome (in the context of a binary encoding scheme, this refers to a single bit in a sequence of bits). Each gene has a mutation probability and mutates independently of other genes. Mutation can also be performed throughout an entity's lifetime, instead of only during reproduction. Several mutation techniques exist, but the most common one is inversion mutation, also known as point mutation or flipping mutation (19).

In inversion mutation, if a specific gene has been chosen to mutate (randomly, with a mutation probability), it will be changed to a value in the set of all possible values it could take. For instance, if an entity is encoded using a hexadecimal encoding scheme, then it will randomly chose one of sixteen possible characters to

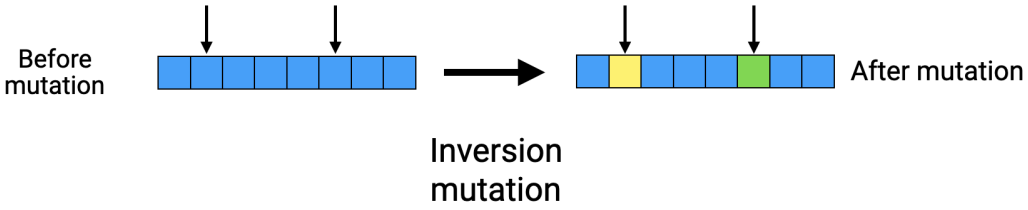


Figure 2.5: Inversion mutation example

mutate into.

The mutation probability of entities is essential. Large mutation rates can lead to good genomes existing for a short period of time only, while small mutation rates can stifle diversity (19).

2.1.2 Relevance to this project

As mentioned earlier in the introduction, many artificial life simulators incorporate some form of evolutionary algorithms to develop a diversity of entities. Genetic algorithms are the most common form of evolutionary algorithms implemented in simulators, as they are after all based on how organisms evolve in real life. This project will include some of the concepts described above.

2.2 Neural Networks

Neural networks are a type of machine learning model that take in a set of inputs, return a set of outputs and are typically used to model a complex function. They are powerful models, widely used in many different tasks from object detection to large language models. At the heart of complex neural networks is the simple perceptron.

2.2.1 Perceptron

Perceptrons consist of a single node, a set of inputs (x_i), a set of weights (w_i), a bias (b) as well as an output (y), as shown in Figure 2.6.

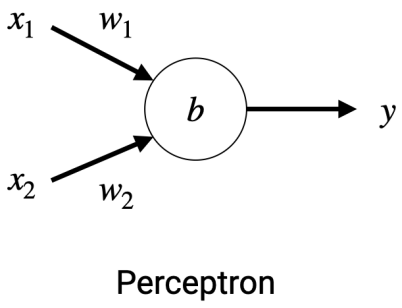


Figure 2.6: Model of a perceptron

To compute the output of a perceptron, each input is multiplied with its corresponding weight and summed up together with the perceptron's bias. This act of calculating its output is also known as a forward pass.

$$y = \sum_i x_i w_i + b \tag{2.1}$$

2.2.2 Activation Functions

Activation functions are functions that are applied to a perceptron's output, and are normally used to add non-linearity to a perceptron. This allows networks of perceptrons to be able to model more complex functions, as they are not just limited to the linear combination of weights and biases. Furthermore, activation functions such as **tanh** and **sigmoid** restrict the range of outputs, which helps to avoid extremely large values being output.

A few common activation functions are shown in Table 2.1.

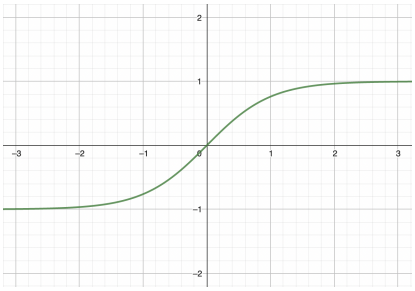
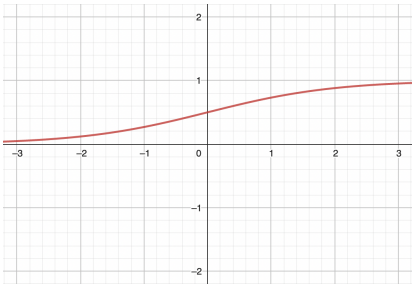
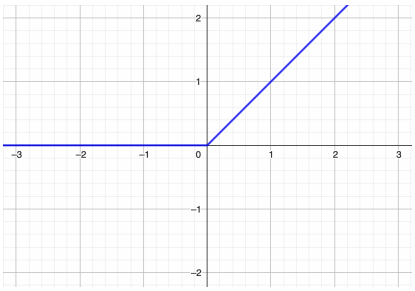
Activation Function	Formula	Graph
tanh	$f(x) = \frac{e^{2x}-1}{e^{2x}+1}$	
sigmoid	$f(x) = \frac{1}{1+e^{-x}}$	
ReLU	$f(x) = \begin{cases} 0 & x < 0 \\ x & \text{otherwise} \end{cases}$	

Table 2.1: Common Activation Functions

2.2.3 Multi-layer Perceptrons

To build more complex networks, perceptrons can be chained together to form a multi-layer perceptron network, as seen in Figure 2.7. Each 'layer' in a MLP network comprises of a set of perceptrons, all with their own weights and biases. Layers are then stacked on top of one another, with every perceptron connected to every other

perceptron in adjacent layers. This is also known as a fully connected network due to how the layers are linked together (21).

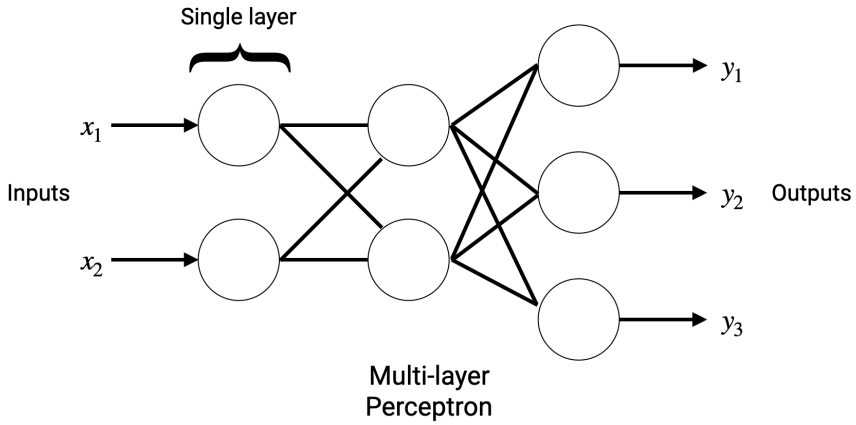


Figure 2.7: Model of a Multi-level Perceptron network

The multi-layer perceptron network is much more powerful than the simple perceptron, as the multiple perceptrons linked together can represent significantly more complex functions than a single perceptron.

2.2.4 Improving Neural Networks

Neural networks are typically incrementally improved upon by using a method known as **backpropagation**. This method calculates the loss (i.e. the difference between the current output and the intended output) of a neural network, then updates the weights and biases of the neural network using derivatives. However, backpropagation is not used in this project, as the focus is on genetic algorithms.

Another important aspect to consider in neural networks is the initialisation of weights and biases. If they are zero-initialised, the network will be useless, as it will just output zeros for any inputs. Random initialisation of these values is ideal, as it allows for different networks to be created. This is desirable as it enables some networks very closely model the target objective function from the start.

However, if the starting values are too large, it is possible to saturate the activation functions, which is not ideal. This would mean that the output of a perceptron would always be 1 or -1 (e.g. if the *tanh* function is used), since a large weight value would mean that the inputs to the activation function is large. The Glorot initialisation scheme (22), as defined by Equation 2.2 is a well known initialisation method that is widely used.

$$W_{ij} \sim U[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}] \tag{2.2}$$

2.3 Typescript Libraries

2.3.1 Rendering Libraries

As this simulator is browser-based, a decision was made early on in the project to use an external engine, instead of manually creating a library to render graphics. A

few options were considered, namely **p5.js**, **Pixi.js** and **Phaser**.

p5.js is a library for "creative coding" (23), which means using code to draw graphics on a canvas. It features a variety of drawing functionalities like drawing different shapes with fills, on top of methods for handling mouse input. However, it lacks more advanced features like collision detection between objects.

Pixi.js is a rendering library used to create graphics (24) and supports WebGL, a Javascript API for "rendering high-performance graphics within any compatible web browser" (25). It is highly performant, and also has support for drawing shapes easily and even sprites in their API. However, like p5.js, it lacks a physics system to easily simulate object movement and collision.

Phaser is a game framework that was originally built on top of Pixi.js. It is also designed to be highly performant, as it can handle hundreds of objects colliding without lagging (26). On top of supporting drawing graphics, it also features a physics system called Arcade (27). This allows developers to easily move objects around the canvas by calling a few API methods. As a game framework, Phaser also has support for handling user input and methods to pass information between game objects.

Phaser was chosen over the other two, owing to the ease of use of the library and its many features. This helped to reduce development time, as complex calculations like object collision and overlap could be handled by the game engine. The presence of a physics system was a key reason why Phaser was chosen over the other libraries as well.

2.3.2 Neural Network Libraries

In order to use neural network to model an organism's brain in this project, a few options were considered. The library's ease of use, speed, size and maintainability were all considerations when deciding which one to incorporate into the project.

Brain.js is a javascript library that supports GPU accelerated neural networks. It is designed to be fast and simple to use and supports a few types of neural networks. For example, one can create a neural network to learn the `xor` functions in just a few lines, as seen in Source Code 1. It is also relatively configurable, with custom neural network sizes, activation functions, learning rates and more available for the user to change. Furthermore, it is decently well maintained, with its latest commit made two months ago (28).

Like brain.js, **Synaptic** is another javascript library used to build machine learning models. It supports a few predefined architectures like Multi-layer Perceptron networks (MLPs), Long Short-Term Memory (LSTMs), among others. Furthermore, developers can configure how network layers connect between each other, which allows them to create complex networks. However, Synaptic does not support GPU accelerated networks, which potentially means a longer training time for more complex tasks. In addition, it is not well maintained, with the latest commit made four years ago (29).

Tensorflow.js is another neural network library written in javascript. It is based off the popular Python library Tensorflow and has a very similar API to its Python equivalent. It is immensely powerful, featuring a very wide range of architectures and layers, just like its Python counterpart. Like Brain.js, it can also utilise the WebGL framework to execute operations on the GPU. Despite its increased functionalities,

```
// create a simple feed forward neural network with backpropagation
const net = new brain.NeuralNetwork({hidden layers: [3]});

net.train([
  { input: [0, 0], output: [0] },
  { input: [0, 1], output: [1] },
  { input: [1, 0], output: [1] },
  { input: [1, 1], output: [0] },
]);

const output = net.run([1, 0]); // [0.987]
```

Source Code 1: Example of how to use brain.js (28)

the Tensorflow.js package is still relatively small and is comparable to the size of the Brain.js library. Furthermore, a study of Javascript-based deep learning libraries found Tensorflow to be the most well featured library based on its support layer types, activation types, optimizer types and developer support (30). However, one downside is that it is more complex to use and requires more configuration to use properly. For example, Tensors (arrays in the GPU) need to be manually deleted to free its memory if the WebGL backend is used (31).

2.4 Related Works

Artificial life simulators typically involve a small number of organisms in an environment in which they are free to move, consume food (and potentially other organisms), reproduce and more. Evolutionary algorithms like genetic algorithms are normally used to alter the physical characteristics and behaviour of organisms from one generation to the next, in order to produce diversity in the population. Depending on the complexity of the simulator, complex behaviours like symbiosis and collaboration between organisms can also be modelled.

Simulators have a large advantage over investigating such processes in real life. Since evolution and other biological processes typically occur on a very large time scale, simulators allow researchers to observe these processes in a much shorter time frame. Furthermore, it is often cheaper and more resource efficient to run a simulator, as compared to real life experiments. They also provide researchers easier access to data about their entities, while offering repeatable and controllable experiments.

2.4.1 Cellular Automaton Models

Many different artificial life simulators have been developed over the years to solve a variety of problems. Most of them are based on Cellular Automaton (CA) models - grid-based environments where each cell represents a certain organism.

There are rules that govern the interactions between neighbouring cells. These rules typically involve how a cell of interest will react in the next time-step, depending on the number of neighbouring cells and the type of neighbouring cell. For

instance, in John Conway's famous "Game of Life", a few simple rules led to a huge range of creative organisms like gliders and oscillators (32).

This section looks at examples of artificial life simulators that are built using the Cellular Automaton model.

EvoSim (7)

EvoSim is a standalone, desktop-based application that was developed to explore the evolution of organisms in a Cellular Automaton-like environment. A screenshot of the EvoSim's interface can be found in Figure 2.8.

