Simulating Real-World Genetic Methods and Evolution Tactics on Artificial Bodies

Travis Faulkner
School of Mathematics, Computer Science & Engineering

Artificial Intelligence

1. Introduction

1.1 Overview

The goal of any organism is to keep itself alive long enough to replicate its genetic material, and while the tangible results from this process have evolved for billions of years, the techniques used by nature remain largely unchanged. From prokaryotic to eukaryotic replication, genetic mutations that arise in any reproductive process have allowed for the birth of over 8 million unique species[3], each of which having their own genome specialised to increase its odds of survival. Each of these beings, including ourselves, still derive from the first single-celled life forms that lived in earth’s most ancient oceans[4], though over the course of hundreds of millions of years have undergone dramatic changes that have only been possible due to processes such as mutation and genetic duplication. These ‘mistakes’ made by mother-nature force animals to change their way of thinking in order to adapt to their different bodies. If these changes then improve their odds of survival via a better use of their environment, their mutation will quickly become a desirable trait and eventually, given some luck, will become a mainstay in that specie’s genome. This process of evolution is what has led to our diverse ecosystem of life, though it seems seldom thought is put into simulating these complex systems and observing any interesting behaviours or patterns that may arise.

1.2 Aims & Objectives

This project aims to create such a program that will allow for an accurate, though simplified, simulation of how genes and DNA mutate. This will be achieved by creating artificial bodies whose every feature is controlled by its genetic material, and whose genetic material is simulated to follow the same processes that real-world DNA does. These artificial bodies (agents) will be composed from a genome that creates a set of traits and geometric triangles that makeup an agent’s digital being. Its shape will dictate features such as its speed and attack damage, while its traits will impact behaviour, colour, and mating, meaning that every aspect of its existence is based on its genome. These rules apply to every agent, and therefore as a simulation’s population increases, as will entropy, leading to chaotic encounters with neighbours competing for food and mates. An agent’s colour plays a key part in its place within these encounters, as agents who are a similar shade look favourably upon one another, like a clan, while those polarised may try to attack and kill those different to itself. These simple rules based upon nature, when applied to simple cell-like organisms, lead to interesting beings and fascinating exchanges as they attempt to survive the simulation.

2. Literature Review

Real-world genetic makeup and phenomena such as mutation, duplication, and gene flow will be accurately simulated in this project to provide the best look into any patterns that arise or parallels that can be drawn to our own ecosystems. These systems are complex however, and therefore searching for any prior simulation attempts that look more closely at the evolution of genetic algorithms, as well as research that looks directly at evolution/adaptation of world-life creatures over a period of time is paramount to producing an accurate and worthwhile simulation. Outlined are some extracts from relevant academic papers and books, along with how this project will then use/build upon this knowledge to build an accurate and extensive simulation.

2.1 Simulating Agents

The simulation of agents as artificial bodies is a topic that has been tackled before and therefore prior work on this topic is explored below. Specifically concerning certain requirements for these agents, such as being created from a genome and the ability to breed and pass on their genetic material to children.

Breeding

In June 2001 Jefferson's paper titled 'AntFarm: Towards Simulated Evolution'[2] explored the simulation of ant behaviours when foraging for food. It employed the use of a genetic algorithm to simulate the natural evolution of different colonies of ants, specifically looking at any strategies that may arise in the process of transferring food into a central nest. The simulation does offer some interesting insight and elements that are similar to this project's problem, specifically in utilising a male and female system to produce offspring, rather than the more traditional method of having only 1 gender. The paper achieves this by selecting parents based on their 'score' or rating of how well they are doing, and then creates a child genome based on a 'two point reciprocal recombination'. This is where 2 points are selected to cross the two parent genomes, resulting in 2 new genome combinations where one is discarded and the other is given to the child. This method is more accurate to real life than the more common single parent method, however real-world genetic reproduction does this recombination on a chromosome-by-chromosome basis, rather than the entire genome, something that will be simulated within this project. This can simply be achieved by giving my project's agents multiple chromosomes to crossover, with the selection process then being improved further by having a male and female agent needing to be close and in a specific state in order to breed, this then also allows for different areas of the game world to house agents with similar genes, simulating how real-world tribes of animals behave and spread out their genome.

2.2 Real-World Genetic Sequencing

Real-world genetic sequencing is a vast topic containing many interesting research papers and books. Due to the scope of this project, however, only the aspects concerning replication and mutation are explored within this section, specifically relating to Brown's book Genomes 2[6].

Makeup

The genetic strings that the bodies in this simulated are made of will be simply 1s and 0s called base pairs. This binary string will be split into groups of 3 to represent codons, groups of 3 base pairs that makeup a gene[5]. A gene may contain any number of codons, each string of which contains the agent's genetic code that may determine its size, shape, and personality traits. Codons, therefore, need to be decoded to be read by the sequencer, allowing for some codons to have special traits; such as the 'STOP' codon, made of the string '000' that signifies the end of a gene.

