BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

EVOLUTION OF ANIMAL LIFE USING SIMULATION

EVOLUCE ZVÍŘECÍHO SVĚTA FORMOU SIMULACE

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **FILIP KURČÁK**
AUTOR PRÁCE

SUPERVISOR **Ing. MICHAL VLNAS**
VEDOUCÍ PRÁCE

BRNO 2025

Abstract

This thesis focuses on the evolution of simplified animals within the context of a changing world. The goal of this thesis is to create a simulation of animal populations and monitor their genetical adaptations to their environment. Focus is also on the changes that accompany shifts in the animal's environment, like changes in overall temperature, access to water and other environmental changes and disasters. The end result of this thesis is a 2D simulation in the Godot game engine, and analysis of the simulation's findings.

Abstrakt

Táto bakalárska práca sa zameriava na evolúciu zjednodušených živočíchov v kontexte meniaceho sa sveta. Cieľom tejto práce je vytvoriť simuláciu populácií živočíchov a sledovať ich genetické adaptácie na prostredie. Dôraz je tiež kladený na zmeny, ktoré sprevádzajú posuny v prostredí živočíchov, ako napríklad zmeny celkovej teploty, prístupu k vode a iné environmentálne zmeny a katastrofy. Výsledkom tejto bakalárskej práce je 2D simulácia v hernom engíne Godot a analýza zistení zo simulácie.

Keywords

evolution, evolutionary techniques, godot engine, simulation, natural selection

Kľúčové slová

evolúcia, evolučné techniky, godot engine, simulácia, prirodzený výber

Reference

KURČÁK, Filip. *Evolution of animal life using simulation.* Brno, 2025. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Michal Vlnas

Rozšírený abstrakt

Zvyšujúca sa frekvencia diskusií týkajúcich sa environmentálnych zmien, ako je globálne otepľovanie a ekologické narušenia, poukazuje na potrebu nástrojov na pochopenie a predpovedanie ich vplyvu na prírodné systémy. Simulácie ponúkajú silný prístup k preskúmaniu týchto zložitých dynamík, najmä pri dlhodobých procesoch, ako je genetická adaptácia, keďže umožňujú predvídať možné výsledky a proaktívne zvažovať reakcie. Táto práca sa zameriava na túto potrebu prostredníctvom návrhu a implementácie dvojrozmernej (2D) agentovo založenej simulácie ekosystému. Hlavným dosiahnutým cieľom bolo vytvorenie funkčnej platformy pomocou herného enginu Godot, ktorého základná logika simulácie bola rozšírená v jazyku Rust kvôli výkonu. Tento systém úspešne demonštroval, ako sa populácie virtuálnych zvierat vyvíjajú a adaptujú v priebehu času ako reakcia na simulované environmentálne tlaky a používateľom definované parametre, pričom schopnosti zberu dát umožnili analýzu populačných a genetických trendov.

Táto práca spadá do oblasti výpočtovej vedy, konkrétne využíva techniky z oblasti umelej života (ALife) a agentového modelovania. Využíva princípy inšpirované genetickými algoritmami (GAs) na modelovanie adaptácie a procedurálne šumové funkcie (PNFs) na generovanie rozmanitého prostredia, to všetko v rámci simulácie v reálnom čase.

Počítačové simulácie poskytujú všeobecnú metodológiu, ktorá umožňuje vytváranie modelu napodobňujúceho dynamický systém, ktorý sa vyvíja v diskrétnych krokoch od počiatočných podmienok, pričom sa sleduje vznikajúce správanie. Genetické algoritmy tvoria základ pre evolúciu agentov, pričom agenti majú jednoduchú štruktúru podobnú chromozómu (genotyp) a podliehajú operátorom selekcie, kríženia a mutácie. V tejto simulácii je fitnes emergentnou vlastnosťou, nie explicitne definovanou funkciou. Procedurálne šumové funkcie sa používajú na generovanie heterogénneho a prirodzene pôsobiaceho prostredia vytváraním počiatočných máp teploty a vlhkosti, ktoré následne určujú rozdelenie biomov a dostupnosť zdrojov. Výber enginu Godot ako platformy, rozšíreného o Rust pomocou GDExtension, bol urobený s cieľom vyvážiť rýchly vývoj a vizualizačné schopnosti (Godot) s výkonným výpočtom pre jadro simulačnej logiky (Rust).

Architektúra simulácie je postavená na jasnom oddelení zodpovedností, pričom jednotlivé „stavebné bloky“ definujú agentov, ich prostredie a pravidlá riadiace ich interakcie a evolúciu.

Návrh agentov: Zvieratá sú autonómni agenti, ktorých správanie je riadené hierarchiou potrieb (prežitie, reprodukcia, prieskum). Každý agent má `Genotyp` (súbor normalizovaných génov reprezentovaných reálnymi číslami). Tento genotyp určuje aj agentov `Fenotyp` a ovplyvňuje jeho interakcie a pravdepodobnosť prežitia.

Návrh prostredia: Svet je 2D šesťuholníková mriežka reprezentovaná v jazyku Rust. Každá dlaždica má statické vlastnosti odvodené z procedurálneho šumu počas inicializácie a dynamické vlastnosti, ktoré sa menia počas simulácie.

Inicializácia simulácie: Používatelia konfigurujú parametre (rozmery mapy, genetické faktory, environmentálne pravidlá) cez predvolené hodnoty, voliteľné načítanie CSV alebo grafické rozhranie v Godote. Godot vygeneruje počiatočné statické dáta mapy a odovzdá ich Rust backendu, ktorý potom inicializuje svoj vnútorný stav sveta. Počiatočná populácia zvierat sa následne vygeneruje v Ruste.

Evolučné mechanizmy: Fitnes je emergentný a je určený schopnosťou zvieraťa prežiť a reprodukovať sa. Selekcia prebieha prirodzene, keď menej adaptované jedince neuspokoja potreby alebo zomrú iným spôsobom. Kríženie spočíva v náhodnom výbere génov od rodičov a mutácia zavádza malé náhodné zmeny hodnôt génov.

Dynamika simulácie: Simulácia postupuje v diskrétnych kolách, riadených fázovou slučkou v Ruste. Zmena nastáva vďaka vnútorným dynamikám (cykly zdrojov, populačné zmeny, genetická evolúcia, pohyb) a môže byť ovplyvnená parametrami meniteľnými používateľom alebo konceptuálne navrhnutými environmentálnymi udalosťami.

Simulácia bola realizovaná hybridným prístupom. Herný engine Godot pomocou GDScriptu spravuje používateľské rozhranie, počiatočné generovanie máp, vizualizáciu biomov a počtu zvierat na dlaždiciach a celkovú kontrolu simulácie (spustenie/zastavenie, vstup parametrov, spúšťanie kôl, zobrazenie aktuálneho stavu, uloženie dát do CSV).

Jadrová logika simulácie, vrátane správania agentov, zmien environmentálneho stavu a evolučných procesov, je implementovaná v knižnici jazyka Rust, integrovanej cez GDExtension. Tento Rust backend obsahuje centrálnu štruktúru `Simulation`, ktorá uchováva stav simulácie (mapa sveta, zoznam zvierat, parametre, štatistiky).

Slučka simulácie v Ruste je štruktúrovaná do samostatných fáz: pasívne aktualizácie máp, rozhodovanie zvierat, riešenie interakcií (boje, párenie), vykonávanie akcií, aktualizácia zdrojov a veku a kontrola úmrtí, vykonanie pohybu a nakoniec aplikácia narodení a úmrtí.

Implementovaná simulačná platforma úspešne splnila svoje hlavné ciele. Hybridná architektúra Rust/Godot poskytla výrazný výkon pre hlavnú simulačnú slučku, čo umožnilo simuláciu rozsiahlych populácií zvierat v priebehu dlhších období, zatiaľ čo Godot poskytol flexibilné prostredie pre interakciu používateľa a vizualizáciu.

Výkon: Rust backend preukázal efektívne spracovanie kôl. Paralelizácia úloh aktualizácie máp pomocou Rayon pozitívne prispela k výkonu, najmä pri väčších mapách.

Populačná dynamika: Simulácie vykazovali uveriteľné ekologické vzory, vrátane počiatočného rastu populácie, stabilizácie okolo environmentálne určenej kapacity a výkyvov spôsobených dostupnosťou zdrojov a emergentným správaním agentov.

Evolučná adaptácia: Jasné adaptačné trendy v priemerných genetických znakoch boli pozorované v priebehu viacerých generácií ako reakcia na konkrétne environmentálne tlaky, čím sa potvrdil mechanizmus emergentného fitnes a selekcie. Populácie sa konzistentne konvergovali k adaptívnym genetickým profilom vhodným pre ich prostredie.

Aj keď model zjednodušuje mnohé komplexnosti reálneho sveta, slúži ako robustný nástroj na preskúmanie základných princípov výpočtovej evolúcie a agentovo založeného ekologického modelovania.

Táto práca predstavuje návrh a implementáciu 2D agentovo založenej simulácie ekosystému, ktorá úspešne demonštruje evolučnú adaptáciu prostredníctvom emergentného fitnes a selekcie v procedurálne generovanom svete. Hybridná architektúra využívajúca Godot pre frontend a Rust pre výkonnostne náročné jadro sa ukázala ako efektívna.

Evolution of animal life using simulation

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Ing. Michal Vlnas. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Filip Kurčák
May 13, 2025

Acknowledgements

V této sekci je možno uvést poděkování vedoucímu práce a těm, kteří poskytli odbornou pomoc (externí zadavatel, konzultant apod.).

Contents

Chapter 1

Introduction

In the modern era, discussions about changes in the natural environment, such as global warming and environmental disasters, are becoming more prevalent. It is now easier than ever to access information about endangered or extinct animal species, as well as examples of species adapting to these environmental changes. However, waiting for such changes before formulating theories and responses is often far from ideal. This is where simulations become invaluable, enabling us to predict potential future outcomes and work on implementing proactive measures before real-world changes fully manifest. Simulations are particularly useful when addressing long-term subtle shifts, such as those in climate patterns or the genetic adaptations of fauna and flora to these changes, which may not be immediately evident.

The primary goal of this thesis was achieved through the design and implementation of a two-dimensional (2D) ecosystem simulation using the Godot game engine, extended with Rust for performance. The resulting simulation successfully demonstrated how animal populations change over time under simulated environmental pressures and illustrated the dynamic nature of evolutionary processes. By allowing users to define initial world parameters, the system allowed the analysis of agent adaptation across various environments and the examination of impacts from runtime environmental changes. Data collected from each run are subsequently used to generate graphs that can provide insights into population dynamics and genetic trends.

The theoretical foundation of this work is discussed at length in Chapter 2. The design of the application is shown in Chapter 3, and finally the implementation is dissected in Chapter 4.

Chapter 2

Theoretical Foundations for Ecosystem Simulation

In this chapter, a detailed look at the theory that underlies the simulation of an ecosystem is provided. The focus is on understanding computer simulations, genetic algorithms that are used to simulate fauna, and the creation of worlds for simulation runs.

2.1 Computer Simulations as Tools for Understanding the Real World

Computer simulations have become a cornerstone of contemporary scientific inquiry, providing a virtual means to explore systems that are complex, large-scale, or otherwise inaccessible to direct experimentation. They allow researchers to construct a `model` of a real-world system and then conduct experiments on this model, observing the consequences of varying parameters and conditions [2, 1]. In many fields, from climate science to astrophysics, simulations have come to complement theory and physical experimentation, providing insights when direct experiments are too costly or impossible [2]. Crucially, simulations can reveal emergent behaviors of complex systems that might be analytically intractable, thus improving our understanding of underlying processes [7, 12].