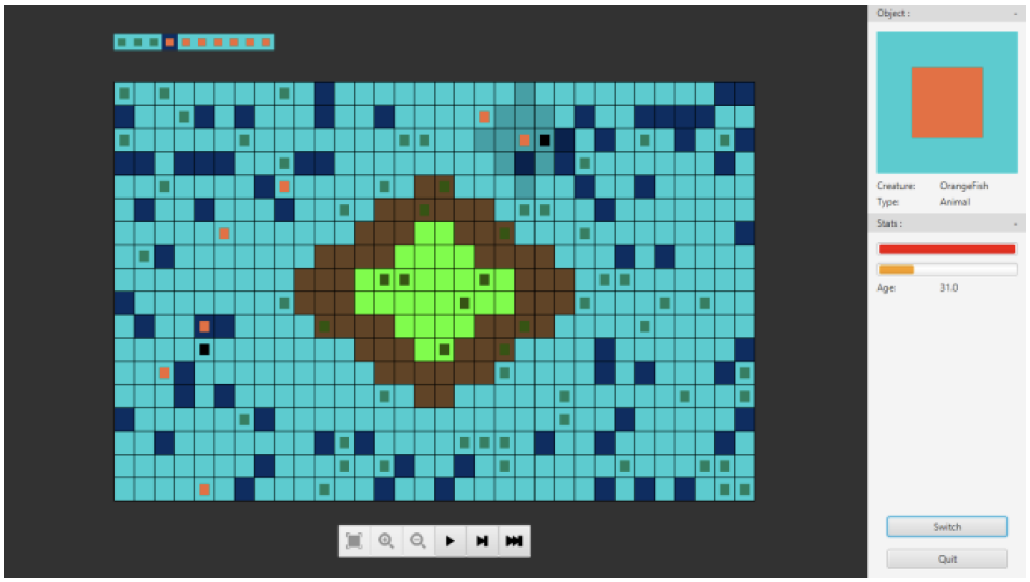


Figure 2.8: Screenshot of EvoSim simulator

Organisms in the simulator make decisions based on a sense-think-act model. There are systems such as health, age and energy levels that govern the life cycle of each organism. Each organism has the ability to move around, attack other organisms and breed, amongst other actions.

Each organism's physical attributes are governed by their genetic code, which is a 80-character long hexadecimal string. This is then expressed by the organism through its physical attributes and behaviour. For example, the first six hexdecimal characters control the colour of the organism. Over time, each organism's genetic code mutates randomly, leading to different behaviours evolving.

The simulator also features biomes. Each tile has a distinct biome (e.g. sea, forest, etc) - this affects the movement speed of organisms, spawn rate of plants and other simulator dynamics.

Overall, EvoSim is a well-designed artificial life simulator with many interesting features like biomes and genetic codes. These features serve as an inspiration for this project's simulator implementation.

However, it is a desktop-based simulator that is not as easily accessible as an online simulator. Furthermore, as a Cellular Automaton based simulator, the movement of organisms and their interactions are discretised, which could limit the organisms' complexity of behaviour.

Rapid Evolutionary Simulator (REvoSim) (8)

REvoSim is another Cellular Automaton based simulator built in C++, used to study macro-evolution of large populations on a large timescale.

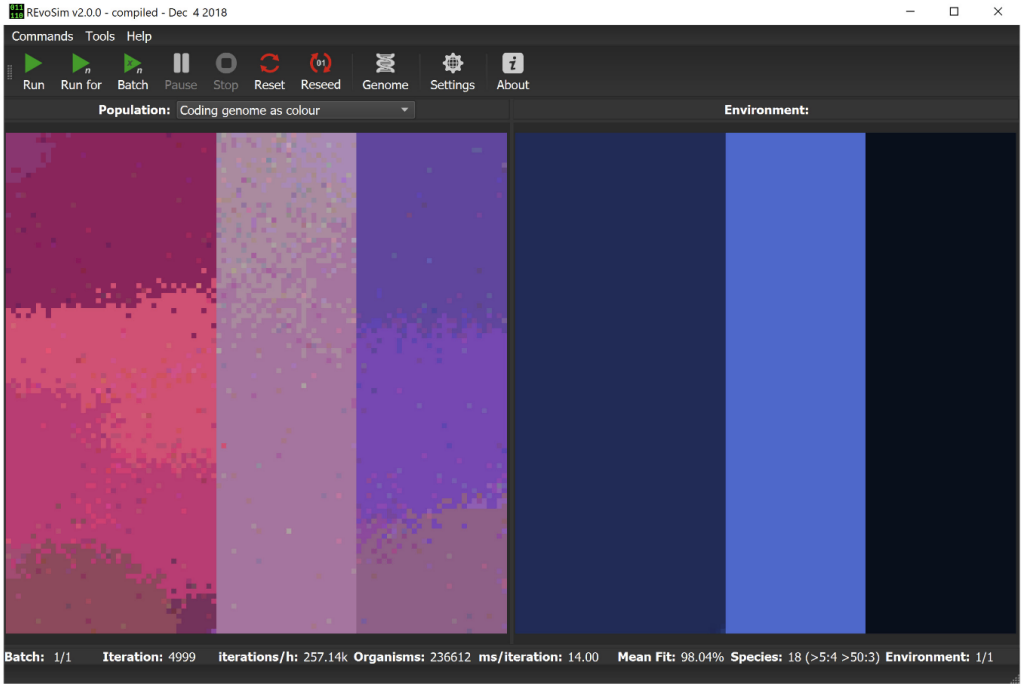


Figure 2.9: Screenshot of REvoSim simulator

The simulator features organisms which have an age, an energy system and a fitness level. The age of an organism controls how long it will be present on the map, while the energy of an organism controls whether the organism can and will breed. The fitness level of an organism determines the rate at which it gains energy.

Organisms in REvoSim have a 64-bit long genetic code which is used to calculate its appearance and its fitness level. They are not able to move between squares. Rather, movement is simulated when organisms reproduce a new offspring that settles in a neighbouring square. When organisms reproduce, genetic algorithms are used to mutate the offspring's genetic code, based on several complex rules. Organisms cannot exhibit complex behaviours, since they do not sense and act according to the environment.

As REvoSim was built to investigate macro-evolution of large populations, it does not offer many features for organisms to exhibit complex behaviour. Rather, the team that built it focused on computational efficiency, which resulted in a simulator with simple rules that can run at very high simulation speeds.

2.4.2 Physics-based Simulators

In contrast to Cellular Automaton model based simulators, physics-based simulators tend to have organisms that are not constrained to discrete grid-like environments. Organisms can move freely in the world in any direction. However, most core ideas are the same. They typically have the ability to move autonomously, reproduce and create offspring using genetic algorithms and evolve over time.

Canvas Artificial Life Evolutionary Simulation (33)

This artificial life simulator is a simple simulator that can be found online at https://cs.stanford.edu/~karpathy/canvas/evolve.html.

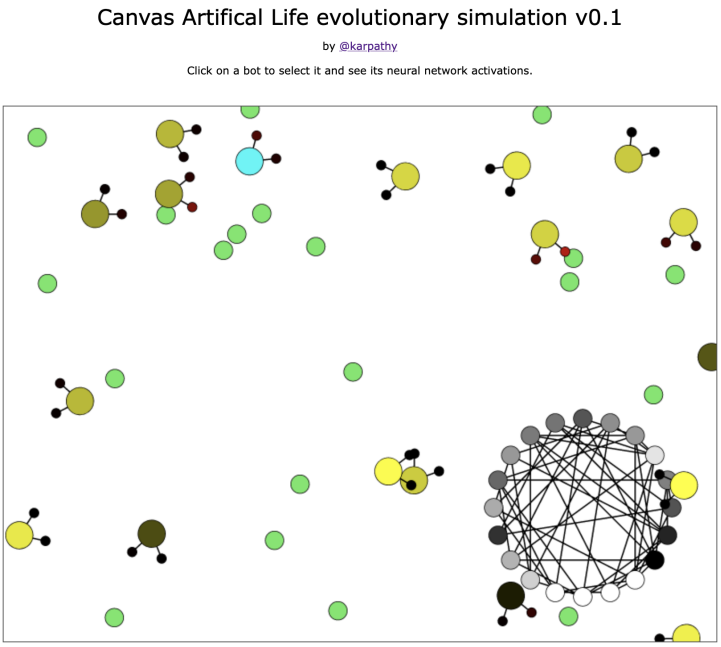


Figure 2.10: Screenshot of the Canvas Artificial Life Evolutionary Simulation

Organisms are represented as yellow dots, and food is represented as green dots. Food spawns periodically and randomly across the environment. Each organism's behaviour is controlled by a neural network which acts as its brain. It also has two eyes which can sense food in its vicinity - they act as inputs to its brain. If an organism eats enough food particles, it will reproduce with a mutation in its neural network weights.

After running the simulation for a long period of time, the organisms eventually learn to move towards food particles instead of spinning around in circles.

This simulator, while simple, serves as a good starting point for this project. It has straightforward mechanics that are simple to understand and a complex way of representing an organism's behaviour. Furthermore, it is available online, which means it is easily accessible by all.

Artificial Life Environment (ALIEN) Program (9)

ALIEN is a desktop-based simulation tool that features complex physics simulations built on the CUDA framework, a parallel computing framework that uses Nvidia Graphics Processing Units (GPUs).

Organisms are made up of multiple cells, each with their own specialisation. For example, there are cells responsible for computation and can make decisions for the organism based on a stored memory. There are also muscle cells, sensor cells and digestion cells, each with their own purpose. Some cells can even be programmed with an assembly-like language to produce extremely complex behaviours.

Users can also create custom organisms through a well developed editor, as seen in Figure 2.12.

Figure 2.11: Screenshot of the Artificial Life Environment (ALIEN)

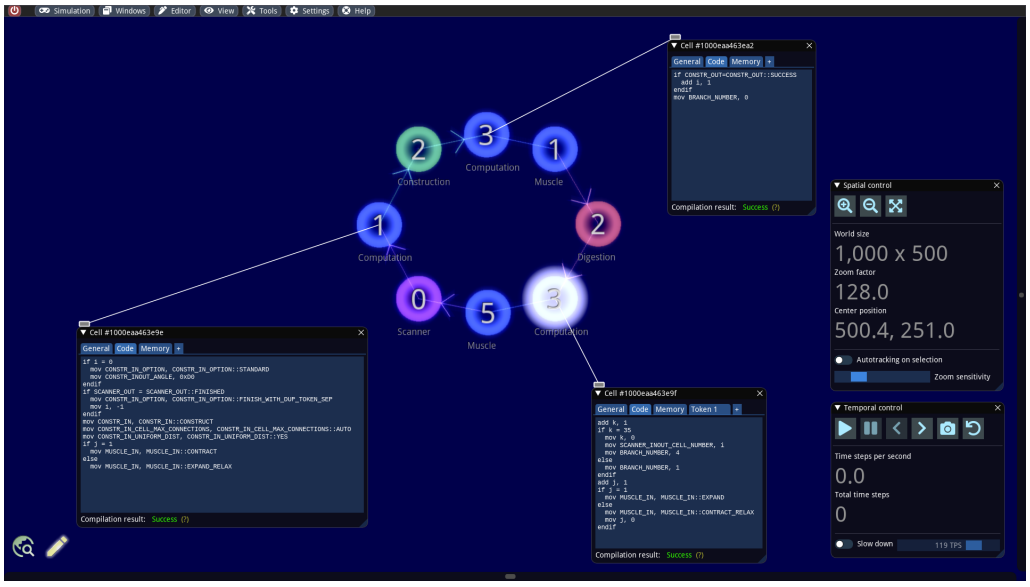


Figure 2.12: Screenshot of the Artificial Life Environment (ALIEN) editor

Overall, ALIEN is a complex simulator that has many interesting features. It would be interesting to attempt to incorporate some way for the user to edit organisms manually in this project, to see how well they can design an organism to survive in the simulator.

However, like EvoSim and REvoSim, this is a desktop-based simulator which limits its accessibility.

Chapter 3

An Artificial Life Simulator

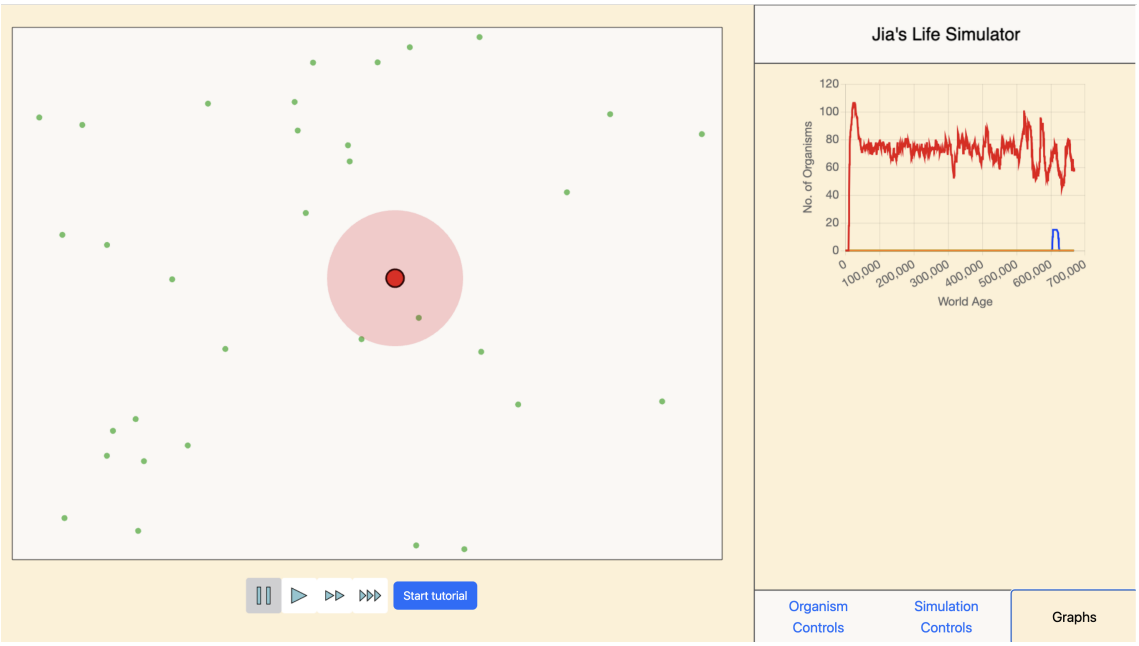


Figure 3.1: Screenshot of the whole simulator

3.1 Simulation Mechanics

The simulator is built using Phaser 3, a game framework for creating games in Javascript / Typescript. It features a built-in game engine that can handle object movement, collision detection and more, which is extremely useful when building a simulator.