Duplication

The method of changing/creating genes (rather than just inheriting them) will likewise be based as much in real-world methods as possible. The book Genomes 2 by T.A. Brown[6] outlines the 2 most common ways of gene acquisition; gene duplication and inter-species breeding, with the duplication being particularly interesting as it can occur in many different ways. The 3 main ones in this simulation being; duplication of the entire genome, duplication of a single/parts of a chromosome, or duplication of single genes. The duplication of the entire genome most commonly occurs during the meiosis process shortly after conception, and occurs where the gametes produced are diploid rather than haploid, meaning they have 2 copies of each chromosome rather than 1. If this occurs and the diploid cell then fuses with a normal haploid cell it can lead to the given organism being sterile[6.1], however if 2 diploid cells fuse then the result will be a type of autopolyploid. This does not lead directly to gene expansion however, as the copy doesn't contain any new genes, though it does make it more likely for mutation to occur due to there being double the chance of something going wrong. This, therefore, may be beneficial to simulate by, rather than changing the genome at this stage, having just the chance of a mutation occurring increase. During this early stage the other two duplication errors may also occur, with both leading directly to mutated genomes. Both are very similar and may occur via an unequal crossing-over of parent chromosomes, or via a process called 'replication slippage'. In which the genes may just mutate randomly, though this is more likely with smaller gene sequences, which may also lead to the rate of mutation changing based on the chromosome/gene it is being applied to.

Mutation

As outlined above, mutation is extremely important in the process of creating new genes, and is also discussed in Genomes 2[6.2]. Here it goes into great length discussing the causes and effects of different types of mutations[fig. 1], and how many mutations that do result in nucleotide sequence changes have no effect on the functioning of the genome, making them useless in this type of simulation. Mutations that occur in the coding regions of genes, however, are much more likely to have a tangible effect and therefore will be the ones that we will be simulating. The main idea behind this phenomenon is just changing, deleting, or adding amino acids (or in our case bits) to the string of the gene; causing either synonymous mutation, where the change doesn't affect the function of the gene (produces the same codon) or non-synonymous mutation, where the change does affect the codon and therefore the function of the gene, to varying degrees. The 4 types of mutations[fig. 1] that will be simulated are as follows: *Nonsense* mutations are mutations where individual or multiple DNA strands mutate and cause a given codon to be converted to a stop codon. This is an instance of non-synonymous mutation due to the gene being shortened to wherever the mutated codon is placed, usually leading to a heavy impact on the organism as it causes a gene to be split into 2. There are also readthrough mutations, where the stop codon is mutated so the gene is extended past its normal length. These types of mutation usually happen as point mutations, where only one bit is mutated, but can also occur where multiple bits mutate at once, though much more rare. There can also be instances of deletion and insertion mutation where bits are removed or added from the genome. Where, if occurring in groups of 3, this will simply add or remove codons, though if not in triples any of the above mentioned mutations may occur, leading to an even more drastic change in the organism's genome. The simulation of these mutations, therefore, will be centred around flipping bits in the agent's chromosomes, at the birth stage, that will cause any of the above mutations to occur. However, choosing a method to decide whether a mutation will occur (mutation-rate) is much trickier due to the wealth of factors that go into whether a mutation will occur or not.

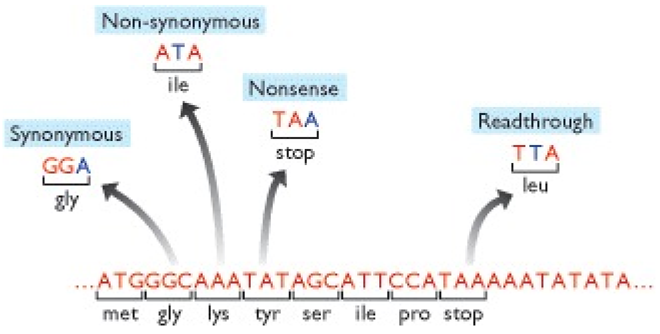

Non-synonymous
ATA
ile
Nonsense
TAA
stop
Synonymous
GGA
gly
Readthrough
TTA
leu
...ATGGGCAAATATAGCATTCCATAAAAATATATA...
met gly lys tyr ser ile pro stop

Fig. 1. Examples of mutation types on a section of DNA

Mutation-Rate

An article on ScienceDirect titled ‘Mutation Rate’[7] states that the mutation rate in humans is estimated to be on the order of 10^{-4} to 10^{-6}. To use this rate would require the simulation to either run at an extreme speed or for users to wait a *very* long time for any noticeable evolution to occur. Because of this it would be beneficial to not simulate this area of the real-world accurately, but rather allow the user to control the mutation rates themselves.