2.1.1 Types of Simulations

There is a broad spectrum of simulation approaches. At one end are `mathematical simulations`, which involve solving equations or running algorithmic models derived from theoretical principles (for example, simulating fluid dynamics or planetary orbits). These often extend traditional mathematical models by leveraging computational power to handle nonlinearities or large datasets. At the other end are `agent-based models` (ABMs), which simulate the interactions of individual agents (e.g., organisms, people, or components) to study how collective phenomena emerge from micro-level rules [7]. Agent-based simulations have proved especially valuable for exploring emergent phenomena in ecology and social science, as they allow patterns to arise *from the bottom up* in a way not easily captured by equations [7]. Indeed, ABMs can instantiate hypothetical scenarios as a form of fiction: they create an artificial world populated by agents, which researchers interpret to glean insights about real-world dynamics [7]. By contrast, equation-based simulations rely on aggregate quantities and often require different interpretation, focusing on the cor-

respondence between computed outputs (e.g., temperature trends or stress distributions) and observed data or analytical solutions.

2.1.2 Model Construction and Validation

Developing a reliable simulation involves several stages. First, scientists build a model by selecting key features of the real system and translating them into formal rules or equations [12]. This process invariably involves *idealizations* and simplifications: only the most relevant aspects of the target phenomenon are included, while complicating details are omitted. Next, the model is implemented as a computer program. Careful *verification* is needed at this stage to ensure the code correctly implements the intended model (e.g., debugging to eliminate programming errors). After the model is running, it must be *validated* against empirical evidence [6]. Validation checks whether the simulation outputs correspond to real-world data or known behaviors of the system. A variety of techniques can be used: comparing simulation results with experimental measurements, calibrating model parameters to improve fit with observations, and performing sensitivity analyses to see how robust the outcomes are to changes in assumptions [2, 6]. It is widely recognized, however, that no simulation can be definitively *proven* true; natural systems are open-ended and many different models might reproduce the same observations. As Oreskes *et al.* (1994) famously argue, full verification and validation of numerical models is impossible in principle—models can only be confirmed partially by their agreement with data, never absolutely verified [6]. Therefore, model validation is an ongoing, iterative process of building trust in the simulation's predictive power rather than a one-time conclusory test [2].

2.1.3 Interpreting Simulation Data

Once a simulation model is validated to a satisfactory degree, scientists use it to conduct virtual experiments and generate data about possible behaviors of the system. These simulation outputs must then be interpreted with care. Often, the output of a large simulation is itself a complex dataset (for instance, gigabytes of climate model predictions or thousands of runs of an agent-based model), requiring statistical analysis and visualization to extract meaningful patterns. In this sense, simulation results function similarly to experimental data: researchers examine trends, uncertainties, and anomalies in the output to draw inferences about the real-world system [1, 7]. One challenge is that the correspondence between simulation and reality is indirect. The simulation data reflects the behavior of the model, not the world itself, so scientists must map those results onto their real-world counterparts via theoretical understanding and domain expertise [12]. For example, an emergent pattern observed in an ecological ABM might suggest a hypothesis about real ecosystems, but additional reasoning is needed to argue that the pattern would occur outside the computer. Because simulations can sometimes create detailed narratives about what could happen under certain conditions, interpreting their data often requires a careful approach [7]. The researcher must distinguish which features of the output are artifacts of the model and which are candidate explanations for phenomena in nature.

2.1.4 Philosophical Perspectives on Simulation

The rise of simulation in science has prompted significant philosophical debate about how simulations yield knowledge. Some scholars initially argued that computer simulations pose no fundamentally new epistemological issues: they claimed that any challenges in

using simulations are analogous to those in traditional theoretical modeling [10]. From this perspective, simulations are essentially extensions of mathematical models, so the usual concerns about idealization, representation, and confirmation apply without the need for a novel analysis [10].

However, other philosophers contend that simulations do introduce new issues and opportunities. A central epistemological question is whether and how simulation results can constitute evidence about the real world. Critics of simulation noted that if a result is merely derived from assumptions, it might be a *prediction* rather than evidence – something that still awaits empirical confirmation. Recent work has addressed this concern: for example, one recent analysis argues that simulation outputs can serve as *indirect evidence* for hypotheses about real-world phenomena, by indicating the likely existence of observable evidence even before that evidence is obtained – consider how climate models inform policy by providing evidence of future warming trends. The key is that the credibility of a simulation's findings rests on the quality of its construction and its track record of empirical adequacy [2, 6].

Philosophers have also noted the role of *models as fictions* in science: even if simulations involve imaginary scenarios or simplified agents, they can still be immensely useful in understanding reality, much like how fictional stories can illuminate truths [7, 12]. The fictive nature of simulations means scientists must interpret them with caution, but it does not invalidate the knowledge gained.

In summary, computer simulations are indispensable tools in modern science for understanding the real world. They enable experiments on models that complement theoretical analysis and physical observation, often uncovering emergent phenomena and subtle insights that would otherwise remain hidden. Building and using simulations involves careful model design, continual validation, and thoughtful interpretation of data. While initially met with skepticism, simulations have earned their epistemological credentials: when properly validated, they contribute to scientific knowledge not just by illustrating known theory, but by generating new hypotheses and evidence about the world [2]. As our ability to simulate complex systems grows, so too does our appreciation of the nuanced role simulations play in bridging models and reality.

2.2 Genetic Algorithms

Genetic algorithms (GAs) are a family of computational models inspired by evolutionary and biological processes. These algorithms encode potential solutions to a problem with simple chromosome-like data structures and apply recombination operators, such as selection, crossover, and mutation, to these structures while preserving critical information [13].

GAs were modeled on real-world biological processes in the hope of harnessing nature's great adaptability, and its ability to specialize and fill in niches in its environment. They allow for the modeling of natural systems by that they were initially inspired. These models are not intended to be precise simulations of the real world, but rather `idea models` [5] that offer us the opportunity to better understand these systems. With these models in place, the data extracted by running simulations can be used to test our hypothesis, observe the tendencies of the models, and see how these systems react when change is introduced into the parameters of the model.

Individual solutions are evaluated on the basis of their genetic information using a fitness function, that serves as the method used to guide the GA toward a solution. This means

that each problem to be solved using a GA has to have a uniquely designed fitness function, so that its generated solutions align with the desired solution.

An iteration of a GA consists of an evaluation process using the fitness function and the application of the operators of selection, crossover, and mutation, that enable the algorithm to transition from one population of candidate solutions to a new and, hopefully, better population of candidate solutions. When every individual has had its fitness value computed, this value is then utilized in the selection operator, that preferentially selects individuals with a higher fitness value. Once a pair of individuals has been selected, the genetic information of these individuals is `mixed` using the crossover operator that enables the algorithm to create new individuals from its combinations [11]. After the crossover operator, some implementations of GAs also utilize the mutation operator, that introduces a small probability to add randomized changes to the genetic information of the new individuals.

The use of the fitness function and these operators constitute one iteration of a GA. The goal of this iteration is to create a new and fitter population of candidate solutions from an existing population. This process is repeated multiple times and ends once a termination event occurs, that can be defined as either reaching the maximum number of simulation steps, achieving the desired solution, or simply when the overall rate of change for our population decreases below a certain threshold.

However, GAs are not universally effective in providing the best solutions. The use of GAs is generally considered desirable when the search space is large, is not known to be smooth or well understood, or if the fitness function for candidate solutions appears to include some noise [5]. If some of these traits do not apply to the problem at hand, it might be worth to analyze the problem in more depth, and to try and find out if it would not be better to solve the given problem using algorithms that rely on domain-specific heuristics, rather than by using general-purpose methods like GAs.

The first thing that has to be done before the start of a GA is to initialize it with a starting population of candidate solutions. This first population is required to initialize our algorithm, as all future populations will be derived from this first generated set of individuals. It is generally desirable to have the initial population vary in its genetic information, because a homogeneous population would render both the selection and crossover operators obsolete until some change could be introduced by gene mutations.

2.2.1 Important terms for genetic algorithms

When it comes to GAs, there are multiple terms that are commonly used and need to be explained so that the processes that make up a GA can be properly understood. In this section, there is a brief overview of the basic terms and operators that have been adapted for GAs from the real world evolutionary processes. It is important to state that even though these terms have been adapted from the real world, they represent only major simplifications of their real-world namesakes.

Individual

An individual in GAs broadly represents a candidate solution to the problem the GA is trying to solve. It is defined by its genetic information that is stored in its genotype.

Population

the population is the set of all current individuals.

Gene

A gene represents a specific trait of an individual solution – this can mean a single gene will define the height of an animal, while another might describe its feeding patterns. A gene can be represented by a wide array of data types, such as a binary string, an integer value, or a real value where the selected data type should be based on the specific needs of the implementation.

Chromosome

A chromosome signifies a grouping of genes. However, in GAs, it often makes up the entirety of an individual's genetic information, making it interchangeable with terms such as genotype or candidate solution.

Genotype

The genotype represents the entire genetic information of an individual, that during fetal and later development gives rise to the phenotype of the organism [5]. So in effect it is composed of a set of chromosomes that are further made from a set of genes, but in GAs a genotype is often seen as being made up of a single chromosome – making the use of these terms interchangeable.

Phenotype

The phenotype is the physical expression of the animal's genotype – phenotype is derived from an animal's genotype.

Fitness function

The fitness function enables GAs to evaluate and compare candidate solutions, so it must be uniquely designed to allow the algorithm to accurately assess how well an individual aligns with our desired solution. This means that for each problem that is to be solved using GAs, there is a need to create a unique fitness function. For each individual a numerical value is first computed and later evaluated against the fitness value of an `average individual` from the current population. This results in a relative ranking, that will be used to determine the probability for an individual to be selected during reproduction, and thus its chance to contribute to the creation of our next generation [3].

In the same way that animals compete in nature for survival and reproduction, the fitness function enables GAs to simulate this natural competition artificially. The important part is to define the conditions under that an individual is considered fit, as these conditions directly influence the overall effectiveness of this algorithm. However, the definition of a fitness function can sometimes be the hardest part in implementing a GA, and in some cases the creation of one is entirely impossible.

Selection

Selection in GAs is a process governed by the fitness function, that serves as the basis for selecting pairs of individuals from the population for reproduction. The primary goal of this process is to improve the average quality of the population with each successive generation. By focusing on the fitness values of individuals, selection aims to accelerate

convergence toward an optimal solution while reducing the risk of becoming trapped in local optima. However, the impact of an individual's fitness on its chance of reproduction must be carefully balanced because, on the one hand, a strong correlation can lead to a loss of diversity within our population and, on the other hand, a weak correlation will simply slow our rate of convergence [3].

The selection process is designed to mimic natural selection, as described in Darwin's theory of evolution. In nature, more capable individuals are better equipped to survive against predators, overcome diseases and obstacles, live longer, and produce offspring. Similarly, in GAs, fitter individuals are given a higher probability of being selected as progenitors for the next generation. This strategy helps to ensure that beneficial traits are propagated, thereby increasing the overall fitness of the population over time.

Popular implementations of the selection operator are:

1. **Roulette wheel selection:** Selects individuals with a probability proportional to their fitness values. Showcased in Figure 2.1.
2. **Tournament selection:** Randomly samples a group of individuals and selects the best among them.
3. **Rank selection:** Ranks the population by fitness and selects based on position in the ranking rather than raw fitness.

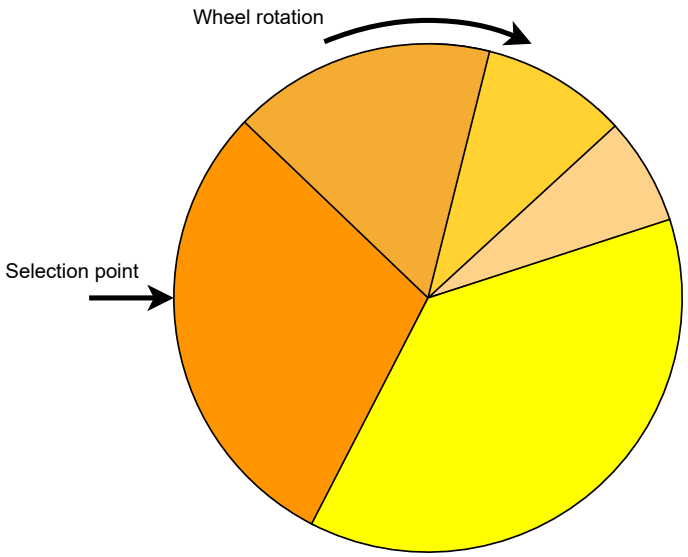

Figure 2.1: Example of roulette wheel selection operator with 5 individuals. Each individual is shown representing some portion of the roulette wheel, the size of this portion is proportional to the individual's fitness value. The roulette wheel preferentially selects for highly fit individuals, but also makes it possible for less fit individuals to be selected, thereby helping keep the population diverse.