The simulator runs in a continuous loop, with an update function called every few milliseconds. During this time, many things are happening simultaneously. Game objects can query the environment to find out about the world around it, then act on these inputs. They are also re-rendered graphically on the user's screen, based on their current position and velocity. Scenes that control the creation of new game objects and track their numbers are also updated. This complex loop involving multiple objects is handled by the Phaser framework.

3.1.1 Environment

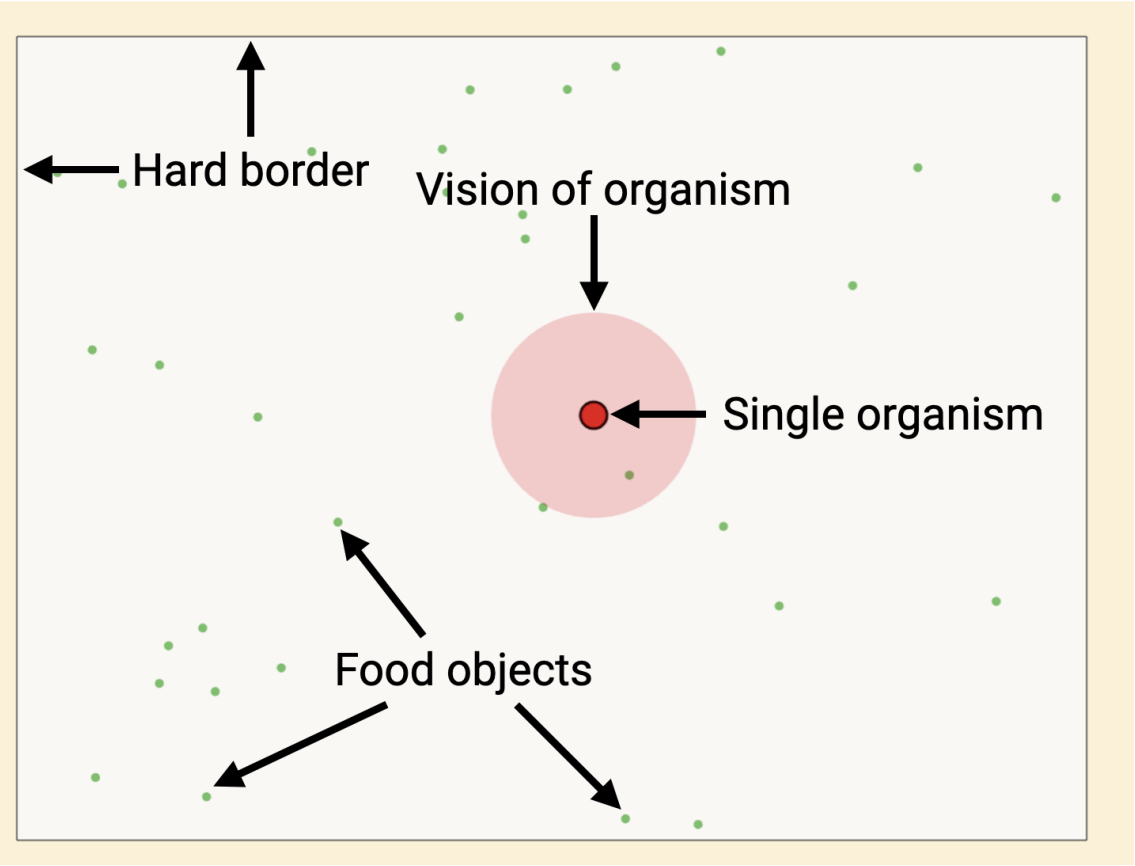


Figure 3.2: Annotated screenshot of the simulator environment

The simulator consists of a rectangular world with hard world borders (i.e. organisms will not wrap-around from the right edge to the left edge). A hard border was chosen over a warping border, since that more realistically models the world and makes the simulator slightly less confusing.

Food, in the form of plants, is represented by small green dots as seen in Figure 3.2. At set intervals, a new piece of food is spawned randomly within the world borders, if the food limit has not been hit. All food objects are equal and provide the same amount of energy to organisms, and are always the same size. The maximum number of food objects, the frequency at which it spawns and the amount of energy gained per food object can all be controlled by the user to investigate different dynamics.

To incentivise organisms to stay in the middle and away from the edges, food does not spawn within a set distance from the edges.

3.1.2 Organisms

Organisms are circular objects that can sense their environment, move around and reproduce. They are characterised by their physical features and their brain.

Physical attributes

Each organism has its own colour, size, speed and vision radius, which is determined

at birth and does not change over their lifetime. There is no relationship between an organism's physical attribute - an organism can both be big and fast at the same time. Every organism can 'see' around it in a circle, and it can discern between food and other organisms if it wishes to. The maximum distance they can see is determined by their vision radius.

There are hard limits (both a maximum and a minimum) to the size, speed and vision radius of organisms for a variety of reasons. First, there are practical limitations of the simulator. There is a physical limit to the size of the simulator, and having organisms that are too large and take up too much space in the world, which is unrealistic. Organisms that are too fast can also break the simulator by teleporting across the world if the update rate is too low.

Second, organisms cannot be too small due to the simulation mechanics. Smaller organisms need less energy to reproduce. When organisms are too small, they will reproduce too quickly and crash the simulator. An example of this can be seen in Figure 3.3.

As such, these hard limits are important to ensure the simulator works properly.

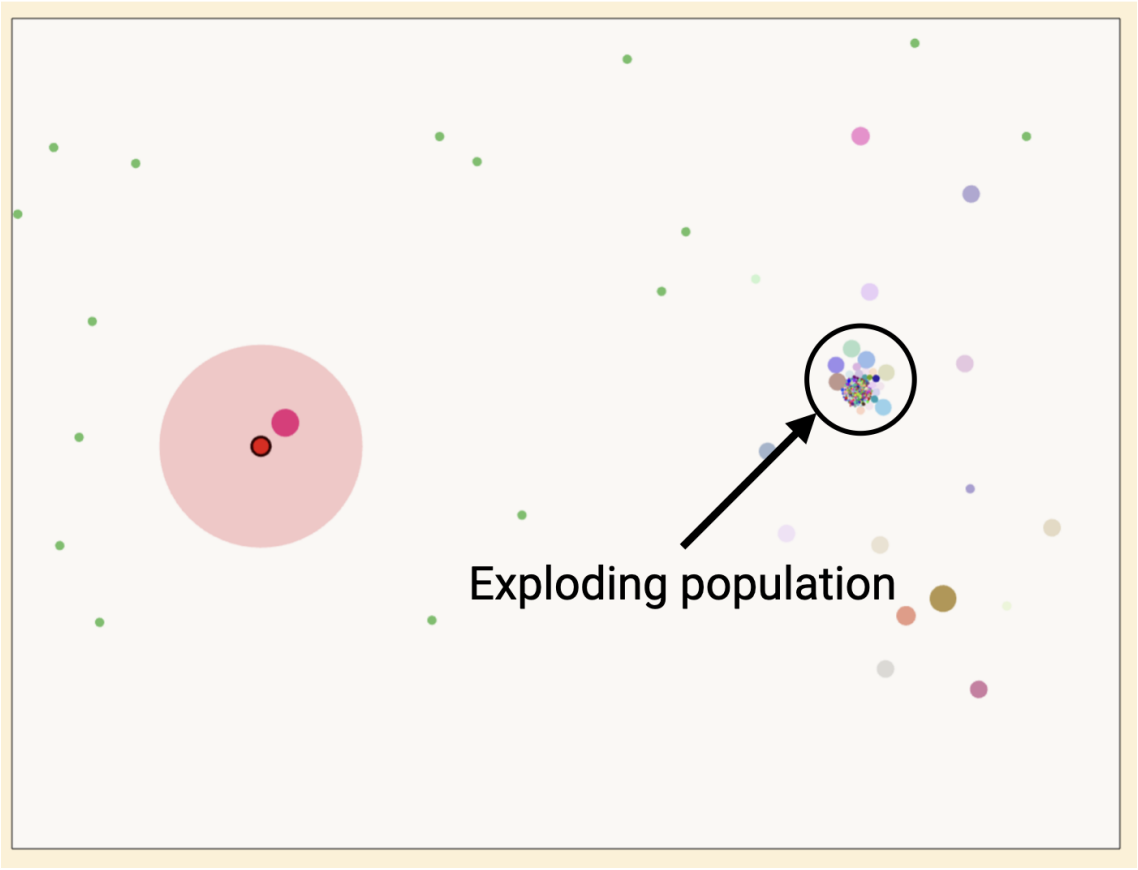


Figure 3.3: Exploding populations when organisms are too small (Simulator froze when screenshot was taken)

Brain types

Organisms can have different brain types as well. Their brains determine the actions they take in the environment and plays a key role in determining if they survive. The simulator currently supports three brain types - random, vision and neural network. Organism brains types are discussed further in Section 3.3 - **Organism Types**.

Energy system

An energy system governs the life and death of organisms. When created, organisms spawn with a starting energy of 100, although this can be customised by the user. Over time, organisms lose energy based on two factors - their basal energy loss, and their energy loss due to movement. The user can also control the rate of energy loss of all organisms using a slider.

$$\begin{aligned} \text{basalEnergy} &= 0.005 * \text{size}^{0.75} + \text{vision} * 0.00025 \\ \text{movementEnergy} &= (\Delta x + \Delta y) * 0.01 \\ \text{totalEnergyLoss} &= (\text{basalEnergy} + \text{movementEnergy}) * \text{userScale} \end{aligned} \tag{3.1}$$

An organism's basal energy loss is dependent on their size as well as vision radius. The larger an organism, the higher their metabolism, and thus the higher the rate of loss of energy. In this respect, the simulator scales the energy loss based on Kleiber's Law (34), a real life observation of metabolic rate to body mass. Kleiber's Law is defined in Equation 3.2.

$$\text{metabolism} \propto (\text{body mass})^{\frac{3}{4}} \tag{3.2}$$

When organisms move, they lose energy scaled to the distance they have travelled. This does not necessarily penalise faster moving organisms, as they can choose to move slower than their maximum speed or not move to conserve energy. In theory, this should allow organisms to develop more complex behaviour instead of always moving towards the nearest food.

The exact equations used to control an organisms energy loss is defined in Equation 3.1. Scaling values were chosen empirically to ensure that organisms do not die out too quickly while ensuring that the simulator is not overloaded with too many organisms.

When organisms gain enough energy, they will undergo asexual reproduction and lose half of their energy to their offspring. The larger an organism, the more energy it needs to reproduce. This is done to disadvantage larger organisms, to balance out the fact that they can get to food more easily as compared to smaller organisms.

Reproduction

When organisms reproduce, an almost-identical copy of their physical attributes and brains are copied into their offspring. Depending on the mutation rate (which is set by the user), any of their physical attributes and brains can mutate. For example, if the mutation rate is 0.2, it means that there is a 20% chance that the offspring has a slightly different size / speed / vision radius as compared to their parent.

As reproduction is only asexual and not sexual (i.e. there is only one parent), only inversion mutation is applied to the genes passed from parent to offspring. This introduces some diversity into organisms while ensuring that good genes continue to survive across generations. To ensure that organisms do not reproduce too quickly after gaining sufficient energy, they lose half of their energy to their offspring during reproduction. This mechanism also ensures that offspring receive sufficient energy to survive and explore the world, instead of dying out quickly.

Age

Every organism has an age, which increases the longer they survive in the simulator. Older organisms have the exact same physical attributes and brain types as when they were created (e.g. they do not move slower as they grow older). Once organisms reach a certain age, they will die instantly.

This mechanism means that organisms that have evolved well will eventually give way to younger organisms, preventing them from continually dominating the simulator.