3. Simulation

The simulation engine is based upon tri-engine[10], an open source game engine that has many features to ensure the simulation runs efficiently including (but not limited to): grid-based collision, console/cvar system, events, and geometric shape support. Outlined below are how these features work as well as any other methods of note that the engine provides, along with an overview on how the user may interact with these methods.

3.1 Features

The base engine features a wide range of methods and tools achieved by a multitude of techniques to allow for optimal performance when simulating a large volume of objects. This is achieved by various complex systems and libraries (SDL[8], EventBus[9]).

Console System

A console in any graphical software allows for debug commands to be run in runtime, which proves particularly useful when simulating hundreds of agents across a large game-world. The console system features a small GUI. Commands include:

- *tp/pos* : teleport to a position in the gameworld
- *scale* : set render scale
- *aabb* : toggle debug boxes around game entities

Grid-based Collision

Due to the volume of collision entities at any given time, the traditional method of having a nested for loop for every entity loaded into memory at a computational complexity O(n^2) would be too slow. Therefore a grid-based collision system is used; this works by splitting the entities into abstract grids, then checking each entity on a given grid only to entities in its own or surrounding grids. This reduces the complexity the more numerous and spread out the entities are, which is exactly what the expected behaviour of the agents is.

Events

Defined as "an action or occurrence that can be identified by a program", events allow for different parts of software to share data without having to be explicitly linked. This is achieved by having a data-structure known as an event bus to *post* events, then having listener methods to *listen* for events. Events within this project are primarily used to post game-world changes, such as agent deaths, for (mostly) data recording. This project uses gelldur's EventBus[9] project for these systems.

Spawning Entities

The game system creates *GameObjects* that can be turned into entity containers such as *GeometryObject* and *TextureObject* allowing for spawning of entities such as Agents or Food. This spawning allows for simple and efficient objects to be quickly loaded into memory, while also allowing for information to be passed throughout the codebase via events. This means for example when an Agent is spawned an event will be triggered to tell any listeners that a new agent has been created, allowing for any listeners to be passed the agent's *GeometryObject* to extract data.

Mathematical Methods

Contained in the *maths* header file within the *util* directory are a number of useful methods that allow for increased functionality within the simulation. Some that are of high importance for this paper are

- *ColourDifference* : takes in two colours and returns the 'distance' between them (0-255) [13]
- *StringToDecimal* : uses the *strtoull* method to convert a string (such as 101011) to its decimal counterpart (43), a flag is also included if the output should be using two's complement for negative numbers.

3.2 User IO

How the user interacts with the simulation and how information is represented to give insight to the user.

Inputs

The user has a variety of inputs they can choose to add/customise game objects. For example *mouse1* adds an agent, and pressing a variety of the *J K and L* keys (representing RGB) will change the colour to max out its respective value, so pressing *J* and *Mouse1* will produce a red agent. Furthermore *Mouse2* spawns a food object and *Mouse3* while hovering over an agent will display the agent's name. Pressing *O* and *P* zooms the camera in and out via increasing/decreasing the texture/geometry size.

Console

The most prevalent and constant aspect of real-time data visualisation is the console. This displays[fig. 2] all agent's names, states, hps, and staminas at real-time, allowing for quick analysis of an ongoing simulation. The name of an agent can also be mapped to an object on screen via middle clicking the object to display its name.

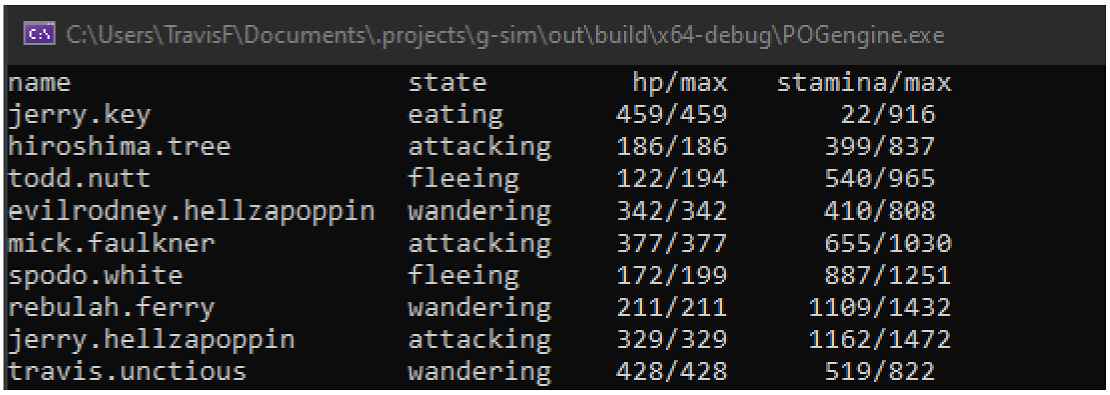

```
C:\Users\TravisF\Documents\.projects\g-sim\out\build\x64-debug\POGengine.exe
name                         state         hp/max   stamina/max
jerry.key                    eating        459/459      22/916
hiroshima.tree               attacking     186/186     399/837
todd.nutt                    fleeing       122/194     540/965
evilrodney.hellzapoppin      wandering     342/342     410/808
mick.faulkner                attacking     377/377     655/1030
spodo.white                  fleeing       172/199     887/1251
rebulah.ferry                wandering     211/211    1109/1432
jerry.hellzapoppin           attacking     329/329    1162/1472
travis.unctious              wandering     428/428     519/822
```