Crossover

Crossover is a fundamental operator in GAs that ensures overall effectiveness by enabling the creation of new individuals through the combination of genetic information from parent individuals [11]. This process occurs at the level of chromosomes and genes, making crossover a critical mechanism for the exchange of information between individuals. It plays a central role in determining the algorithm's rate of convergence, serving as the main means of transferring traits across generations.

Typically, two individuals (parents) are selected from the population, where their genetic material is combined to produce offspring. Ideally, the offspring are of higher quality than their parents, allowing the GA to move along the gradient of the fitness function toward better solutions. There is also a probability that the offspring will have a lower fitness score than their parents. However, this is also desirable as it enables the GA to explore less optimal paths, thereby allowing it to potentially overcome being stuck in a local optima.

When it comes to the crossover operator, there are multiple ways to go about its implementation, where the method that is finally chosen is most influenced by the data type representation that is utilized for individuals' genes.

Popular implementations of the crossover operator are:

1. **One-point crossover:** Chooses one crossover point and swaps the genetic material beyond that point between two parents. Showcased in Figure 2.2.

2. **Two-point crossover:** Chooses two crossover points and swaps the genetic material between those points.

3. **Uniform crossover:** Swaps each gene with a fixed probability independently.

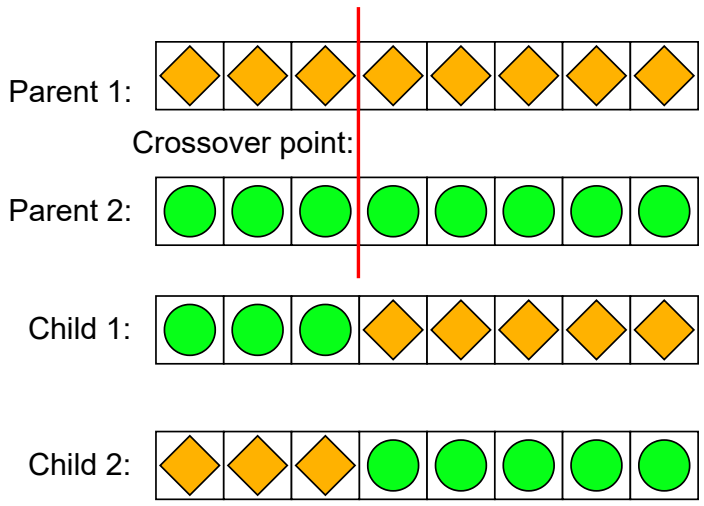

Figure 2.2: In one-point crossover, a single crossover point is selected and used to divide both genomes. By swapping the latter parts of the genomes, 2 new individuals with genes from both parents are created.

Mutation

Mutation in GAs works by modifying values within the genetic structure of individuals and typically occurs after the crossover operator. The application of mutation is governed by a predetermined mutation probability, that is usually set to a very low value [5]. The reason why the mutation probability should be kept at a low value is to ensure that the algorithm does not devolve into a purely stochastic search algorithm. Despite being considered a secondary reproductive operator, mutation plays a crucial role in preventing premature convergence of the population and allows GAs to explore the solution space near the current population. By having a small chance to introduce relatively tiny random changes to an individual's genome, mutation can give GAs access to valuable genetic information that may have been otherwise inaccessible, helping the GA escape local optima.

Popular implementations of the mutation operator are:

1. **Bit string:** Iterates over every bit in the bit string and with a given probability flips their values. An example of using this operator can be seen in Figure 2.3.
2. **Swap:** Selects two positions in the genome and swaps them.
3. **Gaussian:** Adds random Gaussian noise to a gene value.

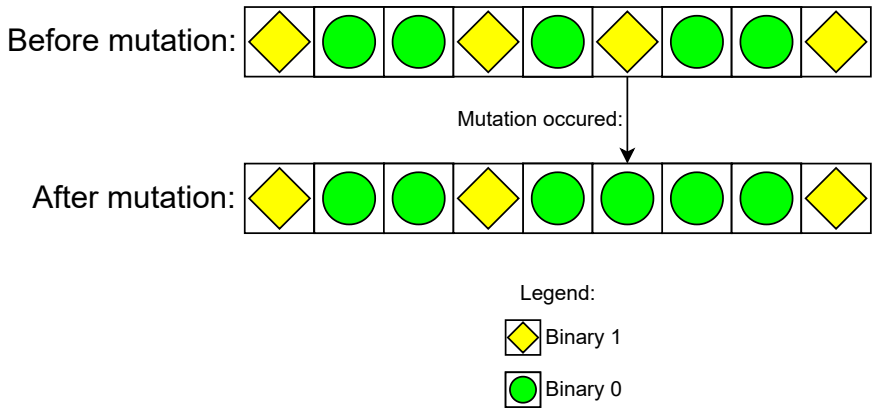

Figure 2.3: Bit string mutation operator iterates over every bit and flips its value with a given mutation rate probability. In this example the 6-th bit of the genome was flipped from its initial value of 1 to 0 by the mutation operator.

2.3 Procedural Map Generation

Procedural map generation is done using procedural noise functions that allow the creation of worlds at minimal expense, and so enables simulation runs to be done in varied worlds, that is imperative, because without the ability of running simulations in a different world every time, the GA would simply optimize to some set of hand-crafted worlds, thereby limiting the simulation's findings.

Procedural Noise Functions

Procedural noise functions (PNFs) [4] in computer graphics are mathematical functions that produce pseudo-random values in a smooth and repeatable way. PNFs enjoy widespread use in areas such as computer graphics and procedural content generation, as well as other fields that benefit from introducing subtle variations into predefined content. Their effectiveness in this aspect comes in part from the fact that humans have difficulties detecting differences in higher-order statistics.

PNFs occupy minimal space because their size is determined by the size of their source code rather than the volume of generated data. They also support random access, meaning any coordinate can be evaluated independently of previous evaluations. Their continuous and multi-resolution characteristics enable them to produce detail at essentially any scale, making them suitable for both large-scale structures and fine-grained features. Finally, PNFs are parameterized, allowing an entire family of related noise outputs to be produced by changing their parameters. Below is a brief overview of some commonly used parameters in PNFs:

1. **Seed**: Drastically changes the output while preserving the overall characteristics of the noise.
2. **Frequency**: Governs the level of detail – with higher frequencies producing smaller, more frequent features, while lower frequencies give broad, gradual variation.
3. **Octaves**: Allow the stacking of noise at varying frequencies and amplitudes to produce richer, more fractal-like results.
4. **Lacunarity**: Defines how much the frequency shifts for each additional octave.
5. **Gain**: Dictates how quickly the amplitude changes for each additional octave.

The process of creating a parametrized Perlin noise is shown in Figure 2.4.

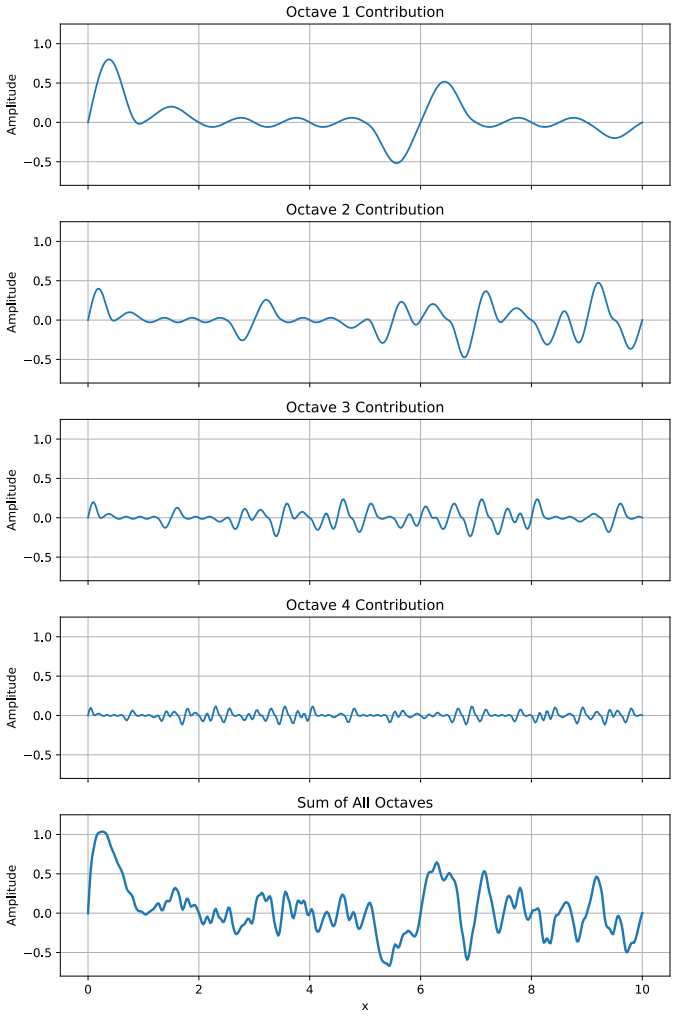

Figure 2.4: Showcase of Perlin noise utilizing parameters Lacunarity, Gain, Frequency and Octaves. The image showcases the generation of noise using 4 octaves, frequency set to 1, lacunarity set to 2 and gain set to 1/2.

Thanks to these qualities, another field that has seen rapid adoption of PNFs is terrain generation for game developers. Here, PNFs are used to programmatically generate vast, randomized, and importantly natural-looking terrains.

PNFS come in many different forms, they were first conceived by Ken Perlin in 1983 and later published in his 1985 article `An image synthesizer`, and so the first PNF bears its authors' name – Perlin noise. However, since then the field of study around PNFs has grown substantially, and today there exist many types of noise.

Here are some notable examples:

1. **Perlin noise**: Developed by Ken Perlin in 1983 [8], Perlin noise produces smooth, gradient-based noise patterns ideal for simulating natural phenomena like terrains, clouds, and water. Its scalability allows for the generation of detailed textures by combining multiple octaves. Showcased in Figure 2.4.

2. **Simplex noise**: Introduced by Ken Perlin in 2001 [9], Simplex noise addresses some limitations of classic Perlin noise, such as computational complexity and directional

artifacts. It is more efficient, especially in higher dimensions, making it suitable for applications requiring 3D textures or animations.

3. **Worley noise**: Introduced by Steven Worley in 1996 [14], this function divides space into cells based on proximity to a set of seed points, creating patterns reminiscent of natural structures like stone or cracked surfaces. Worley noise is often used for the generation of textures that mimic organic materials.

Perlin noise 2D example

Perlin noise falls under the lattice gradient noise function classification [8], where it is defined on a grid where each lattice point on this grid is assigned a pseudo-random gradient vector. These gradient vectors dictate the direction and intensity of variation in the noise function. Showcased in Figure 2.5.

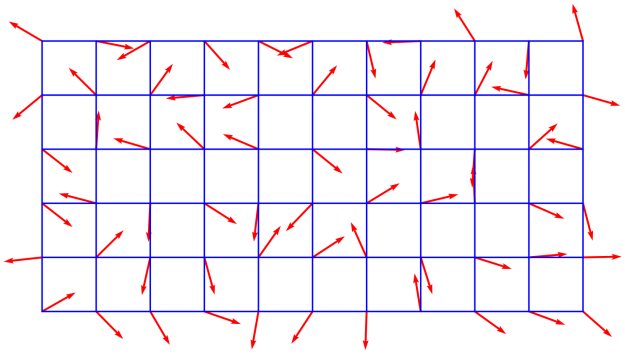

Figure 2.5: Integer lattice where each point has their gradient vector visualized.