3.2 User Interface

3.2.1 Tutorial

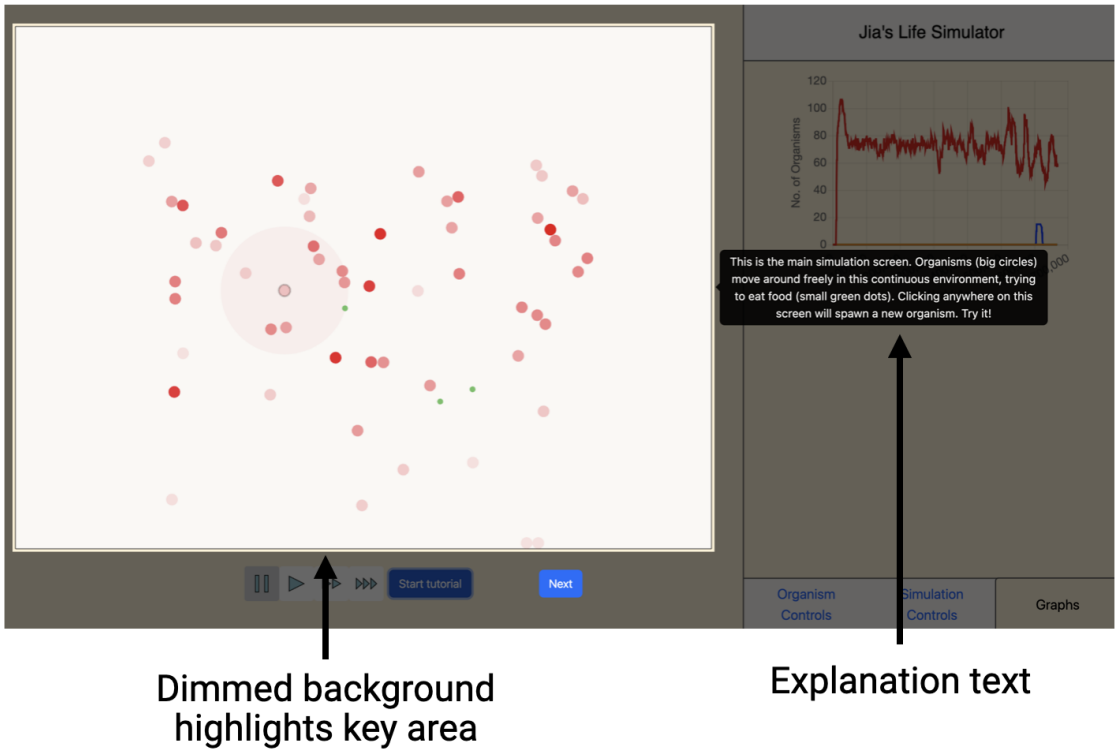


Figure 3.4: A single step of the tutorial

The simulator features a tutorial to walk users through its features and explain simulator mechanics. It is comprised of multiple steps, with each step explaining a part of the simulator, as seen in Figure 3.4. Each step highlights an area on the simulator and has a corresponding explanation text guiding the user. During the tutorial, the simulator continues to run and the UI elements are responsive during the tutorial. This enables the tutorial to let users try the simulator while a help text explains what they are doing, making for a more cohesive experience.

Users can use the arrow keys to navigate the tutorial instead of using the buttons if they prefer to do so. There is also a way to escape the tutorial by pressing the escape key, if they start the tutorial by accident.

3.2.2 Organism controls

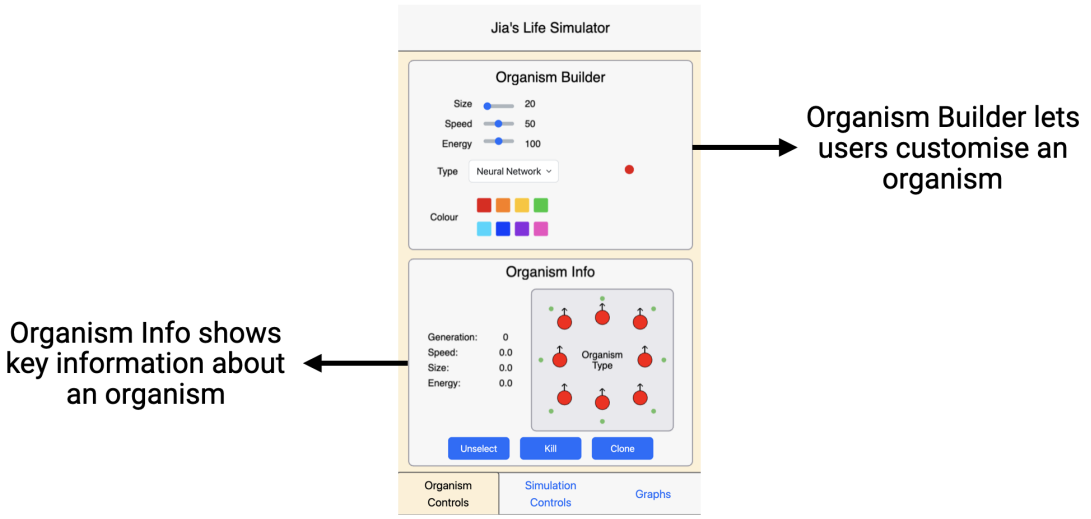


Figure 3.5: Organisms panel of the user interface

The organisms panel is split into two different sections, the **Organism Builder** and the **Organism Information** cards.

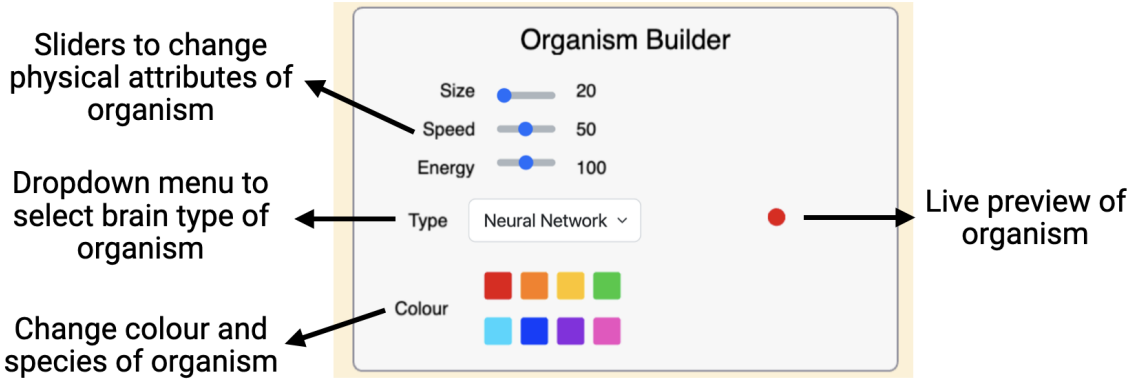


Figure 3.6: The organism builder interface

The **Organism Builder** allows users to build their own organism, by varying physical attributes like size, speed and starting energy of the organism. A live preview of the organism changes size and colour when the user varies those parameters, allowing them to instantly see the effect of their actions. Offering users a visual update reaffirms that their inputs are being used, thereby improving the user experience.

To create organisms, users have to click on the main canvas to spawn in an organism with the same physical and brain attributes as the current organism in the organism builder. To create multiple organisms, users can shift-click on the main canvas to create five organisms at once. These features are taught to users in the tutorial.

The **Organism Information** card allows users to easily view information about an existing organism. To select an organism, users have to click on the organism in

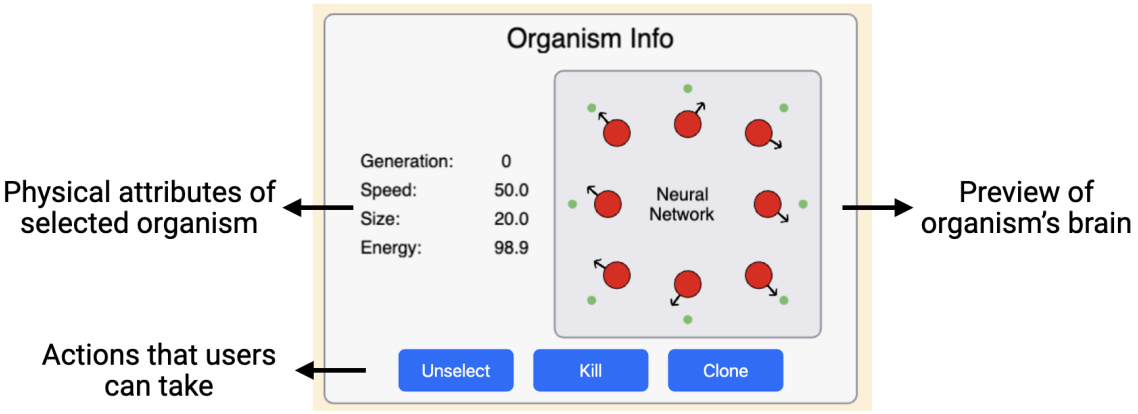


Figure 3.7: The organism information interface

the main canvas. The left hand side of the card holds information about the physical attributes of the organism, and is constantly updated if the value changes. For example, the energy value reflects the current energy of the organism and decreases over time.

On the right, a diagram describes an organism's brain and what it will do when food is nearby. The text in the center reflects the organism's brain type (i.e. Neural Network or Vision or Random). In each compass direction, the diagram shows how the organism will move if it encounters a food object in that direction. For example, the 12 o'clock red circle has an arrow pointing to the top-right. This means that if the organism sees that the nearest food to it is directly on top of it, it will decide to move towards the top-right. This visualisation allows the user to better understand what the neural network organism is thinking. For vision organisms, the arrows will always point towards the nearest food. For random organisms, there are no arrows as they just move randomly and do not consider where the nearest food is.

The bottom has three buttons that reflect certain actions a user can take. They can either unselect an organism, kill an organism or clone an organism. These buttons were developed in response to users requesting for them.

3.2.3 Simulator controls

Controls

The simulator controls panel allows users to change simulator-wide configurations, such as the mutation rate of chromosomes and the rate of loss of energy. These controls affect the simulator rather drastically, allowing the user to investigate how different conditions affect the way organisms evolve. For example, organisms that thrive in a world with high energy loss rate but high food spawn rate may not be so well adapted to survive in a world with low energy loss rate and a low food spawn rate. Each slider has a carefully selected default value to ensure organisms do not die out too quickly, nor do they overpopulate the world too quickly.

The mutation rate affects the probability at which a single gene in any chromosome is randomly mutated when an organism reproduces. The default mutation rate is set to be low at 0.01, as this makes it more likely for good genes to be passed down from parents to offspring, while ensuring genetic diversity.

The energy loss rate affects the rate at which organisms lose energy, as defined in

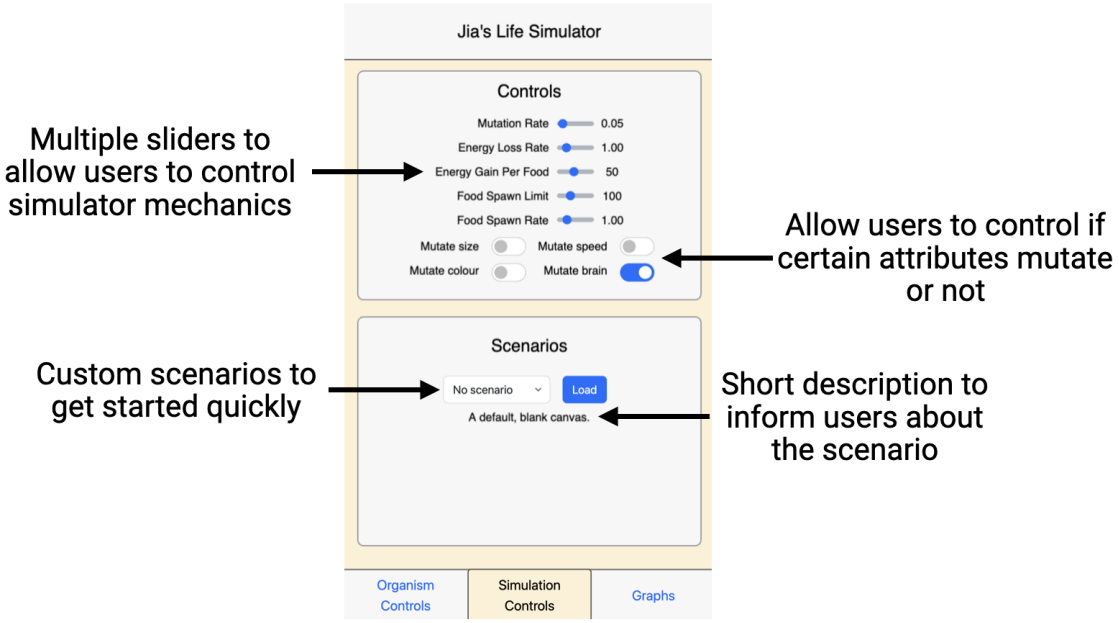


Figure 3.8: Simulator panel of the user interface

Equation 3.1. Large values lead to only the very best organisms surviving, although it often causes the population to die out quickly.

The energy gain per food control affects the amount of energy gained by a single organism when it eats food. Larger values typically lead to overpopulation.

The food spawn limit and food spawn rates affect how food spawns in the environment, by limiting the maximum number of food objects and the rate at which it spawns.

Furthermore, users have the option to control if certain attributes like size, speed and colour mutate or not. This is useful if users only want to investigate the evolution of a single attribute.

Scenarios

Scenarios are pre-made worlds with settings that encourage certain behaviours to evolve. This allows users to quickly get into the simulator and observe evolution without having to change the many settings available. A short description of each scenario is provided to inform the user of what should ideally happen in the simulator.

For example, the 'Battle of Brains' scenario loads three different groups of organisms, all with different brains. This should in theory highlight which organism type is the most optimal at surviving.

3.2.4 Graphs

The graph panel features a graph that updates regularly to reflect the current population in the simulator. Built using Chart.js, a popular charting library, the graph offers a way for users to observe how the population changes over time. For instance, users can increase the energy loss rate and see how different groups of organisms react.