Fig. 2. Console output of g-sim

G-Visualiser

Further data is relayed to a set of files contained within the program that can then be parsed by a tool called g-visualiser. This tool and the data it can process are talked about further within the Analysis section of this paper, though it does include overall simulation statistics as well as information on each individual agent.

4. Methodology

Creating a simulation that follows life's ruleset poses an interesting set of challenges; primarily in converting the real-world methods to the digital simulation while keeping accuracy high. Such challenges and solutions are outlined below.

4.1 Genome Simulation

Genomes, genes, and DNA's behaviours are simulated to be as close to their real-world counterparts as possible, though their makeup in the simulation will be altered slightly as to accommodate for a computer's strengths. The main methods that will be used directly make the most use of basic computer science concepts, the most prevalent of which includes encoding genomes as a string of bits. Utilising these methods will allow for the sequencing of genomes to be fast and reliable, while also being consistent to similar phenomena that occurs in the real-world.

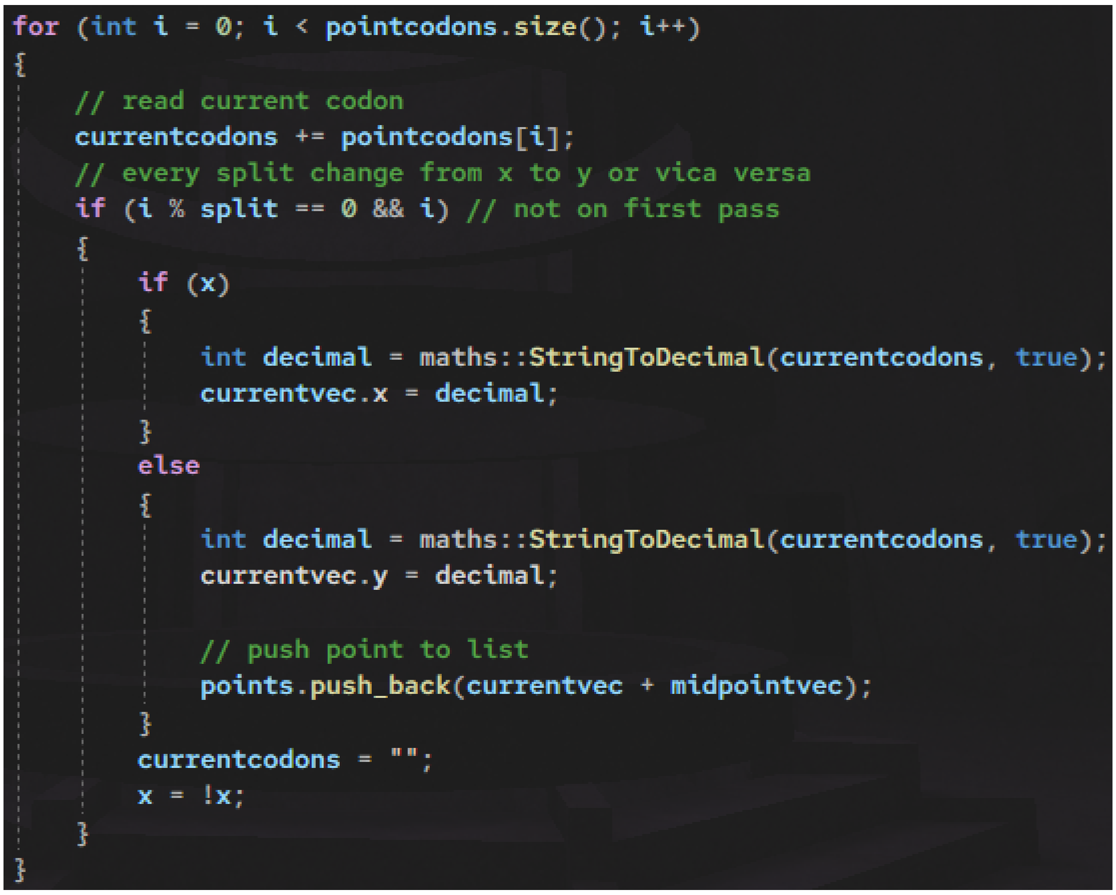
Representation

Genomes are simulated as a string datatype as part of the std library for c++, this allows for easy iteration and concatenation of characters (or codons and dna) that allow for simple, intuitive sequencing of genes. Each gene will be made of codons that are made of triplets of binary data that will then be converted into decimal numbers, such that for example the gene 011101001000 will be split into the codons 011 101 001 000 and then decoded to equal 3, 5, 1, STOP so the end gene's contents are equal to 3, 5, and 1. These numbers can then be used to, say, equal the lengths of the sides of an agent's triangle, or be summed up to equal its anger rating. Every gene will end in a STOP codon that tells the sequencer to stop reading that specific gene's data.