To compute a noise value at a specific point (x, y), the first step is to identify the grid cell that contains this point. Then from each corner of this cell distance vectors are created to the evaluation point (x, y). Showcase of gradient and distance vectors and how they are used is in Figure 2.6.

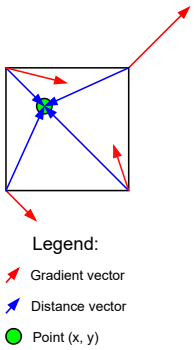

Figure 2.6: Gradient vectors and distance vectors to coordinate (x, y) showcased on a grid cell.

Once the distance vectors have been computed, take for each corner of the cell the dot product of both the gradient vector and its corresponding distance vector, and use a smooth interpolation function to blend the dot products across the grid cell. All that remains now is to repeat this process for all the other coordinates where noise is to be calculated. The result of this process can be seen in Figure 2.7.

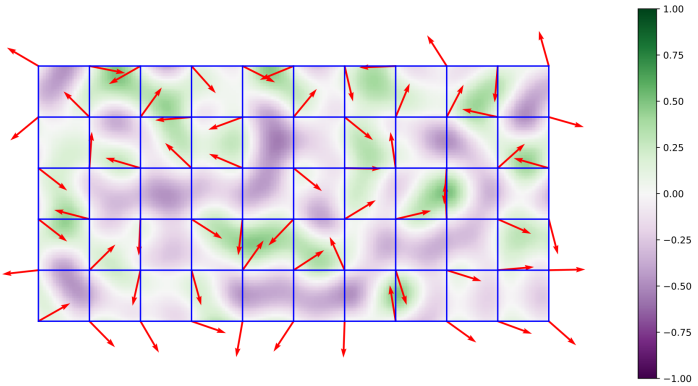

Figure 2.7: The result of using Perlin noise to fill a 2D plane.

Chapter 3

Simulation Design

Building upon the theoretical concepts outlined in Chapter 2, this chapter details the structural design of the ecosystem simulation. The focus here is on the fundamental components or *building blocks* that constitute the simulated world and its inhabitants, the mechanisms driving their behavior, the sources of variation within the environment, and the overall initialization and dynamic flow of the simulation. This design prioritizes modularity and conceptual clarity, laying the groundwork for the implementation discussed in Chapter 4.

3.1 Comparison of Game Engines for Ecosystem Simulation

Ecosystem simulation projects require a game engine that offers a robust architecture and flexible scripting capabilities. We compare Unity, Unreal Engine, and Godot in terms of their general architecture and scripting models, with emphasis on features pertinent to simulation. Unity and Unreal are industry-standard engines with extensive feature sets, while Godot is a newer open-source engine chosen for this work, warranting a deeper analysis.

3.1.1 Unity Engine

Unity is a widely-used proprietary engine known for its component-based architecture. In Unity, every entity in a scene is a `GameObject` to that developers attach modular `Components` (scripts, physics bodies, renderers, etc.) that collectively define the object's behavior and data. This design favors composition over inheritance and allows developers to mix and match behaviors easily. Unity scenes are hierarchies of GameObjects, enabling nested structures (with a root and children) similar to a node graph.

For scripting, Unity uses C# as the primary language, having deprecated its older `UnityScript` (JavaScript-like) and Boo languages by 2017. C# scripts (often subclassing a `MonoBehaviour`) are attached as Components to GameObjects, aligning with the engine's design. Unity provides an integrated development environment and out-of-the-box tools to make the production of games easier for developers. Unity's extensibility is supported by a rich ecosystem of plugins and an asset store, though the engine core is closed-source. Unity's versatility and cross-platform support have made it popular in both game development and simulation domains.

3.1.2 Unreal Engine

Unreal Engine (UE) is a high-end engine renowned for its powerful rendering capabilities and is commonly used in AAA games and realistic simulations. Architecture-wise, Unreal employs an entity-component design broadly similar to Unity's, where game entities are represented as `Actors` (with a class inheritance hierarchy in C++), that can aggregate various `Actor Components` to extend functionality (e.g. a Character actor may have movement, collision, and rendering components).

The engine historically featured its own scripting language (UnrealScript), but since Unreal Engine 4 this was replaced by direct C++ API usage along with an accessible visual scripting system called `Blueprints`. Developers can write performance-critical code in C++ and use Blueprints for higher-level game logic; the latter is especially praised for enabling designers to script behaviors via a node-based interface without writing code. Unreal's source code is openly available to developers (under a license), allowing profound engine customization.

This level of extensibility, combined with advanced graphics and physics, makes Unreal suitable for complex simulations. On the other hand, Unreal's sophisticated architecture comes with a steep learning curve and higher hardware demands, that may be less ideal for small-to-medium projects. Its design is geared toward large-scale applications: the engine supports extensive multi-platform deployment and has garnered industry recognition. For ecosystem simulations requiring photo-realism or very large worlds, Unreal provides strong capabilities, while offering a mix of native code control and rapid development through Blueprint scripting.

3.1.3 Godot Engine

Godot is an open-source (MIT-licensed) engine released in 2014 that has rapidly grown in popularity, especially in the indie development community.Architecturally, Godot is built around a flexible scene graph of `Nodes`. Every game entity is a Node, and nodes are organized in a tree hierarchy `scene`. This system allows a scene to be saved as a reusable resource and instanced as a sub-tree within other scenes, encouraging a highly modular project structure. Nodes come in many built-in types (e.g. spatial, UI, physics bodies, etc.), and can also serve as generic containers for user-defined behaviors. Godot's design philosophy emphasizes object-oriented composition and intuitive workflow. In practice, this means that instead of enforcing an Entity-Component-System paradigm, Godot lets developers structure scenes and nodes in whatever way fits the project, which can benefit clarity in a simulation design. While not an ECS, Godot does support a form of component-based development by allowing nodes to be attached to other nodes (for example, adding a collision shape node under a physics body node acts like adding a component).

Godot's primary scripting language is GDScript, a high-level, Python-like language designed specifically for tight integration with the engine, but C# is also supported. GDScript is object-oriented, imperative, and gradually typed, using indentation-based syntax similar to Python. It was created to make common engine interactions (like manipulating nodes and scene trees) straightforward and optimized; for instance, signals (Godot's messaging system) and timers are built-in language features. The tight coupling of GDScript with the engine yields a very rapid development cycle – ad scripts can be edited and reloaded on the fly in the editor, and the language's performance is sufficient for a large class of gameplay and simulation logic.

For performance-critical code, or to interface with existing libraries, Godot 4 introduced GDExtension (succeeding the older GDNative mechanism). GDExtension is a C-based API that allows the engine to load compiled libraries and treat classes within them as first-class engine classes. This means developers can write modules in C++ or other languages (without needing to rebuild Godot itself) and then use those as if they were built-in. Notably, the community provides official Rust bindings for Godot via GDExtension, allowing Rust to be used for game logic or systems-level programming in Godot. This Rust integration leverages Godot's C API to offer an "ergonomic, safe and efficient" interface between Rust and Godot's runtime. In practical terms, one can implement complex simulation systems in Rust or C++ as plugins, and expose them to the Godot editor and to GDScript. Such extensions operate with near-native performance and can be as seamlessly integrated as core engine features.

The ability to drop down to native code when needed, combined with the high-level ease of GDScript, gives Godot a wide spectrum of scripting extensibility. Another strength of Godot for structured development is its clarity and lightweight design. The engine's complete source is available, that not only allows verification of engine behavior but also enables custom modifications if required. Godot's editor and scripting systems are designed to scale from small to fairly large projects, with support for organized grouping of scripts, native singleton autoloads for global state, and a scene inheritance system for reusing and extending scene templates.

In summary, Unity offers a mature, component-based framework with C# scripting and a vast ecosystem, making it a strong general-purpose choice (including for simulation environments) if proprietary constraints are acceptable. Unreal Engine provides unparalleled graphic fidelity, C++ power, and Blueprint scripting, suitable for large-scale simulations that require engine-level tuning and realism. Godot, meanwhile, provides a balance of simplicity and flexibility with its node-based architecture and choice of GDScript and C#, augmented by the option of native code extensions. Godot's open-source nature and modest footprint make it attractive for academic projects and structured simulations where transparency and extensibility are valued. The choice of engine will ultimately depend on specific project requirements – for an ecosystem simulation emphasizing modularity, customizability, and an open development process, Godot is an appropriate and well-justified selection.

3.2 Simulation Interface

The application was designed with the use of the Godot game engine in mind, so it is divided into 2 distinct components – a *GDScript component* and a *Rust component*. These components interact with each other through exposed functions that are designed to run in batches to lessen the overhead cost of using an FFI.

The GDScript component is responsible making calls to the Rust component – this includes initialization of the Rust component, the extraction of data from the simulation to display the current simulation state, to store CSV data for visualization purposes at the end of a simulation run, and deciding when a turn is to be processed. It also handles the visualization of the simulation state by showcasing the current simulation statistics and the world map.

The simulation run takes place wholly within the Rust component, which stores the data for all the animals and the world map, and handles the processing of animals and the modification of animals and world map data during each simulation step.

3.3 Setting Simulation Input

The simulation parameters govern the fundamental rules and constants of a simulation run. Key parameters include:

- Map dimensions (*width*, *height*).
- Noise parameters for map generation (seed, frequency, octaves, etc.).
- Genetic parameters (*resource cost of genes*, *mutation probability*, etc.).
- Environmental parameters (*duration of scents*, *decay rate of meat*, etc.).
- Simulation constants (*number of initial animals*).

The design supports multiple ways to set these parameters:

1. **Default Values:** Hard coded default values exist within the simulation logic as a fallback.
2. **Configuration File:** This enables the loading of parameters from a CSV file. This allows for easy saving and sharing of specific simulation parameter setups.
3. **GUI Override:** The Godot frontend provides UI elements that allow the user to inspect and override some of the default or file-loaded parameters before starting a run.

Once the parameters are initialized in the Godot component, the parameters are packaged and sent to the Rust component where they will be used. This process is described in Chapter 4.2.

3.4 Environment Design and Variance

The simulation environment is designed as a 2D grid of hexagonal tiles, providing a more naturalistic neighborhood structure compared to a square grid. Each tile has a set of static and dynamic properties, where the user can modify the static properties of the world map. The design is to allow for simple changes to the simulation world and the simulation parameters utilizing Godot component's UI.

3.4.1 Tile Properties

Each map tile is designed to hold these properties:

- **Static Properties:** These define the inherent nature of the tile and are determined during world generation, remaining constant throughout a simulation run unless explicitly changed. They include:
  - `temperature`, `moisture`: Base climatic values influencing other properties.
  - `Biome`: Categorical type derived from temperature and moisture.
  - `Derived properties`: *max plant matter*, *plant matter gain*, etc.
- **Dynamic Properties:** These represent the changing state of the tile based on simulation events:

- `Current resources`: *plant matter*, *meat*, *hydration*.
- `Occupants`: *animal identifiers*.
- `Transient elements`: *scent trails*.

Individual tiles are drawn on the screen based on their *biome* properties. Biomes serve as a form of visualization, where each biome is associated with a certain color, and are specified by extracting the tile's *temperature* and *moisture* data. The specification of a tile's biome property is done using Whittaker's biome system, which groups certain moisture and temperature ranges, as showcased in Figure 3.1.