The graph consists of eight separate line plots, each corresponding to a single

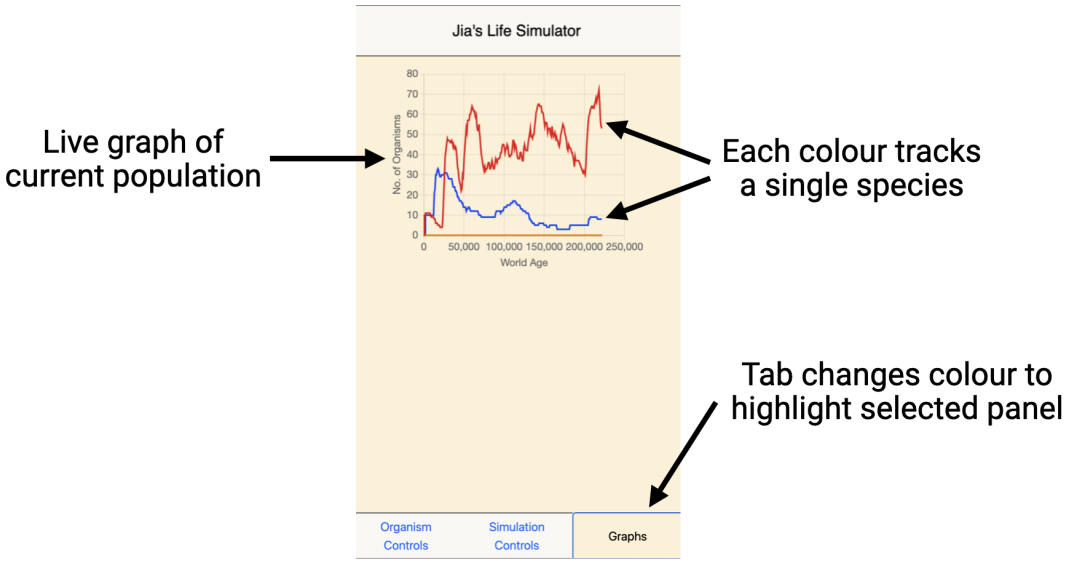


Figure 3.9: Graph panel of the user interface

species based on the colour selected in the organism controls panel. When an organism reproduces, it will increase the count of its species on the graph.

3.3 Organism Types

Every organism has a brain type, and this determines their actions in the simulator. Organisms decide what to do every few milliseconds, when their update function is called by the game engine. This allows them to make quick decisions and react quickly to the environment around them.

3.3.1 Random Organisms

Random organisms were the first organism type to be developed, as they are very simple. They move with a fixed x-velocity and y-velocity for a predetermined amount of time, before changing to a new, random set of velocities.

It is important to decide how long they move for before changing directions. If the chosen value is too large, they will easily get stuck at edges, since they will be moving forward for a long time. If the chosen value is too small, it is more likely that they will be stuck in a local area for a long period of time, which means they do not explore the world sufficiently.

Due to the random nature of their movement, they are ideal for investigating the importance of physical attributes on an organism's survival in the simulator. These organisms cannot evolve their brains, as they are essentially moving randomly regardless of the environment around them. As such, only their physical attributes like their size and speed will determine how well they survive in the world.

3.3.2 Vision Organisms

Vision organisms make use of their vision to find the nearest food object, and moves in a direct line at their maximum speed towards the food. When there is no food in their vision, they behave like random organisms and wander around the environment.

These organisms are an example of the optimal selfish organism, as they move towards food and ignore everything else. They prioritise their own survival before others and will push against other organisms in their way.

Due to their superior decision making, vision organisms typically thrive in the simulator and out-compete most organisms with other brain types.

3.3.3 Neural Network Organisms

Neural network organisms have a neural network to represent their brains. They use their vision to sense the nearest food object and the nearest organism near them. These values, along with their x/y position in the world and their current energy, are passed as inputs to their neural network.

Their neural network has two output values, their x velocity and y velocity. Since the *tanh* function is used by default in neural networks, these values are then multiplied by their maximum speed and used to move the organism. Note that the outputs of a *tanh* function is $[-1, 1]$, which means that organisms can move in both the positive and negative direction.

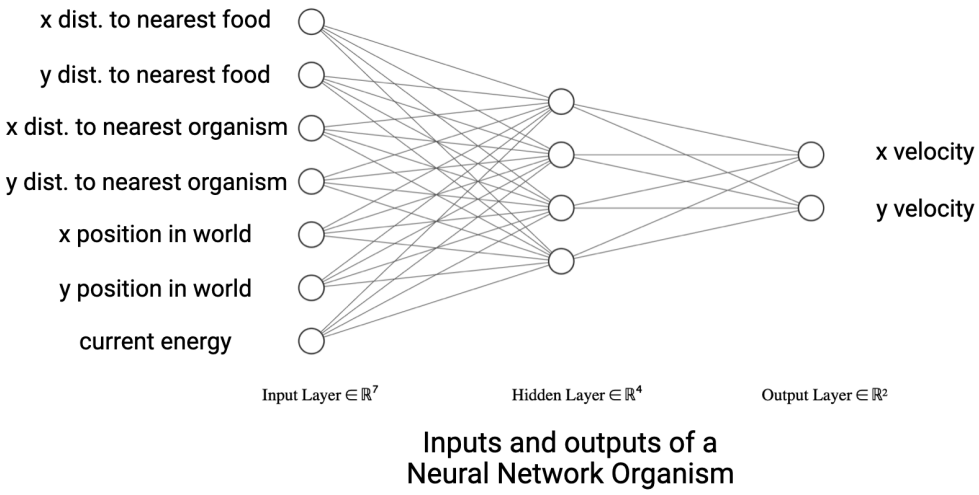


Figure 3.10: Diagram of Neural Network Organisms' brains [1]

The neural network weights in these organisms are initialised randomly using the Glorot initialisation scheme (22), while the biases were initialised to 0. No backpropagation of loss is carried out. Rather, network weights and biases are randomly picked to be changed based on the mutation rate of the simulator. If picked, their values are drawn from a uniform distribution in the range $[-0.5, 0.5]$. Biases are scaled to be one-tenth of their weight counterparts if mutated.

The neural network faces a huge difficulty in learning a complex objective function that is dependent on multiple features like the energy cost it takes to move towards food, the amount of energy require to reproduce, etc. For example, if there

[1] Created with NN-SVG (35)

is a food object nearby but another organism is nearby, the organism has to decide if it is optimal to simply ignore it and continue searching for other food objects. These are hard questions that the neural network has to figure out in order to survive in the simulator. To help the neural network better model the 'optimal' action to take, a few measures have been taken.

First, the food spawn rate, food spawn limit and the rate of energy loss has been carefully tuned in the default settings to best allow neural network organisms to evolve properly. If there is little food and/or the rate of energy loss is too high, these organisms will die out quickly and be unable to reproduce. Evolution takes at least a few generations to properly take effect, so it is essential that organisms are able to reproduce in their lifetimes before dying. If there is too much food available, then organisms with very poor decision making can also survive, simply by moving randomly and running into food objects. This stifles effective evolution, since almost all organisms can reproduce, not just the ones that are actually better at finding food. An abundance of food also often leads to overpopulation, which causes the simulator to perform poorly.

Second, the weight and bias initialisation as well as mutation values have been carefully selected. Initially, neural network organisms had a tendency to move to the edges of the environment, ignoring food that was within its visual radius. It was thought that there was an issue with the neural network implementation, hence the preview of the organism's brain was created to better understand it was thinking. A few experiments revealed that the organisms were always going in a single direction, regardless of where food was around it. This was due to large bias values massively outweighing the effects of the inputs. As such, when biases are mutated, their values are intentionally scaled down to one-tenth of the weights. This value was empirically chosen to reduce the effect of organisms moving to the edges, while ensuring that biases still had an impact on the overall neural network.

Third, the mutation rate in this simulator is set to 0.01. Mutation rates are key parameters in genetic algorithms and are essential to effective evolution of organisms. As discussed by Vié et al., large mutation rates means that good genomes exist for a short period of time only, while small mutation rates can stifle diversity (19). While mutation rates in nature are estimated to be in the order of 10^{-8} (19), these rates are too low for an artificial life simulator. In the context of an artificial life simulator, low mutation rates can also mean that a longer period of time is needed to observe emergent behaviour as a result of evolution. As such, the value of 0.01 was selected empirically to ensure that given the maximum speed of simulation, some evidence of evolution can be observed on a small time scale.

3.4 Software Design

The simulator can largely be separated into a few key packages, namely the genetics library, the neural network library, the organisms library and the scenes library.

3.4.1 Genetics

The genetics library is a largely standalone part of the simulator that is inspired by a genetic algorithm project on Github (36).

Chromosomes are a core concept in the library and are based on the Template Method Pattern. They are generic abstract classes that accept a type parameter `T`, which represents a basic unit of genetic material. Each concrete chromosome class must implement several abstract methods, which are used when chromosomes are undergo mutation or crossover.

```
export abstract class Chromosome<T> {
  /* Genetic makeup of a chromosome, its key information */
  private genes: T[];

  /* Returns a random gene */
  public abstract getRandomGene(): T;
  /* Express this chromosome as an attribute of an organism, e.g. size */
  public abstract toPhenotype(): any;
  /* Construct an instance of this chromosome from a given attribute */
  public abstract fromPhenotype(phenotype: any): Chromosome<T>;
  /* Returns a clone of this chromosome */
  protected abstract getCopy(): Chromosome<T>;
```

Source Code 2: Snippet of the Chromosome class

By structuring chromosomes like this, mutation functions can be written that work on all types of chromosomes, as they can just access the relevant functions to work. This avoids code duplication, as each chromosome does not need its own mutation function.

Before this library was written, mutation was handled in a haphazard way where a mutation function could only work on chromosomes with a specific type. It was also unwieldy to use as it involved converting a phenotype to a certain format before it could be passed to the mutation function.

3.4.2 Neural Network

Initially, Tensorflow was the solution of choice based on the discussion in Section 2.3.2. However, it caused the simulator to freeze and crash after a period of time, potentially due to the need to manually release memory used by tensors in the GPU.

The simulator only required a fully connected neural network to be able to calculate a forward pass, set parameters and get parameters. Since it did not requite the complex functionality offered by Tensorflow and even the other libraries, a decision was made to create a custom neural network. This allowed the neural network to be simple and decently performant without adding an additional library, which would decrease page loading times. In addition, the neural network used by the simulator is small and each forward pass is calculated separately (one for each organism), so the benefits of using the GPU would have been negligible.

A drawback to this approach was the additional development time needed to create and test the custom neural network library. The lack of optimisation in the custom neural network as compared to the other libraries was another drawback to the approach.

Despite changing libraries midway through the project, the use of a `Network Interface` was extremely helpful in reducing the impact of the refactor, as seen

Source Code 3. This is an example of how the adapter pattern can be useful in software engineering. When parts of the simulator need a neural network, they use the `Network Interface` type signature instead of the actual implementation. This also allows the use of other neural network libraries in the future if so desired.

```
export interface Network {
  /* Performs a forward pass through the neural network */
  forward(values: number[]): number[];

  /* Performs backpropogation */
  backprop(loss: number[]): void;
  updateWeights(): void;

  /* Getters and setters for the internal matrices */
  setParams(params: LinearModelParams): void;
  getParams(): LinearModelParams;
}
```

Source Code 3: Network Interface [2]

3.4.3 Organisms

The template method pattern (37) is again employed to implement organisms. `Organism` is an abstract class that represents all organism types. It is responsible for most functionalities of organisms, for instance creating the physics body, creating the visible body, handling energy calculations and so on. There were then some abstract hook methods for subclasses to implement.

For each new brain type, a new concrete class was create to inherit from the abstract `Organism` class. For example, the `VisionOrganism` inherits from the `Organism` class, and implements methods like `onDestroy()` and `onUpdate()`. In the `onUpdate()` function, it looks for the nearest food and moves towards it. This consolidates duplicate code into the abstract class, making it easier to edit common functionality in a single place.

3.4.4 User Interface

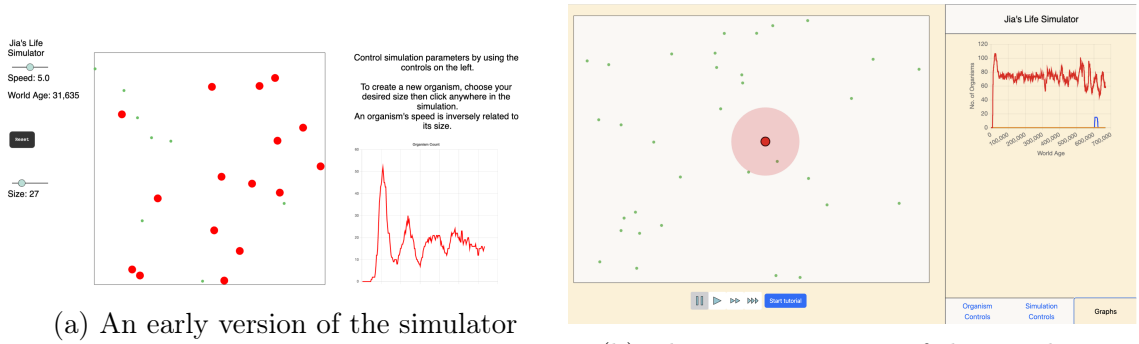
The user interface library is built to reduce code duplication by consolidating the customisation of UI elements into a single place. For example, the `BootstrapFactory` handles the creation of buttons, sliders and dropdown menus using the external Bootstrap library. It also handles the styling of these elements. To create a button element, this class is called and a button that has a consistent look is returned. By providing consistency in the UI elements, the user's experience is improved.

[2]The back-propagation methods were written to support a reinforcement learning organism, but they are not used in the final simulator

Chapter 4

Evaluation

4.1 User Interface



(a) An early version of the simulator

(b) The current version of the simulator

Figure 4.1: Progression of the simulator design

The simulator was iteratively built over a few months, taking into account users' feedback. Many features, such as the tutorial, visualisation of the neural network and keyboard shortcuts were user requested features that were implemented as a result of their feedback.

For example, early on in the simulator, users were tasked to use the simulator and try to figure out its purpose and features without any help. Many of them had to ask what the red and green dots represent, which means their purposes was unintuitive. As such, the tutorial was developed to explain the simulation mechanics as well as how to use the interface.