Sequencing

The sequencing of a genome within this simulation is fairly straightforward, with the sequencer only containing a handful of methods. The main goal is to take the *String* representing a genome, break it down into its genes, and break those genes down to their codons to be read directly. The sequencer is also responsible for the generation of genomes, meaning it states in which order the genes will be read (it's only hard-coded aspect).

The main goal for the sequencer was to allow for mutations to be as 'natural' as possible and to take place without any special cases for the sequencer. This requires a lot of flexibility which is achieved by having simple rules to both read and write to a given gene. For example the 2 genes that make up an agent's digital being are its *midpoint* and its *outerpoints*. The *midpoint* is by default a string of 4 codons that are split into 2 equal groups by the sequencer; to create equal sets of codons to be used for its x and y coordinate. The *outerpoints* are calculated by taking a larger string of codons (by default 24) and reading them sequentially[fig. 3].



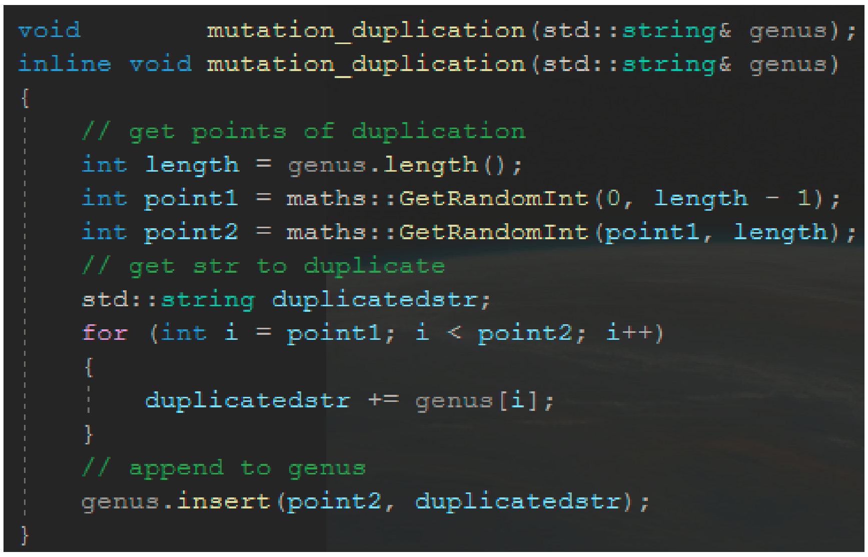
```cpp
for (int i = 0; i < pointcodons.size(); i++)
{
    // read current codon
    currentcodons += pointcodons[i];
    // every split change from x to y or vica versa
    if (i % split == 0 && i) // not on first pass
    {
        if (x)
        {
            int decimal = maths::StringToDecimal(currentcodons, true);
            currentvec.x = decimal;
        }
        else
        {
            int decimal = maths::StringToDecimal(currentcodons, true);
            currentvec.y = decimal;

            // push point to list
            points.push_back(currentvec + midpointvec);
        }
        currentcodons = "";
        x = !x;
    }
}
```

Fig. 3. Code to Sequence Outerpoints of a Agent

Here a variable called *split* controls at what point the value of *currentcodons* is converted to decimal, and then assigned to a coordinate on a point. The coordinate it choses (x or y) is based on a boolean variable *x* that switches based on when the amount of passes equals *split*. Therefore this method may convert any number of codons into a set of valid points.

Mutations

Genomes in this simulation may mutate in 2 ways via point mutations, or duplication mutations. To achieve this the genus is passed through a mutation method before being sequenced that gives a point mutation a 1/10 chance of happening, and gives a duplication mutation[fig. 4] a 1/20 chance to occur. The point mutation method simply takes a random bit of the genome and flips it, while the duplication mutation method takes a random chunk of the entire genome and duplicates it. These methods are as simple as possible in order to accurately simulate real-world examples, though the probabilities of such events occurring are much greater in this case in order to showcase how mutations affect the agents in a timely fashion.



```cpp
void        mutation_duplication(std::string& genus);
inline void mutation_duplication(std::string& genus)
{
    // get points of duplication
    int length = genus.length();
    int point1 = maths::GetRandomInt(0, length - 1);
    int point2 = maths::GetRandomInt(point1, length);
    // get str to duplicate
    std::string duplicatedstr;
    for (int i = point1; i < point2; i++)
    {
        duplicatedstr += genus[i];
    }
    // append to genus
    genus.insert(point2, duplicatedstr);
}
```

Fig. 4. Code that handles Duplication Mutation

Agent Genes

An agent is made of 7 genes: *Midpoint, Triedges, Aggression, Sex,* and *R, G, and B*. These genes, as discussed prior, are made of a different number of codons and each is terminated by a *STOP* codon. The *Midpoint* and *Triedge* genes represent the agent's body via the points that it occupies in 2d space, with the triedges being the outer points and the midpoint being their offset. The *RGB* set of genes are each 8 codons long and work by the average of each codon being taken for that bit in the colour, for example '010' will be equal to a 0.