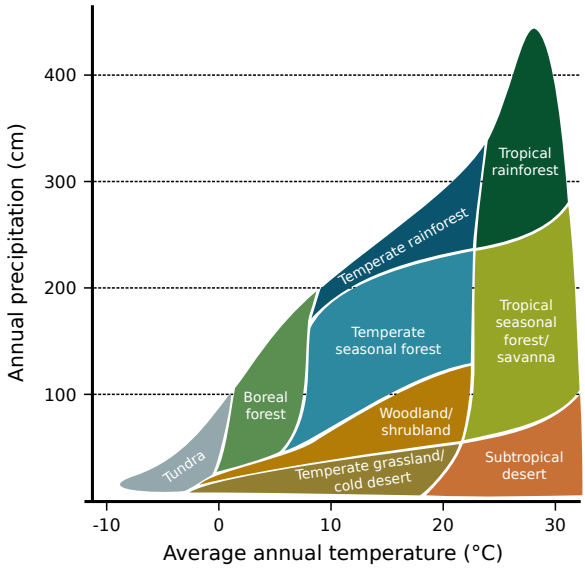

Figure 3.1: Biomes are specified for every tile contained within map, where each tile's biome is decided by the temperature and moisture ranges into which their properties fit.

3.4.2 Tile Resources

Each tile has 3 distinct types of resource that it can contain, which animals need to consume to survive. Available resources include:

- **Meat:** When an animal dies, an amount of meat proportional to the animal's *mass* characteristic is placed on top of the tile that it previously occupied. Meat spoils over time, after which it is removed from the tile properties.
- **Plant matter:** Plant matter accumulates on tiles at a tile-specific rate every turn until a tile-specific maximum amount is reached.
- **Hydration:** Hydration is fully replenished every turn.

3.5 Agent Design: The Simulated Animal

The primary actors within the simulation are autonomous agents that represent individual animals. Animals are designed with their qualities separated into a heritable genotype that is used in GA operators, animal characteristics (phenotype) that are derived from the animal's genome, and animal's current state.

3.5.1 Genetic Representation (Genotype)

Each agent possesses a genotype, conceptually analogous to its DNA, that dictates its inherent traits. This genotype is designed as a collection of individual *genes*, where each gene corresponds to a specific quantifiable characteristic. Based on the requirements identified in the theoretical exploration of Genetic Algorithms (Section 2.2), the genes are designed to be represented by normalized floating-point values within the range (0, 1). This allows for fine-grained variation and facilitates mathematical operations during inheritance and characteristic calculation. The core genes designed for this simulation include:

- **size**: Influences mass, resource needs, and fighting prowess.
- **speed**: Affects the ability to move on the World Map and during fights.
- **food preference**: Determines dietary inclination on a scale from pure herbivore (0) to pure carnivore (1).
- **mating rate**: Governs the frequency of attempting reproduction.
- **stealth**: Affects the likelihood of avoiding detection by other agents.
- **detection**: Affects the likelihood of detecting other agents.

This set forms the inheritable blueprint passed down through generations, subject to genetic operators of crossover and mutation as described in Section 2.2.

3.5.2 Observable Traits (Phenotype)

The genotype along with certain simulation parameters directly influence the phenotype of an animal. This forms a set of characteristics that are not directly inheritable. Key characteristics include:

- **metabolic rate**: Describes the rate at which an individual will lose resources every turn. This value is dependent on the animal's gene values and *simulation parameters* – where each gene has an associated cost defined. The calculation is showcased in Chapter 4.4.
- **life span**: Maximum potential age. If an agent reaches their maximum life span they are immediately marked for deletion.
- **vore type**: A categorical trait (carnivore, herbivore, omnivore) determined by thresholding the *food preference* gene value. *Carnivores* and *herbivores* represent exclusive *meat eaters* and exclusive *plant eaters*, while Omnivores represent a union between these two types. An Omnivore chooses its primary strategy as that of a *herbivore*, or of a *carnivore* – the option is influenced by the gene value of *food preference* with higher values meaning a preference for a *carnivorous* strategy, and lower values signifying a preference for a *herbivorous* diet.

The most important of these traits is the *metabolic rate* of an animal that determines the resource cost that is deducted from the animal each turn based on the genetic information of the animal. All genes have user-definable resource costs that contribute to the total resource drain that is deduced from the agent's resources every turn.

In addition to these relatively static characteristics, each animal maintains a dynamic internal `state` that changes each turn.

3.5.3 Agent Decision Making

Animal behavior is designed to be driven by fundamental needs, forming a simple hierarchy for action selection:

1. **Survival Needs (Highest Priority):** Maintaining its vital resources above critical levels. Agents below certain thresholds for these critical resources are designed to prioritize actions like eating, drinking, or hunting. The type of food an animal will intake is described by an animal's `vore` type (Herbivore, Omnivore and Carnivore) – where a Herbivore is only capable of ingesting `plant matter` for nutrition, a Carnivore is only capable of ingesting `meat`, and an Omnivore represents a union of the two.

2. **Reproduction Need (Medium Priority):** If survival needs are met, animals that are ready for reproduction will prioritize seeking compatible mating partners. An animal can reproduce with a different animal only if the other animal is also ready to mate and if the `euclidean distance` of their genomes is within a definable range. If no mate was found, the second step is to look for scent trails on the current tile and determine if a potential mate recently passed the current tile.

3. **Default Behavior:** If no higher priority needs are active, the agent is designed to perform a default random movement action.

Unlike Carnivores and Herbivores that are forced into their respective dietary strategies, an Omnivore instead randomly chooses every turn based on its `food preference` gene value whether to first try a Carnivorous or Herbivorous strategy, and if the initial strategy fails, the other strategy is further explored in search of resources. This design uses simple, prioritized rules based on internal state and local environmental perception to generate emergent behaviors, rather than complex pre-programmed plans. The implementation details were deferred to Chapter 4.4.

3.6 Simulation Initialization

Setting up a simulation run involves configuring parameters, generating the initial world state, and populating it with the first generation of agents. The design allows for flexibility in this process.

3.6.1 World Generation and State Transfer

Based on the configured parameters (especially map dimensions and noise settings), the static world map is generated:

1. **Noise Generation (Godot):** The Godot frontend uses its *FastNoiseLite* capabilities to generate the raw noise values for temperature and moisture across the grid.

2. **Static Tile Initialization (Godot):** Godot iterates the grid, creates instances of its `Tile` class, populates them with the calculated static properties, and stores these objects in a `Dictionary` keyed by the tile indices.

3. **State Transfer (Godot to Rust):** This dictionary of `Tile` objects is passed to the Rust backend.

4. **Internal Map Creation (Rust):** Rust receives the dictionary, iterates through it, parses the static properties from each Godot object, calculates derived static values, initializes the dynamic properties, and stores them.

This process establishes the initial environmental conditions within the Rust simulation core.

3.6.2 Initial Population

The creation of an initial population is a mandatory step before the simulation can start. The size of the initial population can be specified by the user in a variety of options as discussed in Chapter **??**. After the map and parameters are set in Rust, the initial animal population is created on the side of Rust. The creation of the initial population is done by creating a user specified number of animal instances and placing them at random locations within the world map.

3.7 Simulation Dynamics and Capacity for Change

The simulation is designed to evolve dynamically over discrete time steps (turns), driven by both internal mechanics and potential external influences. Snapshots of simulation data are periodically stored in a CSV file which is then fed into a Python script used for visualization purposes.

3.7.1 Turn-Based Progression

The core dynamic for this simulation is a turn-based loop that divides each turn into distinct phases. These phases and the actions that are performed during each phase are described in Chapter 4.3. Phases are designed to be logically separate and aim to reduce tensions between animal actions while also allowing for easier parallelization on these discrete steps.

Each turn progresses through agents deciding on their actions based on needs and local conditions, followed by the resolution of interactions, execution of relevant actions, state changes (resource consumption, aging), the realization of movement, and the application of births and deaths while also handling the *replenishing* of map resources. This cyclical process allows complex behaviors and population-level trends to emerge over time from simple agent rules and environmental interactions.

3.7.2 Internal Dynamics

Change arises naturally from within the simulation:

- **Resource Fluctuation:** Plants, water, and meat are consumed by animals and regenerate/decay according to tile properties, creating dynamic resource landscapes.
- **Population Changes:** Births increase the size of the population, while deaths (starvation, dehydration, age, and predation) decrease it, leading to population dynamics influenced by the carrying capacity and the local genetic landscape.
- **Genetic Drift and Selection:** Random inheritance and mutation, combined with emergent selection pressures based on survival and reproduction, cause the genetic makeup of the population to constantly change over generations.

- **Movement and Spatial Distribution:** Animals moving across the map leave behind *scent trails*, which can be used by animals to locate prey or mating partners, not just on the local tile, but also by following the animal's scent.

3.8 Evolutionary Mechanism Design

The simulation implements core concepts from genetic algorithms, adapted to an agent-based ecosystem model:

- **Genotype**: Represented by the `AnimalGenes` struct, containing floating-point values for key traits.
- **Fitness**: Unlike classic GAs with an explicit fitness function calculated each generation, fitness here is *implicit and emergent*. An animal's fitness is determined by its ability to survive (find food and water, avoid fatal fights) and reproduce within the simulated environment. There is no single numerical fitness score; success is the metric.
- **Selection**: Selection is also emergent. Animals that fail to meet their nutritional or hydration needs die and are removed. Animals that lose fights may die. Only surviving animals capable of finding compatible mates get to reproduce. This constitutes differential survival and reproduction based on trait interactions with the environment.
- **Crossover**: Handled by using a form of uniform crossover where each gene of the offspring is inherited from one of the two parents randomly.
- **Mutation**: Implemented within the gene inheritance logic. A small probability determines whether a gene's value will be slightly altered by adding random noise from a defined range and clamping the final value between 0 and 1.

This emergent approach aims to model natural selection pressures more organically than a predefined fitness function might allow.

Chapter 4

Ecosystem Simulation Implementation

This chapter details the practical realization of the simulated ecosystem as described earlier. Building upon the theoretical foundations of evolution and the design choices favoring a performant core, the work translates the abstract concepts into a functional digital environment. The result is a tool where users can observe virtual creatures, represented by simple data structures, as they live, reproduce, and adapt over many generations within a dynamically changing world.

The core focus of this work is the implementation of a flexible 2D ecosystem simulation where the agents' (animals') traits are genetically determined and subject to evolutionary pressures like resource scarcity, predation, and mate selection. It demonstrates how environmental factors interact with genetic traits to shape population dynamics and drive adaptation over simulated time, reflecting real-world ecological and evolutionary processes in a simplified, observable manner.

The goal of this work was successfully achieved by implementing a simulation platform within the Godot Engine. This platform features a Rust-based core that efficiently manages the environment state and simulates the life cycles, interactions, and genetic evolution of a population of animal agents according to defined parameters. The system allows users to configure initial world conditions, observe animal statistics, view population movements visually, and extract detailed statistical and individual agent data over time, enabling the analysis of emergent population trends and adaptive responses to environmental pressures.

4.1 Rust Backend Implementation

The computationally intensive core of the simulation was implemented as a separate Rust library crate, named `Simulation_Core`, designed to be compiled into a dynamic library (`cdylib`). This library is integrated into the Godot project using the GDExtension system, leveraging the `gdext` crate (godot-rust bindings) for seamless communication. The standard Rust package manager, Cargo, was used for managing dependencies, that primarily include `gdext`, the `rand` crate for random number generation, and the `rayon` crate for parallelism.

At the heart of the Rust library lies the `Simulation` struct, defined in `src/lib.rs` and exposed to Godot using the `#[derive(GodotClass)]` and `#[godot_api]` macros. This struct serves as the central state manager, encapsulating:

- The `world_map` (`HashMap<Vector2i, RustTileProperties>`): Storing the complete dynamic state for each tile.
- The `animals` collection (`HashMap<i64, Animal>`): Holding all active `Animal` instances, keyed by unique IDs for efficient access.
- The `params` (`Option<SimulationParameters>`): Containing simulation settings loaded from Godot.
- The `stats` (`AnimalStatistics`): Used for accumulating persistent statistics like death counts.
- An instance of `rand::rngs::ThreadRng` for stochasticity.
- A counter `next_animal_id` for generating unique agent identifiers.

This encapsulation ensures that the core simulation logic operates independently from Godot's scene tree and rendering loop, interacting only through the well-defined GDExtension interface. Key data structures like `RustTileProperties`, `Animal`, and `AnimalGenes` were implemented as Rust structs, mirroring their conceptual counterparts described in the design and holding all necessary state variables for the simulation's internal logic.