4.2 Webpage Performance

4.2.1 Methodology

The performance of a website is critical to its success, as a slowing loading website negatively impacts the user's experience significantly. For example, the BBC noted that 10 percent of users stopped using the site for every extra second their page took to load (38). In order to profile websites effectively, Google has developed Google Lighthouse, an open source tool that measures multiple metrics such as First Contentful Paint and Largest Contentful Paint (39). Based on the raw results

of these metrics, Lighthouse first calculates a metric score for each one, based on a distribution derived from real world data from the HTTP Archive. Then, an overall performance score is calculated using a weighted average of metric scores (39). A summary of metrics considered can be found in Table 4.1.

Metric	Description
First Contentful Paint (FCP)	Time until first DOM element is rendered
Speed Index (SI)	How quickly visible content is loaded
Total Blocking Time (TBT)	Time between FCP and TTI
Largest Contentful Paint (LCP)	Time at which largest text or image is rendered
Cumulative Layout Shift (CLS)	Movement of visible elements on page
Time To Interactive (TTI)	Time until page is fully responsive

Table 4.1: Google Lighthouse Performance Metrics (1) (2)

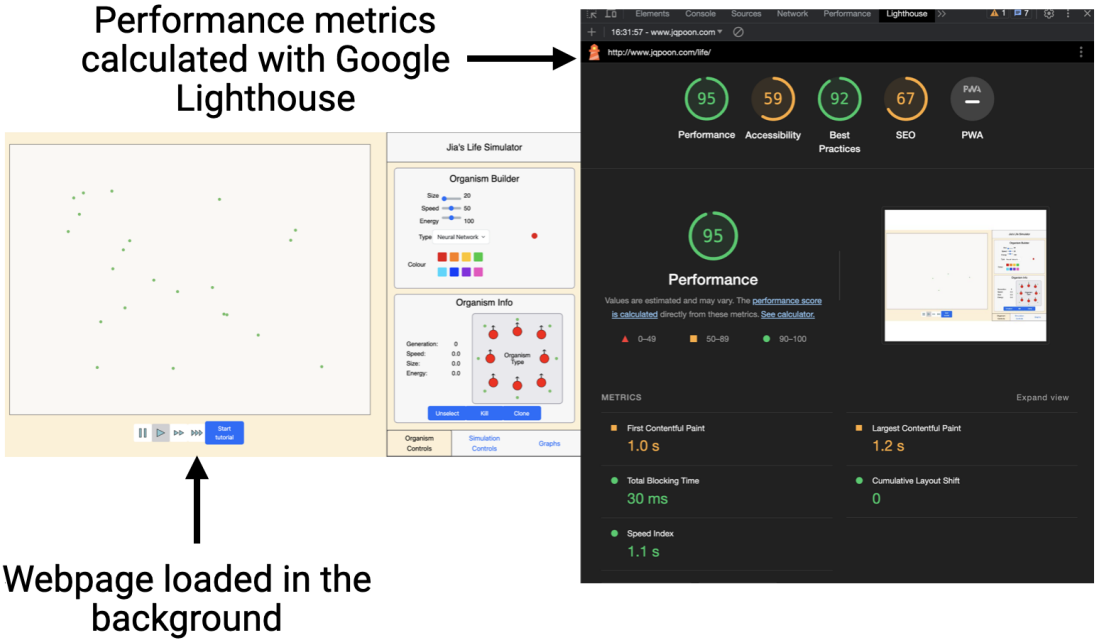


Figure 4.2: Example of Google Lighthouse performance metrics

These performance metrics provide insights into the different ways in which developers can improve their website's user experience. However, these time-based measurements can differ from run to run, depending factors such as the website's load and a user's network speed. As such, studies by Heričko et al. (40) have shown that using the median of five consecutive performance tests greatly reduces variability. To properly evaluate the performance of the simulator, Google Lighthouse was ran five times and the median of those times were evaluated. The results are summarised in Table 4.2 [1], while the full results are provided in Appendix A.1.

4.2.2 Results and Analysis

[1]Detailed metrics with better precision can be viewed by clicking on the 'See Calculator' button
[2]This was calculated using the Lighthouse Calculator, after the medians of the other metrics have been calculated.

Metric	Median of five runs	Metric Score
Overall Performance Score [2]	-	95
First Contentful Paint	1046ms	84
Speed Index	1046ms	95
Total Blocking Time	32ms	100
Largest Contentful Paint	1243ms	89
Cumulative Layout Shift	0	100

Table 4.2: Webpage Performance Results

Based on the metric scores, the overall performance of the website is rated as 'Good' (39). The worst performing metric is First Contentful Paint, with a metric score of 84 corresponding to a 'Needs Improvement' rating.

On top of providing performance scores, Google Lighthouse also details opportunities and diagnostics - practical advice on how to improve performance. Each advice is explained in detail as well which is valuable information for developers.

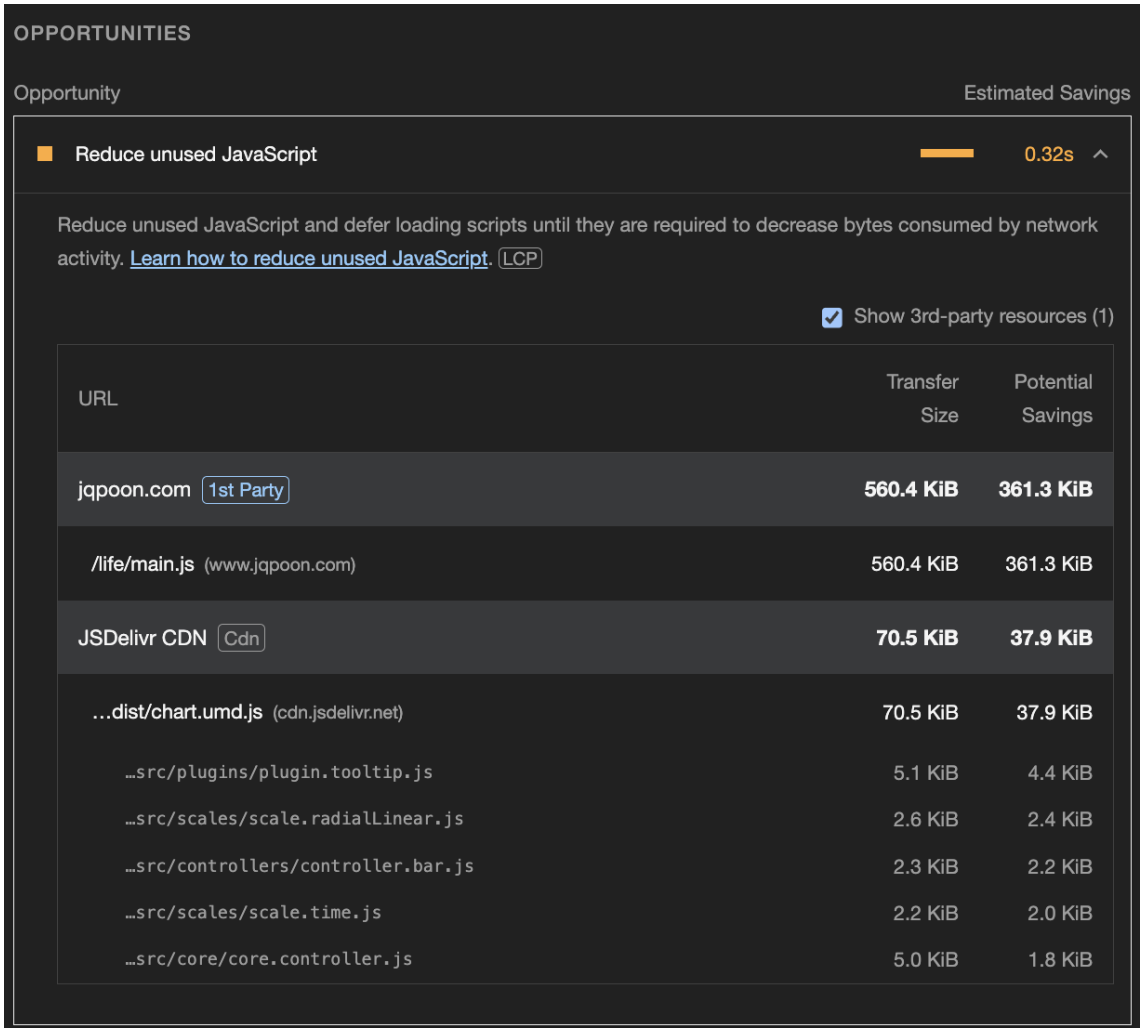


Figure 4.3: Google Lighthouse analysis of the website

For the webpage, Google Lighthouse recommends to 'Reduce unused Javascript' as an opportunity to reduce up to 0.32s of load time. The large size of the `main.js`

file could be attributed to the fact that it has to contain the Phaser 3 library, the Rex Plugins library as well as the Bootstrap library. All of these are essential to the simulator, and thus cannot be removed.

To further reduce the size of the `main.js` file, a technique known as tree-shaking can be used. By only importing functions that are actually used in the code, the size of libraries can be reduced, leading to faster loading times (41). However, the author of Phaser, Davey, has stated that tree-shaking will not help reduce file sizes (42). Another potential option would be to manually create a custom bundle based on Phaser, but that would require substantial time and effort that can be better spent on developing the simulator itself.

Overall, the simulator webpage loads quickly at around 1 second with an overall performance rating of 95/100 according to Google Lighthouse. There is room for improvement although it will require substantial effort to implement.

4.3 Simulator Performance

4.3.1 Methodology

Having a webpage that loads fast is only half of the story. Importantly, the simulator itself must be performant and well featured as well. To quantify the simulator's performance, the simulator will be pushed to its limits by spawning many organisms, and its frames per second (FPS) will be recorded using Google DevTools. The maximum number of organisms in a test is defined as the number of organisms that the simulator can support while still maintaining the same frames per second as the baseline. The same test will be repeated for different organism types as well as simulator speeds. A baseline was first run with zero organisms and at the base speed (3x) to determine the base FPS of the simulator. Figure 4.4 depicts an example of how this test was carried out.

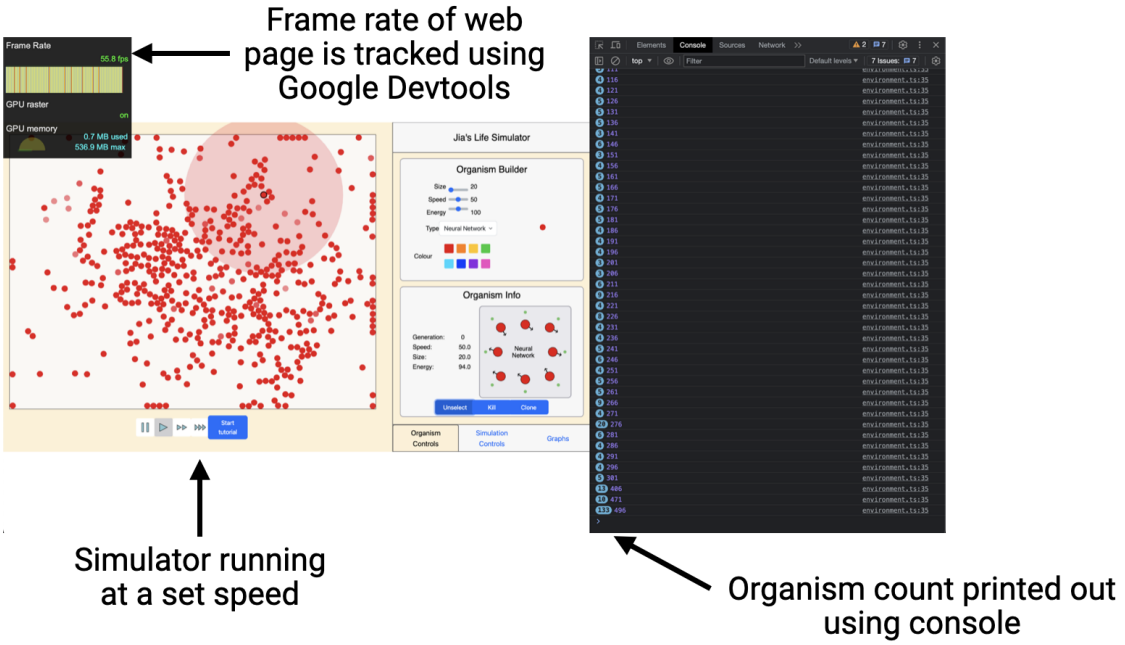


Figure 4.4: Stress testing the simulator using only neural network organisms

4.3.2 Results and Analysis

Organism Type	Simulator Speed	Max Number of organisms
None (Baseline)	3x	0 (59.4 fps)
Random	3x	500
	10x	340
	30x	205
Vision	3x	515
	10x	360
	30x	210
Neural Network	3x	500
	10x	300
	30x	200

Table 4.3: Simulator Stress Test Results

Overall, the results suggest that the faster the simulator speed, the lower the maximum number of organisms supported, as expected. Across all organism types, the maximum number of organisms is roughly 200 at 30x speed, 300 at 10x speed and 500 at 3x speed.

The simulator running at 3x achieved a performance approximately similar to a Phaser 3 collision test with 500 organisms all colliding into each other (26), suggesting that the game engine is the limitation. In a separate Phaser 3 test with object-object collision turned off (43), the game engine ran smoothly with up to 40 000 objects. This suggests that turning off object-object collision can be a way to massively improve the performance of the simulator. However, this would lead to an unrealistic environment as organisms would be able to pass through one another.