4.2 Behaviour Simulation

The artificial bodies that make up the genetic algorithm are shaped and controlled by a mix of their genomes and some pre-defined behaviour.

Finite-State Machines

Agents are controlled by a finite-state machine that dictates what the agent's current 'goal' is. The FSM has 5 states: *Wandering, Attacking, Fleeing, Eating, Mating* that relate to one another by the following particular set of rules:

- Agents will default to the *Wandering* state if no special requirements are met
- If 2 agents are close and in each other's sightline while wandering or eating a check will be made if they can mate, if so they will both change state to *Mating*
- If an agent's stamina is below 10% and it is wandering, its new state will be *Eating*.
- If an agent (agent 1) can see and is close to another agent of a different clan (agent 2) an aggression check will be made and agent 1 may move to *Attacking* in an attempt to kill agent 2
- If an agent is being attacked and doesn't fight back it will attempt to escape by changing state to *Fleeing*

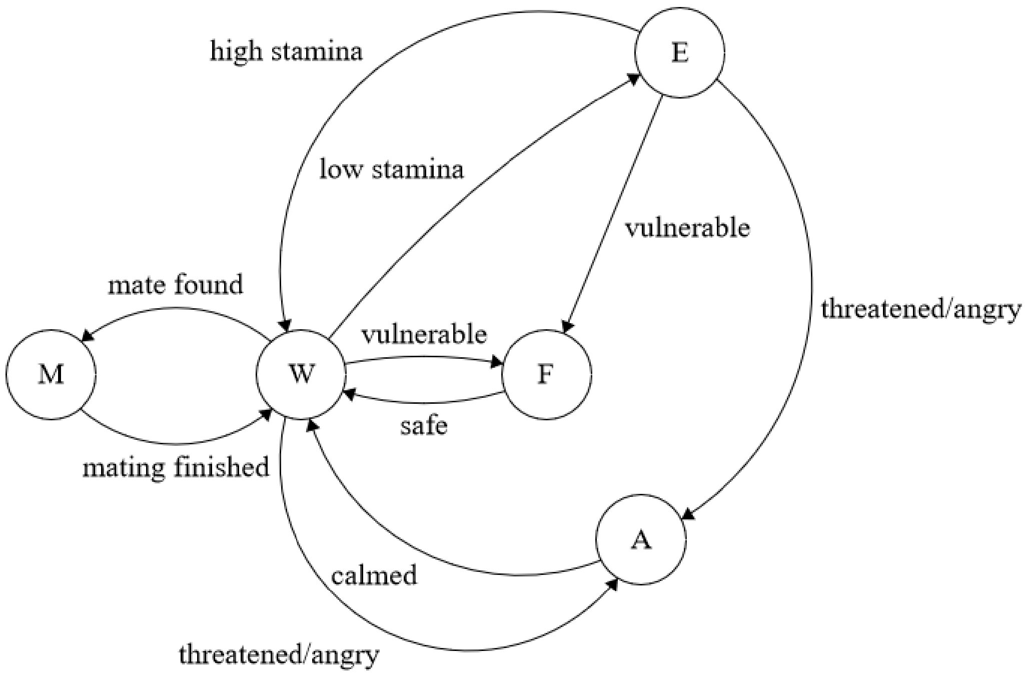


Fig. 5. Diagram of a agent's FSM (simplified)

W = *Wandering* F = *Fleeing* A = *Attacking* E = *Eating* M = *Mating*

Each states logic is controlled by its own method within its class, each method corresponds to the following actions:

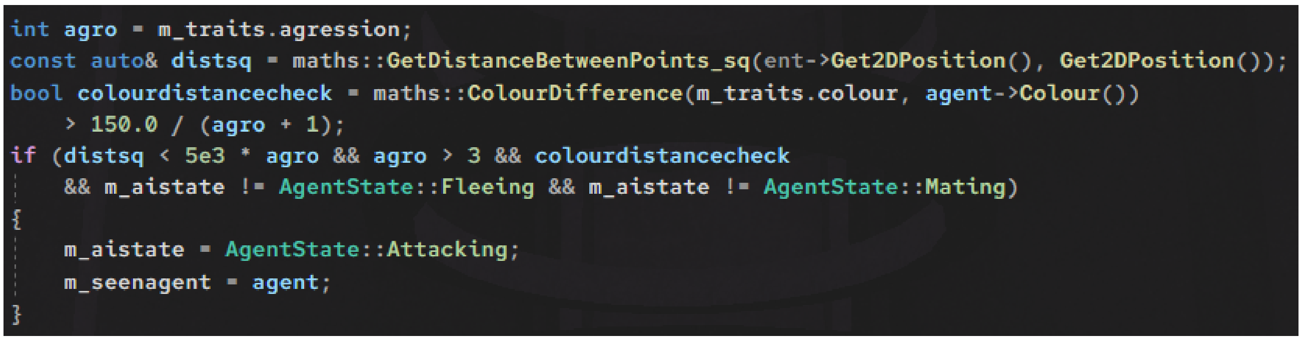
- *Wandering* is where an agent is free to look and move around randomly with no particular goal in mind. This is achieved by randomly selecting a goal point within a 600 unit radius, and then travelling to that point, repeating the process when the agent reaches its goal. This allows for the agent to scope out food, other agents, and mates.
- *Eating* is where an agent's hunger becomes its main priority and it will try to find food to eat. First the agent will check its memory for any food that it may have spotted along its journey, if so returning to the location it last saw that food. If the agent doesn't remember seeing any food or if the food is no longer where the agent remembered, it will wander around the world trying to find some. If the agent reaches food, it will stop to eat.
- *Attacking* is where an agent chases and attacks another agent. The given agent's attack strength is calculated relative to its current status (how healthy it is), its 'spikeyness', and its size compared to the agent it is attacking.
- *Mating* is where an agent and another agent reproduce by mating for between 1000 - 2000 game ticks, producing an offspring. Mating is started when 2 agents of a similar colour seek each other out and get physically close to each other.
- *Fleeing* is where an agent runs away from another agent when threatened or being attacked

Colour

The agent's colour, foremost used to tell the renderer what colour the triangles should be, is also used to determine relationships between other agents. For example an agent that is a specific shade will behave more favourably toward other agents of a similar shade, while being more hostile to agents with a contrasting colour.

Aggression

Contained within an agent's code are a set of traits that control how the agent behaves in the simulation, most of which concern its digital makeup, though its aggression is also a trait given at birth. This trait controls how aggressive the agent is to other agents, including the distance needed to attack another agent[fig. 6].



```
int agro = m_traits.agression;
const auto& distsq = maths::GetDistanceBetweenPoints_sq(ent->Get2DPosition(), Get2DPosition());
bool colourdistancecheck = maths::ColourDifference(m_traits.colour, agent->Colour())
    > 150.0 / (agro + 1);
if (distsq < 5e3 * agro && agro > 3 && colourdistancecheck
    && m_aistate != AgentState::Fleeing && m_aistate != AgentState::Mating)
{
    m_aistate = AgentState::Attacking;
    m_seenagent = agent;
}
```

Fig. 6. Aggression check to change state to *Attack*

Mating

Agents of opposite gender (represented as an integer 0 or 1) can mate to create offspring that contain their parents genetic material. The mating process is controlled by multiple factors including the given agent's colour, such that 2 agents that share similar colours are more likely to start the mating process. When agents see a potential mate their state is set to *Mating* and they will move toward their 'target'. If both agents then collide, and both are ready to mate, the '1' gender agent will be given a *BabyAgent* struct that contains both parent's genomes and the baby's *aliveticks.* After the agent has carried this baby for 3840 ticks (1 minute) the baby will be born. Mating between agents may only occur when agents are above 1200 ticks old.

Navigation

The agents navigate the gameworld in an unpredictable way; their movement is always goal-based, and the path they take is based upon their genomes/random chance. The two goals an agent can navigate towards are a vector *targetpos* and an agent pointer *seenagent*. The *targetpos* variable is set when the agent moves toward a given goal. This can be set, for example, randomly within a radius of 600 when wandering. The agent will navigate to its target via rotating toward its goal and moving toward it, though randomly the agent will veer off course (as controlled by a *Turnobj*).

Food

Searching for and eating food makes up a large portion of an agent's lifespan as failing to do so has mortal consequences. Agent's achieve this via having a section in their memory that stores the location of food, meaning that when agents become hungry (stamina below a fifth of its maximum) they will seek out the last remembered location of food. The food itself is an animated texture object that visually shows how much food is left, with its maximum amount being defined by the simulation and its placement.