4.2 GDExtension Interface Implementation

The communication bridge between Godot and the Rust was implemented via functions exposed on the `Simulation` struct. Careful implementation of data conversion was necessary to facilitate data transfers between Godot and Rust, particularly for complex types. Functions that were exposed to Godot were also designed to work in batch, due to the overhead cost of making FFI calls.

Key interface functions were implemented as follows:

- **`set_simulation_parameters`**: Accepts a `Variant` from Godot, expected to be either a Godot `Simulation_Parameters` object instance or a `Dictionary`. It attempts to parse the input using object property access or dictionary key lookup, converting values to the corresponding Rust types within the `SimulationParameters` struct stored internally. Error handling and defaults ensures robustness against malformed input.
- **`set_map_for_rust`**: Accepts the initial map state as a Godot `Dictionary` where keys are `Vector2i` and values are Godot object instances (`Gd<RefCounted>`) of the `Tile_Properties` class defined in GDScript. The implementation iterates this dictionary. For each tile object, it extracts static properties (`temperature`, `moisture`, `biome`). Based on these, it calculates derived static properties and initializes the dynamic state fields within a `RustTileProperties` struct. This fully populated struct is then stored in the internal Rust `world_map` HashMap. This approach correctly handles the transfer of custom Godot objects.
- **`spawn_predetermined_animals`**: Takes an integer count. It loops `count` times, selecting a random valid tile position from the initialized `world_map` keys, generates a new `Animal` instance using the specific predetermined gene generation logic, assigns a unique ID, and inserts the animal into both the main `animals` HashMap and the corresponding tile's `animal_ids` list within the `world_map`.

- `process_turn_for_all_animals`: This function triggers the core simulation logic detailed in Section 4.3. It takes no arguments and returns nothing, modifying the internal Rust state directly.
- `get_tile_animal_counts`: Provides a map data retrieval method, returning a Godot `Dictionary` mapping tile coordinates directly to the *count* of animals on each tile. This functionality is used to visualize the number of animals on a given tile.
- `get_animal_statistics_dict`: Calculates snapshot statistics (averages, ranges) by iterating the current `animals` HashMap, combines them with accumulated statistics, and serializes the result into a Godot `Dictionary`.
- `get_all_animal_data`: Constructs a Godot `VariantArray`. It iterates the internal `animals` HashMap. For each animal, it creates a Godot `Dictionary` containing the `id, vore_type, age, and genes` (as a nested `Dictionary`) and pushes this onto the array. This function is specifically formatted for the Data_Logger class which is responsible for storing simulation data in a CSV file.

4.3 Core Simulation Loop Implementation

The `process_turn_for_all_animals` function orchestrates the advancement of the simulation state by one discrete time step. It implements the phased approach designed to manage dependencies and enable potential parallelism, operating entirely on the internal Rust data structures. Phases happen on each turn in this order:

1. **Reset Turn Statistics:** Temporary accumulators for statistics gathered within the turn (like total nutrition gain) are reset.
2. **Animal Decisions:** A list of current animal IDs is collected. The simulation then iterates through these IDs. For each animal not currently moving, its `decide_action` method is called. This method accesses the animal's state, its current tile's state (read-only from `world_map`), and potentially information about other animals on the same tile (read-only from `animals`). The decided `Action` (e.g., `EatPlant`, `Hunt(id)`, `MoveRandom`) is stored in a temporary `actions` HashMap. Potential conflicts (fights, mating) are queued separately.
3. **Interaction Resolution:** The queued fight requests are processed sequentially. The `Animal::fight` method is called, potentially resulting in an animal being added to a `deaths` list and the winner's intended action being updated (e.g., to `EatMeat`). Mating requests are then resolved; successful pairings result in a new `Animal` being added to a `births` queue and the parents' `ready_to_mate` status being reset.
4. **Action Execution & State Update:** The simulation iterates through the animals again. Non-moving animals execute their planned action from the `actions` map. This involves calling methods like `Animal::eat_plant_matter`, `Animal::drink`, or `Animal::begin_move_to_tile`, that mutate the animal's internal state and potentially the state of its current map tile (via a mutable reference). Following action execution phase, `Animal::update_animal_resources` is called, each agent's age is incremented, and checks for death by starvation, dehydration, or old age are performed, adding relevant IDs to the `deaths` list and updating corresponding counters in `self.stats`. Nutrition gain statistics are also tallied.

5. **Movement Execution:** Processes animals flagged as `is_moving`. Their remaining move timer is decremented, and if a move completes, the animal's `map_position` is updated, its ID is moved from the old tile's `animal_ids` to the destination tile's `animal_ids`, and a new `AnimalScent` is added to the old tile pointing to the destination tile.

6. **Apply Births and Deaths:** The `deaths` list is processed, removing animals from the main `animals` HashMap and their last known tile's `animal_ids` list. Meat is added to the tile where death occurred using `map::add_meat_to_tile_piles`. The `births` list is processed, adding new `Animal` instances to the `animals` HashMap and their respective tile's `animal_ids` list.

7. **Update Map State:** During this phase every tile is processed in parallel, where `plant_matter` is incremented by `plant_matter_gain`, `hydration` is set to its maximum value, instances of `meat` and `scent_trails` have their duration decremented and are removed once their duration reaches 0.

This structured execution allows optimizations to be performed for individual phases when required. For example, the `Update Map State` phase processes map tiles in parallel by utilizing `rayon::par_iter_mut()`, which calls the independent `map::replenish_tile` function for each tile, concurrently updating map state.

4.4 Agent Implementation

The individual agents are represented by the `Animal` struct in Rust. Its state is initialized either randomly (`spawn_initial_animals`) or through reproduction (`spawn_new`). The core behavioral logic resides in the `decide_action` method. It implements a priority-based decision tree:

1. Check if nutrition is below `seek_nutrition_norm` (and lower than or equal to hydration norm). If so, attempt to find food based on preferred strategy as given by `vore_type` and `food_preference`:

   - Carnivorous strategy:
     (a) Check for meat: If meat is on tile do (`Action::EatMeat`).
     (b) Check for *huntable* prey: If prey is on local tile do (`Action::Hunt`).
     (c) Check for *huntable* scents: If scent is on tile do (`Action::FollowScent`).
   - Herbivorous strategy:
     (a) Check for `plant_matter`: if `plant_matter` is on tile do (`Action::EatPlant`).
   - If no suitable food source exists: Do a random move (`Action::MoveRandom`).

2. Else, check if the hydration is below `seek_hydration_norm`. If so:

   - Check for water: If water is on tile do (`Action::Drink`).
   - If no water can be found on tile: Do a random move (`Action::MoveRandom`.

3. Else, check if `ready_to_mate >= 1.0`. If so:

   - Check for potential mates: If an *adequate* mate is on tile do (`Action::Mate`).

- Check for mate scents: If an *adequate* scent is on tile do (`Action::FollowScent`).
- If no adequate mate or scent found: Do a random move (`Action::MoveRandom`).

4. Else (no pressing needs): Do a random move (`Action::MoveRandom`).

An animal's characteristics are based on its genes and `SimulationParameters`. All characteristics are calculated and set by the `set_characteristics` method. Below is shown a code snippet where an animal's `food_consumption` and `water_consumption` are being calculated and set:

```
let base_activity_level = self.genes.speed.powf(params.speed_cost)
   + self.genes.mating_rate.powf(params.mating_rate_cost)
   + self.genes.stealth.powf(params.stealth_cost)
   + self.genes.detection.powf(params.detection_cost);
let base_metabolic_rate = self.mass.powf(0.75); // Kleiber's Law

let metabolic_rate = (base_metabolic_rate * base_activity_level)
   / params.normaliser;

let diet_modifier = self.get_diet_modifier() as f64;
self.food_consumption = metabolic_rate / diet_modifier.max(0.01);
self.water_consumption = self.food_consumption * 2.0;
```

4.5 Evolutionary Mechanisms Implementation

The evolutionary dynamics emerge from the interplay of implemented mechanics, reflecting the principles of GAs discussed in Chapter 2.2.

The `genotype` is directly represented by the `AnimalGenes` struct containing normalized `f64` values. **Crossover** and **mutation** operators are implemented within the function `extract_gene`. These operators were implemented in the following way:

- *Crossover:* For each gene, a parent is chosen randomly (50/50 chance) using the `rand::Rng::gen_range(0..=1)` method. The chosen parent's gene value forms the base for the offspring's gene. This resembles uniform crossover applied gene-by-gene.
- *Mutation:* After selecting the base gene value, a random float from interval (0.0, 1.0) is generated. If this float is less than the a specified `mutation_prob`, a mutation occurs. A random value within the range defined by `(-mutation_half_range, mutation_half_range)` is added to the gene value. The result is then clamped between 0.0 and 1.0 to ensure gene values remain within the valid normalized range.

As designed, **Fitness** and **Selection** are emergent. Animals compete for limited resources (`plant_matter`, `hydration`, `meat`, `other animals`) on tiles. Those whose genes and resulting characteristics (`food_preference`, `speed`, `detection`, etc.) allow them to acquire sufficient resources to survive are given a chance to reproduce, while failure leads to death by starvation, dehydration or predation. Successful hunting requires appropriate traits (`speed`, `detection`, `size`, `food_preference`). Reproduction requires reaching sufficient `ready_to_mate` status and finding a compatible mate by checking their genetic distance using function (`can_mate_with_animal`). These factors collectively act as selective pressures, favoring individuals whose genetic makeup leads to successful survival and reproduction strategies within the specific simulated environment, without requiring an explicit, global fitness function.

4.6 Godot Frontend Implementation

The Godot application provides the user interface, visualization, initial setup, and overall control flow, interacting with the Rust backend via the defined GDExtension interface. The main scene typically consists of:

- A root node called `World` holding the main controller script.
- A `TileMapLayer` node used for visualizing the terrain.
- A `CanvasLayer` for displaying population statistics.

The main GDScript controller is responsible for the following tasks:

- Instantiates the Rust `Simulation` object: `var rust = SimulationCore.new()`.
- Manages user-configurable parameters and stores them in a Godot object.
- On simulation start:
  - Calls `rust.set_simulation_parameters` to send the configuration to Rust.
  - Calls the map generation logic that uses `FastNoiseLite` from Godot to generate temperature and moisture data, determines biomes, and populates a Godot `Dictionary` mapping `Vector2i` positions to initialized object instances containing static environmental data. Dynamic properties have their values set to default values.
  - Calls `rust.set_map_for_rust`, which copies the World map from Godot into Rust.
  - Calls `rust.spawn_initial_animals` to create a user-defined number of animals with randomly assigned genes.
- During simulation run:
  - Periodically calls `rust.process_turn_for_all_animals()` based on a timer.
  - Periodically calls data retrieval functions from Rust:
    - `rust.get_tile_animal_counts()` to get animal locations/counts that are used to update tile overlay labels.
    - `rust.get_animal_statistics_dict()` to get current statistics. The script updates UI labels with the values from the returned dictionary.
    - Calls `rust.get_all_animal_data()` at specified logging intervals, passing the data to a separate `DataLogger` script, which saves these data snapshots to a CSV file.

The `Tile_Map_Class` script, attached to the `TileMapLayer`, handles setting the visual appearance of tiles based on the biome data received during initialization, or when change is introduced. It also manages the creation and updating of `Label` nodes used to display animal counts per tile, based on data from `get_tile_animal_counts`. The `DataLogger` script receives the detailed animal data array from the main controller and implements the logic to format and append this data to a CSV file for later analysis.

4.7 Testing and Results

This section looks at the data that can be extracted from a simulation run. Graphs were generated using a Python script where the X-axis denominates the `simulation_turn` Godot variable. For comparison there will be 2 simulation runs presented, one that resulted in a stable simulation is discussed in Section 4.7.1, while the second one resulted in an unstable simulation and is discussed in Section 4.7.2. These simulation runs were set in identical worlds and used the same `Simulation_Parameters` except for the parameter `normalizer`. By changing the value of normalizer, it changes the value of an animal's *metabolic_rate* without changing the proportionate costs of gene values. By increasing the value of normalizer, animals end up spending less energy every turn than their identical twins would in a simulation run with a lower value.