Unsurprisingly, spawning neural network organisms led to a decreased performance as compared to random and vision organisms. This is expected, as each neural network organism has to calculate a forward pass at every update step, which involves matrix multiplication. However, the decrease in performance is not extremely significant, with an estimated 10-20% decrease in performance as compared to the others across all simulator speeds.

A rather interesting observation is that vision organisms actually performed marginally better than random organisms across all simulator speeds. This is surprising, given that vision organisms have to calculate if there is any food within its vision radius, and then calculate the direction to move in. One possible explanation for this is that generating random numbers in Javascript could involve more mathematical operations than calculating distances between two points. However, this seems unlikely and further investigation and/or testing is needed.

4.3.3 Further Investigation

To further investigate the cause of the decreased performance at high organism counts, the Chrome DevTool Performance panel is used to profile the simulator. Runtime performance was recorded while organism were continually added in until a visible drop in FPS was observed. The results were then reviewed in Chrome DevTools.

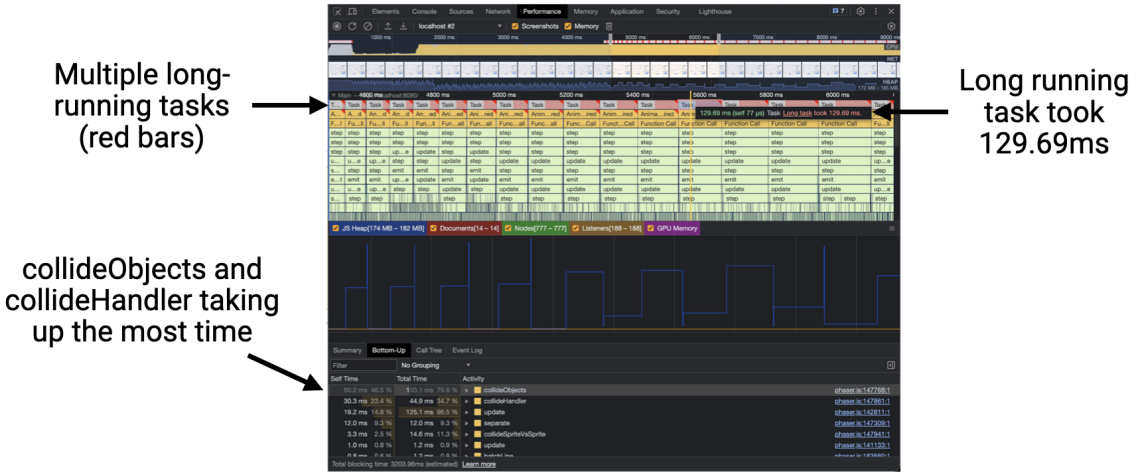


Figure 4.5: Performance analysis of lagging simulator

Based on the report generated by Chrome DevTools, the main function calls taking up the most time to execute were the `collideObjects` and `collideHandler` functions, taking up 46.5% and 23.4% of the time respectively in a single call to `step` (`step` is the Phaser's internal version of `update`). This aligns closely with what was observed earlier, when the game engine performed significantly worse when object-object interactions were turned on. As such, the game engine seems to be the limiting factor in the performance of the simulator.

To improve performance, some measures have been taken. The physics engine of the game has been configured to run at 15fps. This value was chosen empirically based on the trade off between a visibly lagging game (too low fps) versus poor performance (too high fps).

The maximum simulation speed of the simulator was chosen to be 30x based on empirical tests as well. Low simulation speeds led to smoother simulations that could support more organisms, but had the disadvantage of organisms taking too long to evolve.

However, when the simulator is run for long periods of time, its performance drops and it can support fewer organisms before a substantial decrease in framerate. More testing is needed to investigate the reason behind this.

4.4 Comparison to Existing Simulators

4.4.1 Performance

In order to compare the performance of simulators, the following three metrics will be considered: maximum simulation speed of simulator, number of concurrent organisms and type of simulator (cellular automaton vs physics based). By calculating the difference between UTC times between updates, the simulator built was observed to run the update function once every 17ms, or roughly 60 times per second at the fastest simulation speed. As seen in the previous section, the life simulator can support up to 200 organisms at the fastest simulation speed.

EvoSim 2.8 is a desktop-based application built in Java that is based on the cellular automaton model. Based on a few trial runs, it can run its simulation at

around 2.5 steps per second, and has only one speed control. There does not seem to be a maximum number of organisms this simulator can support based on its user interface and report write-up. An estimate of roughly 100-200 organisms was determined based on the size of the maps available, as well as the graphs. Compared to the simulator built, EvoSim runs at a much slower speed of 2.5 steps per second and can support roughly the same number of organisms.

REvoSim is a desktop-based application as well and was designed to study macro-evolution of large population on a geologic timescale. As such, it can support populations of 2.5×10^5 organisms and can run at between 1×10^5 to 1×10^6 iterations per hour (8). This is substantially faster and more powerful than the simulator built. However, it must be emphasised that REvoSim was built for performance and is desktop-based. This means that it can better utilise the CPU to run its simulation as compared to a single tab.

Canvas Artificial Life Evolutionary Simulation is a simple web-based simulator that has no speed controls. It is not obvious the rate at which the simulator runs. However, interesting behaviours take a long time to emerge when using the simulator. Based on the source code, it supports around 10-20 organisms at once. Compared to the simulator built, Canvas is much simpler and does not perform as well.

ALIEN is a desktop-based simulator that is built on the CUDA framework. Based on a few test runs of using the simulator, it can simulate up to 85 steps per second, and supports hundreds of organisms each with multiple body parts. Like REvoSim, ALIEN is vastly more powerful than the simulator built, largely due to the fact that it is desktop-based.

Overall, the simulator built is decently performant compared to the other existing simulators, especially considering that it is running purely on a web browser.

4.4.2 Features

Unlike performance, it is harder to compare features between simulators quantitatively as they do not have a one-to-one correspondence. However, an attempt will be made to compare the overall complexity of simulators by looking at the complexity of the simulation mechanics as well as their user friendliness.

The simulator built features a simple environment in which food spawns randomly, and organisms can be spawned to survive in the environment. These organisms have different physical attributes and brain types, and undergo slight mutations to these attributes during reproduction. There are many ways for the user to interact with the simulation - through organism design, organism information panel, custom scenarios, simulation controls and through informative graphs.

EvoSim features a complex environment that has different biomes for different parts of the world. Each biome has its own features which adds an additional layer of complexity to the simulator. EvoSim has organisms which can not only move around the world but can take different actions as well, depending on the environment around them. In terms of user interface, EvoSim has a simple interface that allows users to build their own world, view statistics and observe organisms. However, it does not allow for the customisation of organisms. It also does not feature more complex organisms like the neural network organism. On the whole, EvoSim has implemented a much better environment, but lacks in customisability.

REvoSim does not feature an environment where organisms can freely explore and move around. Rather, they can only move around when reproducing to create a new offspring. Organisms in REvoSim are simple as well, as they do not react to their environment in the traditional sense. Its energy levels and fitness are directly calculated from its genetic code, and that decides what actions it can take. It has a simple user interface designed around running/stopping the simulator. Compared to the simulator built, REvoSim is much less complex both in terms of simulation mechanics as well as features available to users.

Canvas Artificial Life Evolutionary Simulation has a very basic environment similar to the simulator built, with food spawning randomly around the world. Its organisms have neural network brains, that evolve and mutate over time just like the simulator built. However, it lacks a user interface. Users can only click on organisms to view their neural network, but nothing else. Overall, Canvas is a very simple simulator both in terms of simulation mechanics and its user interface. The simulator built is definitely more well featured than Canvas.

ALIEN has a highly complex organism system, where each organism is made up of multiple cells. These cells have their own functions like movement cells, digestion cells and even programmable cells. It also has a well developed user interface that allows users to design these cells from scratch, amongst other features. Compared to the simulator built, ALIEN is much more complex and well featured.

On the whole, the simulator does not feature an especially complex simulation environment, although it does so sufficiently. The user interface is decently featured compared to some other simulators, although it could certainly be improved.

4.5 Research Questions

On top of developing a browser-based simulator, this project also aims to investigate a key research question.

4.5.1 Multi-agent Collaborative Behaviour

A key question is to investigate if multi-agent collaborative behaviour emerge naturally using genetic algorithms. In order to answer this question, the 'Neuro Evolution' scenario will be used. This scenario features 50 neural network organisms with a size of 20, a speed of 50, a starting energy of 100, a vision of 500, all in the center of the simulator. The simulator was then run for 10 000 iterations (i.e. until the world age was 10 000) and the surviving organisms were investigated.

To start, a simpler neural network organism was considered. These organisms were only given four inputs, the relative x-coordinate of the nearest food ,the relative y-coordinate of the nearest food and their x-y position in the world. This would serve as a baseline to compare the more complex neural network organism against.

As seen from Figure 4.6, most of the neural network organisms had learnt to move towards food, although not perfectly. They were moving in mostly the right direction to get to the food. The sharp spike in population at the start can be attributed to organism with a good initial network immediately finding food and reproducing. However, the organisms with a poor network soon died, causing the sharp drop in population. Towards the end after 5000 iterations, the average population was slightly higher at approximately 45 organisms, as compared to below 40 before 5000

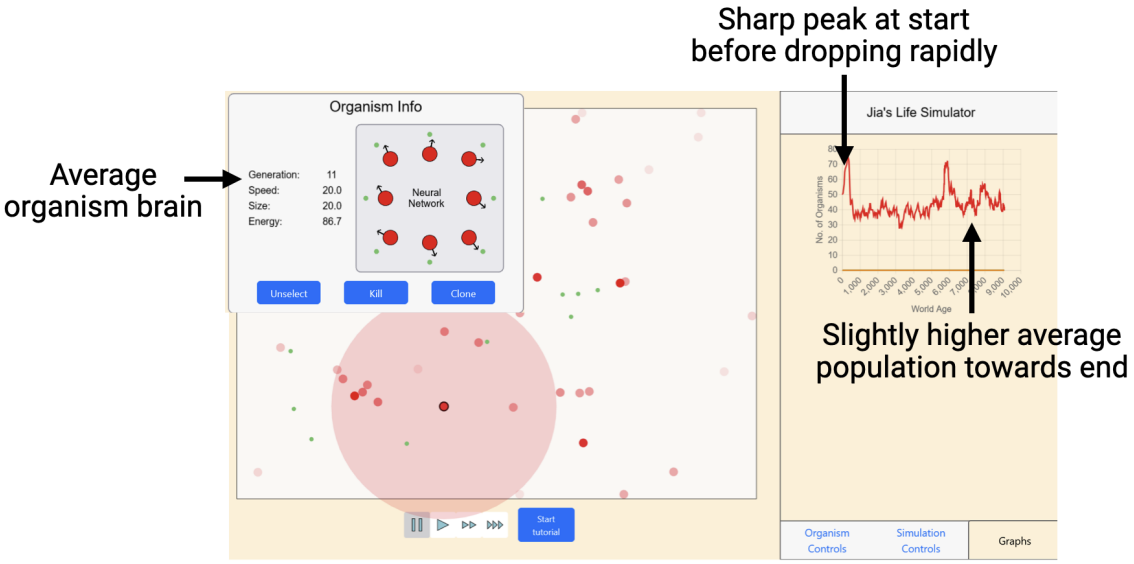


Figure 4.6: Graph of simpler NN organism count after 10000 iterations

iterations. This shows that organism are indeed evolving over time to become more efficient at finding food.

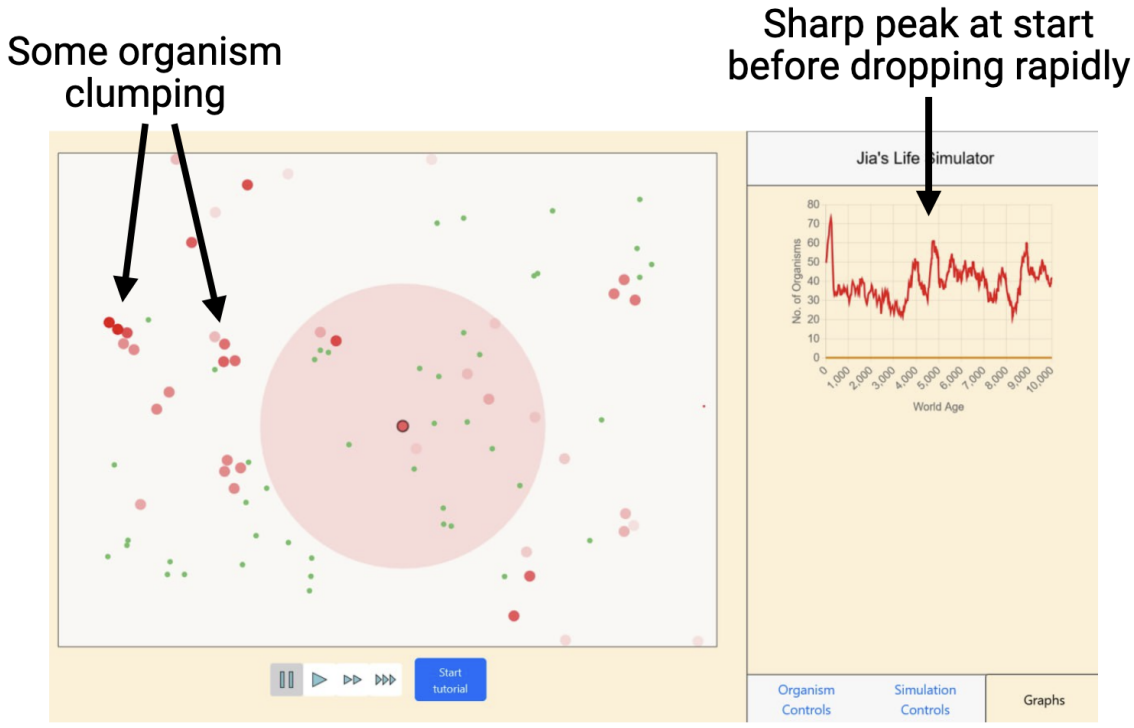


Figure 4.7: Graph of NN organism count after 10000 iterations

When evolving the more complex neural network, a similar trend was observed at the start. Like the previous experiment, the organisms did seem to become more proficient at surviving, with roughly 40 organisms surviving after iteration 5000. Visible 'clumps' of organisms were observed moving around during the simulation, although not all organisms were observed to have this behaviour. This may not be evidence of organisms moving together to cooperate, but rather them simply moving

towards the nearest food at the same speed.

It is disappointing that more complex behaviours did not emerge after many generations, contrary to expectations. One reason could be because the simulator conditions were not set up properly to encourage the evolution of such a behaviour. Conditions like the mutation rate could be too high or too low, although a range of values were tried and none of them worked.

A different environmental setup was tested to see if it was more conducive for effective evolution. Using the same 'Neuro Evolution' scenario, the energy loss rate was doubled but the food spawn rate was double as well. This would ideally reward organisms that worked together to find food efficiently, and severely punish the ones that did not.

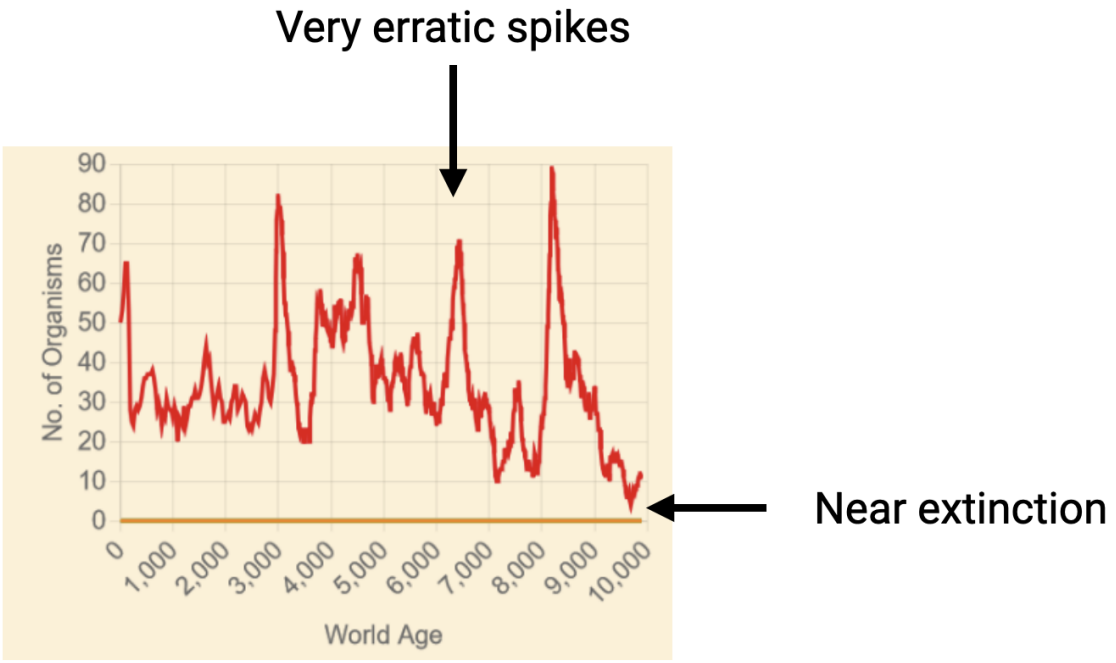


Figure 4.8: Graph of NN organism count after 10000 iterations, different environmental conditions

As seen in Figure 4.8, this led to an erratic population size. Organisms would die quickly if they did not manage to find food, causing steep decreases in population. This led to more food accumulating where there were no organisms. Thereafter, a lucky organism could end up 'finding' this stash of food and reproduce rapidly, causing the population to grow quickly. No complex behaviours between organisms was observed.

Chapter 5

Conclusion

Overall, we have successfully implemented a well featured, browser-based artificial life simulator that is capable of supporting up to 200 organisms simultaneously while the simulation is run at 30x speed. The simulator features complex mechanics with an intuitive user interface that allows users to explore and change different parameters easily. The webpage loads quickly in 1 second, and has a Google Lighthouse performance metric score of 95 corresponding to a 'Good' rating. Despite being a browser-based simulator, its performance is on par with other desktop-based simulators.

A weakness of this project is that in terms of research, it has not managed to conclusively answer if multi-agent collaborative behaviour can emerge using genetic algorithms. Instead, evidence has shown that organism learn to be better at finding food, although they do not seem to cooperate extensively.

5.1 Further Work

There are many directions to take this project in.

Developers can choose to explore more complex organisms, for example by implementing an organism that learns through reinforcement learning throughout its lifetime. Then, when it reproduces, it can pass down its tuned weights to its offspring. These new organisms can then be compared to the existing ones to see how much more efficient they are.

More complex environments can be explored as well. The idea of biomes would be interesting to investigate, whereby some organism are more suited to survive in certain climates as compared to others. This could be expanded to having different speeds, vision and actions to take in different biomes.

Lastly, more complex organism-organism and organism-environment interaction can be explored. Instead of just moving towards or away from other organisms, organisms can potentially have different actions to interact with other organisms and its environment.

Appendix A

Details of Performance Tests

Run Number	FCP (ms)	SI (ms)	LCP (ms)	TBT (ms)	CLS
1	1046	1105	1329	42	0
2	1067	1129	1272	18	0
3	1021	1083	1243	40	0
4	1066	1118	1264	17	0
5	1023	1098	1240	32	0

Table A.1: Detailed Google Lighthouse Results

Bibliography

[1] Google. First contentful paint, Sep 2019. URL https://developer.chrome.com/docs/lighthouse/performance/first-contentful-paint/.

[2] Jasper van Riet, Flavia Paganelli, and Ivano Malavolta. From 6.2 to 0.15 seconds – an industrial case study on mobile web performance. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 746–755, 2020. doi: 10.1109/ICSME46990.2020.00084.

[3] Christopher G. Langton, Charles Taylor, J. Doyne Farmer, and Steen Rasmussen. *Artificial Life II*, pages xiv–xv. Avalon Publishing, 2003.

[4] Hod Lipson and Jordan B. Pollack. Automatic design and manufacture of robotic lifeforms. *Nature*, 406(6799):974–978, 2000. doi: https://doi.org/10.1038/35023115.

[5] Enrico G. Campari, Giuseppe Levi, and Vittorio Maniezzo. Cellular automata and roundabout traffic simulation. In Peter M. A. Sloot, Bastien Chopard, and Alfons G. Hoekstra, editors, *Cellular Automata*, pages 202–210, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-30479-1.

[6] Kyung-Joong Kim and Sung-Bae Cho. A Comprehensive Overview of the Applications of Artificial Life. *Artificial Life*, 12(1):153–182, 2006. ISSN 1064-5462. doi: 10.1162/106454606775186455. URL https://doi.org/10.1162/106454606775186455.

[7] Yigit Uyan. Evosim: Genetic evolution simulation, 2017.

[8] Russell J. Garwood, Alan R. T. Spencer, and Mark D. Sutton. Revosim: Organism-level simulation of macro and microevolution. *Palaeontology*, 62(3):339–355, 2019. doi: https://doi.org/10.1111/pala.12420.

[9] Christian Heinemann. What is alien?, 2022. URL https://alien-project.gitbook.io/docs/.

[10] Richard Davey. Phaser 3 docs, 2017. URL https://github.com/photonstorm/phaser3-docs/blob/master/LICENSE.

[11] Rex. Phaser 3 rex notes, 2018. URL https://github.com/rexrainbow/phaser3-rex-notes/blob/master/LICENSE.

[12] JSFoundation. Webpack license, 2016. URL https://webpack.js.org/license/.

[13] Meta Platforms Inc., Feb 2023. URL https://github.com/jestjs/jest/blob/main/LICENSE.

[14] Mark Otto. Bootstrap license faqs, May 2022. URL https://getbootstrap.com/docs/5.2/about/license/.

[15] Chart.js. Chart.js license, 2022. URL https://github.com/chartjs/Chart.js/blob/master/LICENSE.md.

[16] M. St Laurent Andrew. *Understanding Open Source and Free Software Licensing*, pages 14–15. O'Reilly Media, Inc., 2004.

[17] Pradnya A. Vikhar. Evolutionary algorithms: A critical review and its future prospects. In *2016 International Conference on Global Trends in Signal Processing, Information Computing and Communication (ICGTSPICC)*, pages 261–265, 2016. doi: 10.1109/ICGTSPICC.2016.7955308.

[18] Sourabh Katoch, Sumit Singh Chauhan, and Vijay Kumar. A review on genetic algorithm: past, present, and future. In *Multimedia Tools and Applications*, volume 80, pages 8091–8126, 2021. doi: 10.1007/s11042-020-10139-6.

[19] Aymeric Vié, Alissa M. Kleinnijenhuis, and Doyne J. Farmer. Qualities, challenges and future of genetic algorithms: a literature review, 2020. URL https://arxiv.org/abs/2011.05277.

[20] Seyedali Mirjalili. *Evolutionary Algorithms and Neural Networks*. Springer Cham, 1st edition, 2018. ISBN 978-3-319-93025-1.

[21] Michael A. Nielsen. *Neural networks and deep learning*. Determination Press, 2015.

[22] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR. URL https://proceedings.mlr.press/v9/glorot10a.html.

[23] Qianqian Ye and Lauren Lee McCarthy. p5.js, 2023. URL https://github.com/processing/p5.js/.

[24] Rob Morris and Shukant Pal. Pixijs basics - what pixijs is, 2023. URL https://pixijs.io/guides/basics/what-pixijs-is.html.

[25] MDN Contributors. Webgl: 2d and 3d graphics for the web, May 2023. URL https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API.

[26] Richard Davey. Phaser 3 500 colliding bodies, 2023. URL https://github.com/photonstorm/phaser3-examples/blob/master/public/src/physics/arcade/500collidingbodies.js.

[27] Richard Davey. Phaser 3 arcade physics, 2021. URL https://photonstorm.github.io/phaser3-docs/Phaser.Physics.Arcade.ArcadePhysics.html.

[28] BrainJS Community. Brain.js, April 2023. URL https://github.com/BrainJS/brain.js#examples.

[29] Juan Cazala. Synaptic, 2018. URL https://github.com/cazala/synaptic.

[30] Yun Ma, Dongwei Xiang, Shuyu Zheng, Deyu Tian, and Xuanzhe Liu. Moving deep learning into web browser: How far can we go? *The World Wide Web Conference*, page 1234–1244, May 2019. doi: 10.1145/3308558.3313639.

[31] Tensorflow. Platform and environment, 2022. URL https://www.tensorflow.org/js/guide/platform_environment#memory_management.

[32] Andrew Adamatzky. *Game of Life Cellular Automata*. Springer Publishing Company, Incorporated, 1st edition, 2010. ISBN 1849962162.

[33] Andrej Karpathy. Canvas artificial life evolutionary simulation v0.1, 2012. URL https://cs.stanford.edu/~karpathy/canvas/evolve.html.

[34] Albert Thommen, Steffen Werner, Olga Frank, Jenny Philipp, Oskar Knittelfelder, Yihui Quek, Karim Fahmy, Andrej Shevchenko, Benjamin M Friedrich, Frank Jülicher, and Jochen C Rink. Body size-dependent energy storage causes kleiber's law scaling of the metabolic rate in planarians. *eLife*, 8:e38187, jan 2019. ISSN 2050-084X. doi: 10.7554/eLife.38187. URL https://doi.org/10.7554/eLife.38187.

[35] Alexander LeNail. Nn-svg: Publication-ready neural network architecture schematics. *Journal of Open Source Software*, 4(33):747, 2019. doi: 10.21105/joss.00747. URL https://doi.org/10.21105/joss.00747.

[36] Nathan Soufflet. Darwin. https://github.com/nathsou/Darwin/tree/master, 2021.

[37] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Behavioral Patterns*, page 325–325. Addison-Wesley Professional, 1994. ISBN 9780321700698.

[38] Matthew Clark. How the bbc builds websites that scale. *Creative Bloq*, Jan 2018. URL https://www.creativebloq.com/features/how-the-bbc-builds-websites-that-scale.

[39] Google. Lighthouse performance scoring, Sep 2019. URL https://developer.chrome.com/docs/lighthouse/performance/performance-scoring/.

[40] Tjaša Heričko, Boštjan Šumak, and Saša Brdnik. Towards representative web performance measurements with google lighthouse. *Proceedings of the 2021 7th Student Computer Science Research Conference (StuCoSReC)*, page 39–42, 2021. doi: 10.18690/978-961-286-516-0.9.

[41] Simon Huang, May 2019. URL https://www.diva-portal.org/smash/get/diva2:1318692/FULLTEXT01.pdf.

[42] Richard Davey. Tree shaking for webpack projects, Mar 2018. URL https://github.com/photonstorm/phaser/issues/3351.

[43] Richard Davey. Phaser 3 40000 world bodies, 2023. URL https://github.com/photonstorm/phaser3-examples/blob/master/public/src/physics/arcade/40000worldbodies.js.