4.7.1 Stable Simulation

In this simulation run, the population quickly reached a relatively stable level and maintained it throughout the simulation run, as can be seen in Figure 4.1. In addition, the distribution of `vore_type` is relatively stable, with a clear majority of herbivores, followed by a much smaller group of omnivores, and the smallest group was made up of carnivores. However, this stability might not have lasted were the simulation run longer, since there is a clear trend evident in the evolution of `food_preference` average gene value.

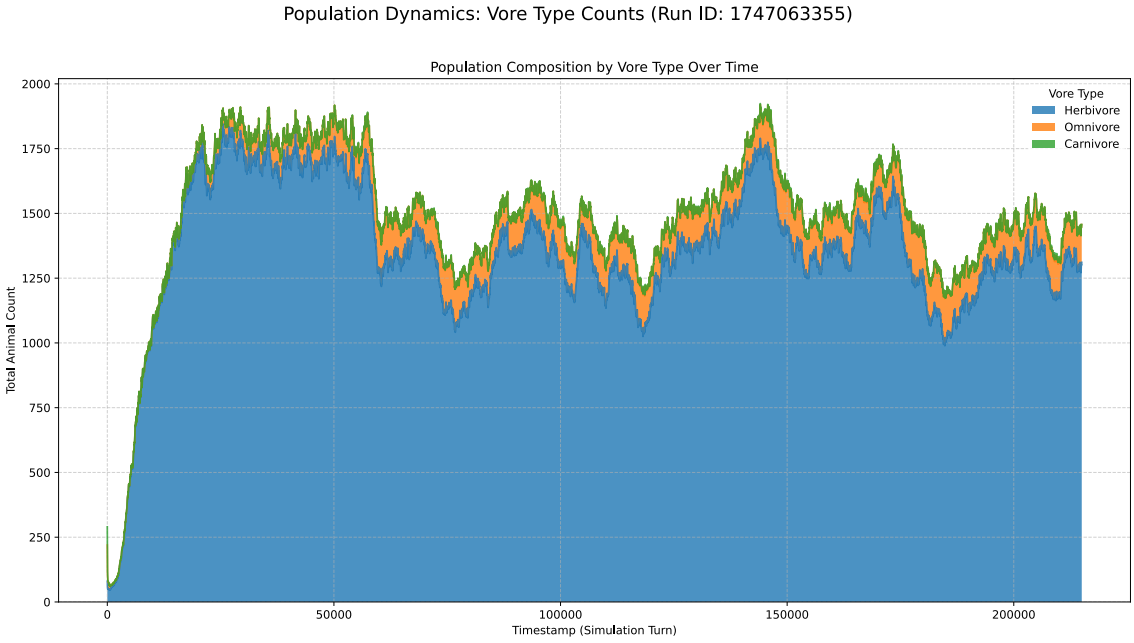

Figure 4.1: Population levels stay relatively constant through the simulation along with a relatively stable distribution of `vore_type`.

As can be seen in Figure 4.2, the evolution of the gene value `food_pref` is stable, and very slowly increasing throughout the duration of the simulation run. This makes sense since the world was initially filled with almost entirely herbivores – since the `carnivorous` strategy made less sense in a sparsely populated world. However, once the animals reached

the carrying capacity of the environment and started competing for resources among themselves, the evolutionary conditions for the rise of omnivore and carnivore populations were set.

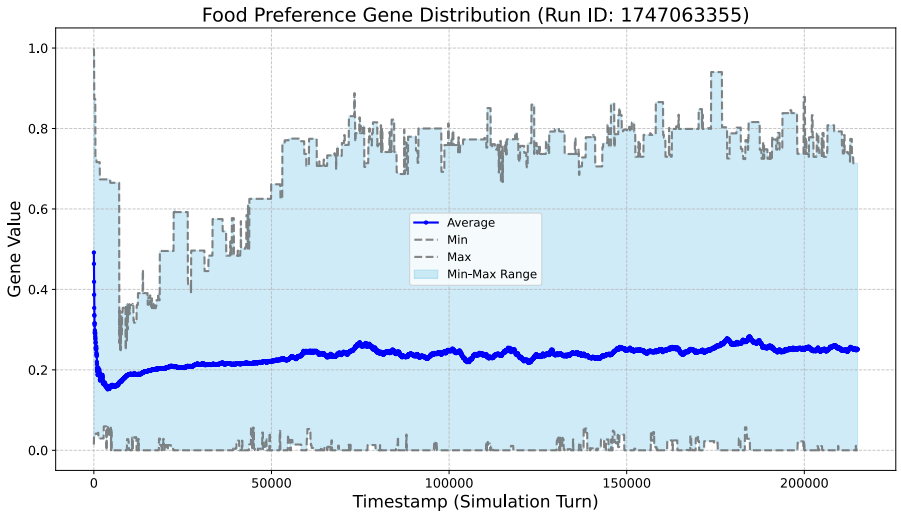

Figure 4.2: Once again the plot seems to be quite stable, though a slight increase in the average value is clearly visible.

4.7.2 Unstable Simulation

This simulation run was made with the `normalizer` parameter set to a large value, which implies that animals had long periods between having to eat or drink, as their resource drain was approximately half of what it was for the stable population discussed in Section 4.7.1. Due to this, the population could keep rising since there was always available food, and large parts of the population had no primary goal to satisfy, so they simply wandered the map and sought opportunities for reproduction. However, this could not go on forever, since at some point the animal population would actually overtake the environments bearing capacity, and the population would collapse. This can be seen happening multiple times throughout the course of the simulation run in Figure 4.3.

The following collapses in the animal population affected *herbivores* the most, because they are only capable of eating `plant_matter`, where an *omnivore* is capable of offsetting the inaccessibility of `plant_matter` with the consumption of meat, and *carnivores* were not negatively affected by the lack of `plant_matter` in their surroundings, while benefiting from the meat that could be found lying around from dead herbivores.

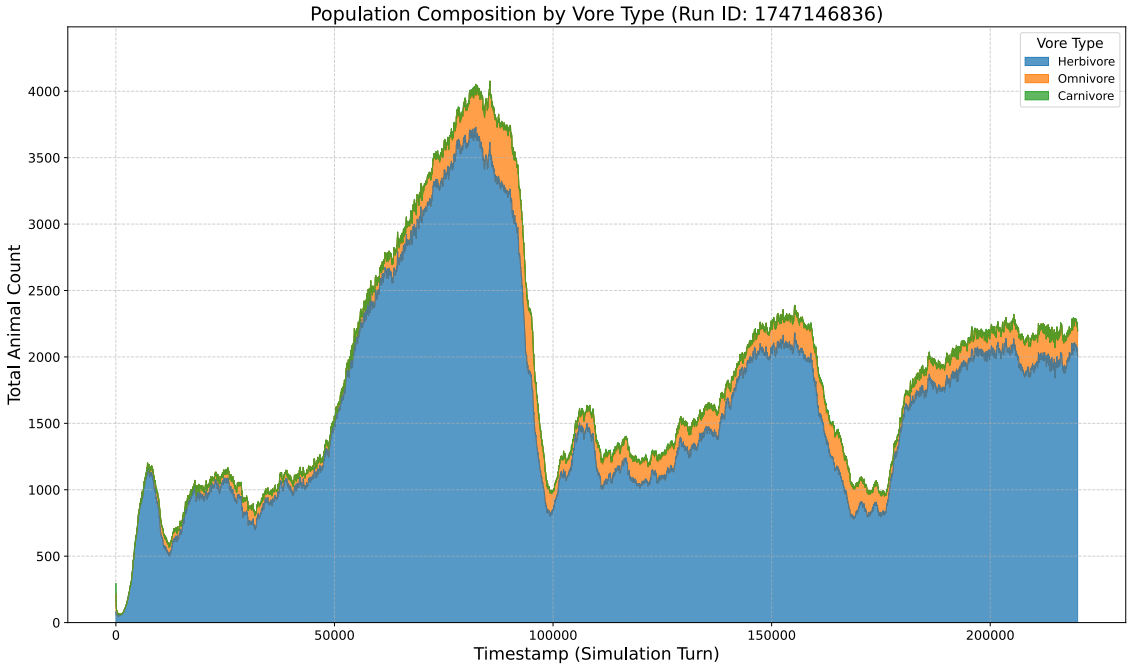

Figure 4.3: The population here seems to be quite unstable, but no real threat of total population collapse seems to have been close at hand, at any point during the simulation run.

Looking at the correlations between the gene values at the end of the simulation run, which can be seen in Figure 4.4, we can clearly see that throughout the simulation run a heterogeneous population existed, with only one clear correlation being visible – that of detection and food_preference, which makes perfect sense if animals were attempting to use stealth to evade predators, but predators were not relying on stealth in order to hunt their prey.

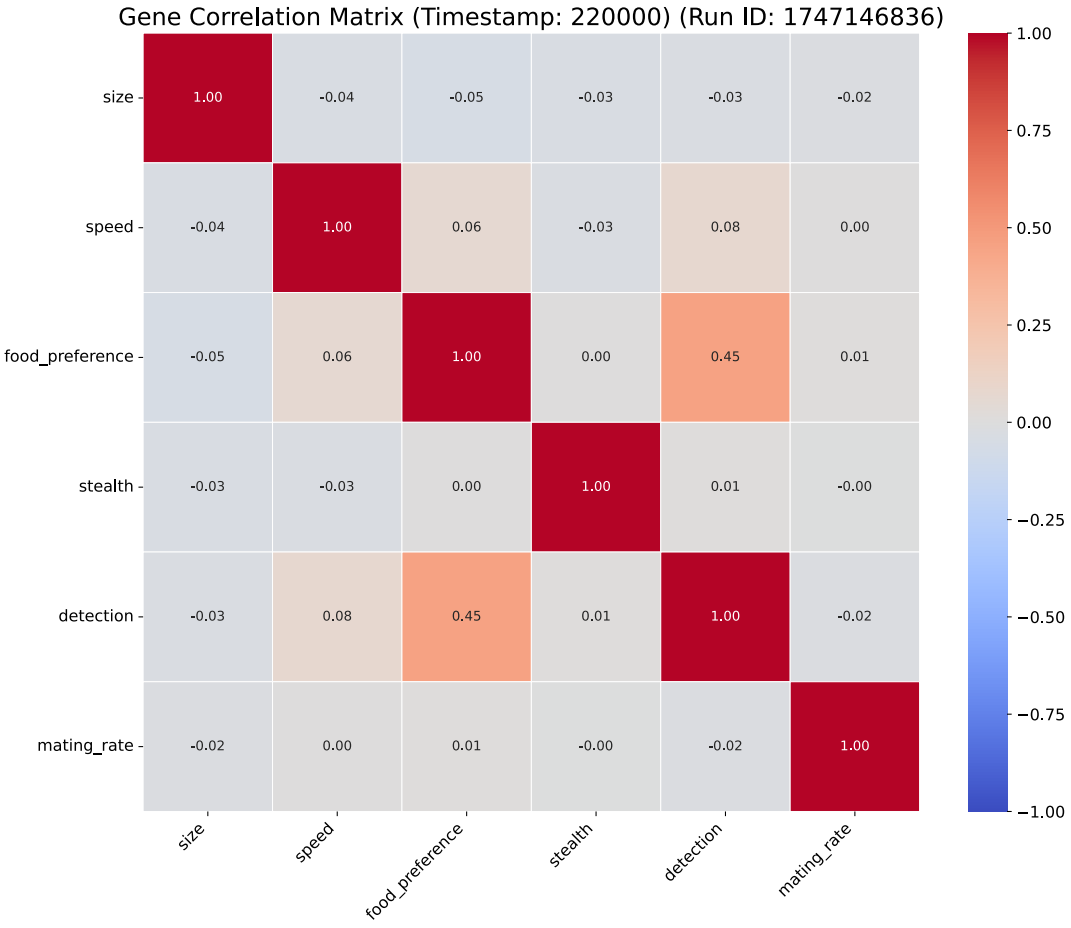

Figure 4.4: Heatmap showing the correlations between each gene pair. A clear correlation exists between the gene's of food_preference and detection, while a slight correlation exists between speed and detection and speed and food_preference.

The evolution of *stealth* can be seen in Figure 4.5. Here we can see that there is some interest in having at least above zero value for the stealth gene, however as can be seen from the heatmap example from Figure 4.4, there is no clear correlation between stealth and other genes.

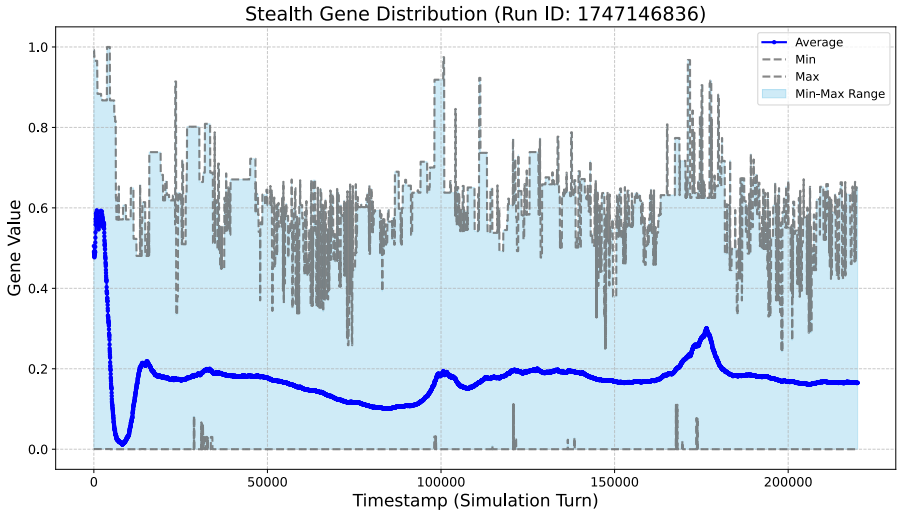

Figure 4.5: Plot showcasing the distribution of the animals stealth gene. After the initial shock, the overall plot seems to be quite stable, though there is a lot of variation present.

The evolution of *detection* is shown in Figure 4.6. Here the average value of detection is very low, which implies that there is no great advantage to be gained by spending resources on having a higher value for this gene when it comes to animals in general. However, from the heatmap example shown in Figure 4.4, we can also notice that there is a clear correlation between detection and food_preference. This implies that the more *carnivorous* an animal's strategy is, the greater the need for a high value of detection exists.

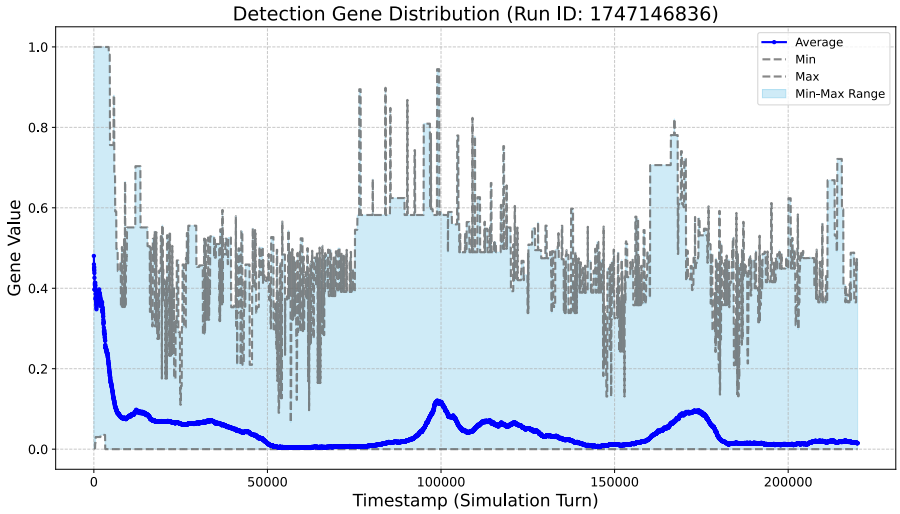

Figure 4.6: Plot showcasing the distribution of the animal's detection gene. The average value quickly descends to almost 0, which implies that the vast majority of the animal population makes no use of this genetic value, while some animal's are highly specialized in this domain.

4.8 Future Work

Although the implemented simulation successfully demonstrates core principles of evolutionary adaptation and provides a performant platform for observation, several avenues for future development could significantly enhance its capabilities, realism, and scope.

Firstly, further performance optimization through parallelism could be explored within the Rust backend. As identified during the design and implementation, the tile update phase (resource replenishment, scent/meat aging) was parallelized using the `rayon` library, yielding performance benefits. However, other phases of the simulation turn also hold potential for concurrency. The animal decision phase, where each animal assesses its state and environment for example, could be parallelized as is, while action execution might also be parallelized on a per-tile basis.

Secondly, the representation of flora and environmental diversity could be greatly expanded. Currently, vegetation is represented abstractly as a single `plant_matter` resource regenerating at a rate influenced by static tile properties. Future iterations could introduce distinct plant species with their own lifecycles, nutritional values, defenses (e.g., toxins requiring specific gene adaptations in herbivores), and dependencies on specific biome conditions. This could lead to more complex food webs and co-evolutionary dynamics. Implementing cyclical resource availability (e.g., seasonal growth patterns) could further increase environmental complexity. Biome diversity itself could also be increased beyond the current types, potentially incorporating features like rivers or impassable terrain, adding new constraints and opportunities for agent adaptation.

Thirdly, significant visual and analytical enhancements are possible. This could involve more distinct graphical representations for animals, perhaps even dynamically generated based on their genes. A more sophisticated approach to visualizing relatedness could be implemented; instead of relying solely on the Euclidean distance for mating compatibility, techniques like K-means clustering could be applied to `AnimalGenes`, that would effectively separate the available gene space into K distinct animal species. The resulting clusters could be visualized as distinct emergent `species` on the map, providing clearer information on population divergence and speciation events.

Finally, incorporating more dynamic environmental events and long-term trends would enhance both the simulation's realism and its utility as an experimental tool. This could include implementing seasonal weather shifts that affect temperature, moisture, and resource regeneration rates; introducing stochastic environmental disasters like droughts, floods, or fires that cause localized population stresses; and simulating slow-moving climatic trends that force gradual adaptation over many generations. These features would enable us to test agent resilience and adaptability under a wider, more challenging range of scenarios, directly controllable by the user to set up specific experiments, thereby enriching the user experience and the scope of potential research questions addressable by the simulation platform.

Addressing these areas would build upon the foundation established by this work, moving towards a more comprehensive and nuanced simulation of ecological and evolutionary dynamics.

Chapter 5

Conclusion

This goal of this thesis was to design and implement a 2D agent-based ecosystem simulation, with the primary objective of observing and analyzing emergent evolutionary dynamics and population changes. The work successfully achieved this goal by developing a hybrid simulation platform utilizing the Godot Engine for user interaction and visualization, and a high-performance Rust backend, integrated via GDExtension, for the core computational logic. This architecture proved effective in balancing rapid development with the computational demands of simulating numerous agents and their genetic evolution over extended periods.

The theoretical groundwork encompassed principles of computer simulations, genetic algorithms, and procedural noise functions, providing a robust foundation for the simulation's mechanics. The design focused on creating a modular system with clearly defined components: genetically distinct animal agents driven by survival and reproductive needs, a spatially heterogeneous environment with dynamic resources, and explicit mechanisms for inheritance and mutation.

The implementation phase translated this design into a functional tool. Key achievements include the successful integration of Rust for core processing, the development of a clear interface for Godot to control and query the simulation, and the implementation of a phased simulation loop that manages complex agent interactions and environmental updates. Data logging capabilities were integrated to capture detailed statistics and individual agent data, facilitating the analysis of simulation outcomes.

Analysis of simulation runs, particularly the comparison between stable and unstable scenarios (as detailed in Section 4.7), demonstrated the system's ability to model plausible ecological and evolutionary phenomena. These included population fluctuations around environmental carrying capacities, and adaptive shifts in average genetic traits in response to selection pressures. The performance benefits of the Rust backend and the parallelization of specific tasks were also evident, allowing for more extensive simulation runs.

This work contributes a flexible and extensible platform for studying artificial life and computational evolution. It showcases a practical approach to combining a user-friendly game engine with a high-performance compiled language for complex simulations.

While the simulation successfully models core dynamics, limitations exist, primarily in the current abstraction level of ecological interactions and agent behaviors. These, along with potential enhancements, are discussed in detail in Section 4.8, outlining paths towards creating an even more comprehensive and nuanced simulation tool.

Bibliography

[1] Barberousse, A.; Franceschelli, S. and Imbert, C. Computer simulations as experiments. *Synthese*, august 2009, vol. 169, p. 557–574.

[2] Gruner, S. Eric Winsberg: Science in the Age of Computer Simulation. *Minds and Machines*, may 2012, vol. 23.

[3] Hynek, J. *Genetické algoritmy a genetické programování*. Grada, 2008. ISBN 8024726953.

[4] Lagae, A.; Lefebvre, S.; Cook, R.; DeRose, T.; Drettakis, G. et al. A Survey of Procedural Noise Functions. *Computer Graphics Forum*, 2010, vol. 29, no. 8, p. 2579–2600. Available at: https://doi.org/10.1111/j.1467-8659.2010.01827.x.

[5] Mitchell, M. *An Introduction to Genetic Algorithms*. The MIT Press, february 1996. ISBN 9780262280013. Available at: https://doi.org/10.7551/mitpress/3927.001.0001.

[6] Oreskes, N.; Shrader Frechette, K. and Belitz, K. Verification, Validation, and Confirmation of Numerical Models in the Earth Sciences. *Science*, 1994, vol. 263, no. 5147, p. 641–646. Available at: https://www.science.org/doi/abs/10.1126/science.263.5147.641.

[7] Peck, S. L. Agent-based Models as Fictive Instantiations of Ecological Processes. *Philosophy, Theory, and Practice in Biology*, 2012. Available at: http://dx.doi.org/10.3998/ptb.6959004.0004.003.

[8] Perlin, K. An image synthesizer. *SIGGRAPH Comput. Graph.* New York, NY, USA: Association for Computing Machinery, july 1985, vol. 19, no. 3, p. 287–296. ISSN 0097-8930. Available at: https://doi.org/10.1145/325165.325247.

[9] Perlin, K. Improving noise. *ACM Trans. Graph.* New York, NY, USA: Association for Computing Machinery, july 2002, vol. 21, no. 3, p. 681–682. ISSN 0730-0301. Available at: https://doi.org/10.1145/566654.566636.

[10] Roman Frigg, J. R. The philosophy of simulation: hot new issues or same old stew? *Synthese*. Springer, 2009, vol. 169, no. 3, p. 593–613.

[11] Thede, S. M. An introduction to genetic algorithms. *Journal of Computing Sciences in Colleges*. Evansville, IN, USA: Consortium for Computing Sciences in Colleges, october 2004, vol. 20, no. 1, p. 115–123. ISSN 1937-4771.

[12] Weisberg, M. *Simulation and Similarity: Using Models to Understand the World.* Oxford University Press, 2013. ISBN 9780199933662. Available at: https://doi.org/10.1093/acprof:oso/9780199933662.001.0001.

[13] Whitley, D. A Genetic Algorithm Tutorial. *Statistics and Computing*, october 1998, vol. 4. Available at: https://doi.org/10.1007/BF00175354.

[14] Worley, S. A cellular texture basis function. In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques.* New York, NY, USA: Association for Computing Machinery, 1996, p. 291–294. SIGGRAPH '96. ISBN 0897917464. Available at: https://doi.org/10.1145/237170.237267.