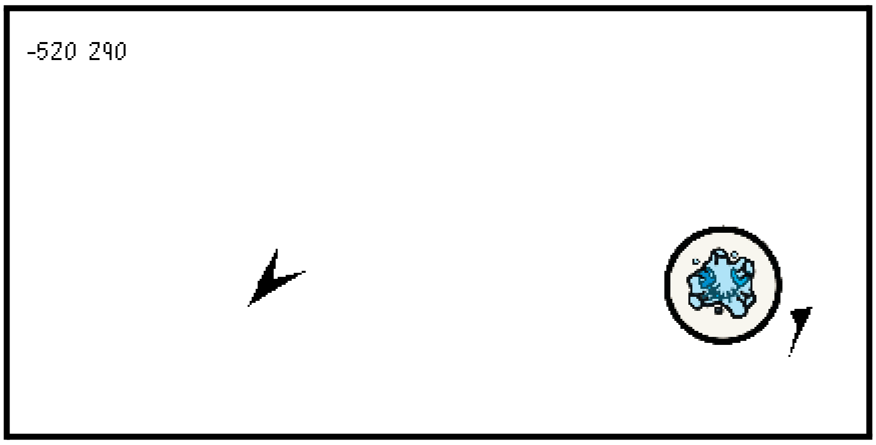


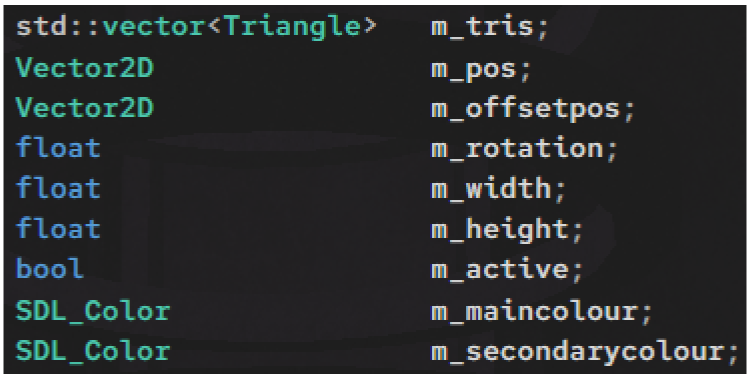
Fig. 7. 2 Black Agents, 1 Eating Food

4.3 Digital Makeup

An agent's makeup is shaped by an algorithm that converts a string of 1s and 0s into triangles that connect together to produce an agent's shape. Agents are therefore given traits that are influenced by both this geometric structure and its genes. The assignment of these traits and how agent's genes affect its makeup, and what this means for an agent in the gameworld, are explored in this section.

Geometry

Agents are made of polygons represented by a set of *Triangle* objects contained within a *Geometry* class[fig. 8]. A *Triangle* consists of 3 points in 2d space, with the 3rd point always corresponding as to where the *Geometry's* midpoint is. This is because the tris are generated in a way where their vertices always align, and they all share their 3rd vertex (midpoint). The *Geometry* class contains a list of these *Triangles* as well as other variables that determine how/where it is rendered in the gameworld.



```
std::vector<Triangle>   m_tris;
Vector2D                m_pos;
Vector2D                m_offsetpos;
float                   m_rotation;
float                   m_width;
float                   m_height;
bool                    m_active;
SDL_Color               m_maincolour;
SDL_Color               m_secondarycolour;
```

Fig. 8. Geometry Class private variables

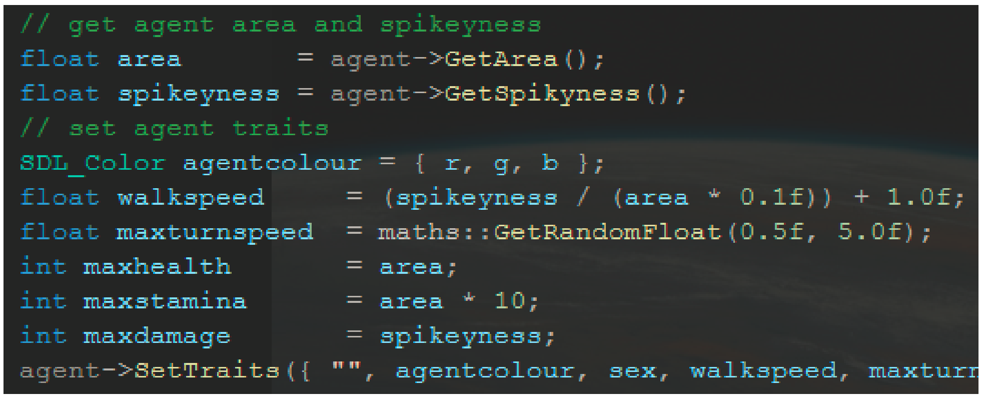
This system allows for a set of random points (with the origin 0, 0) to be generated and assigned to an agent with a position in the gameworld as the geometry object can just be offset when it needs to be rendered.

Spikeyness

'Spikeyness' is a measure of how pointy an agent's shape is and is used for its damage calculations when attacking another agent. It works by measuring the angle of each outer vertex and averaging them out, then when attacking an agent 15 is divided by the spikeyness value, meaning that shapes with a lower average angle will deal more damage. The idea behind this is to make the agent's shape (and hence genome) impactful in their activities as well as their base traits, meaning that if an agent evolves to be more or less spikey it will impact how much damage it may do.

Traits

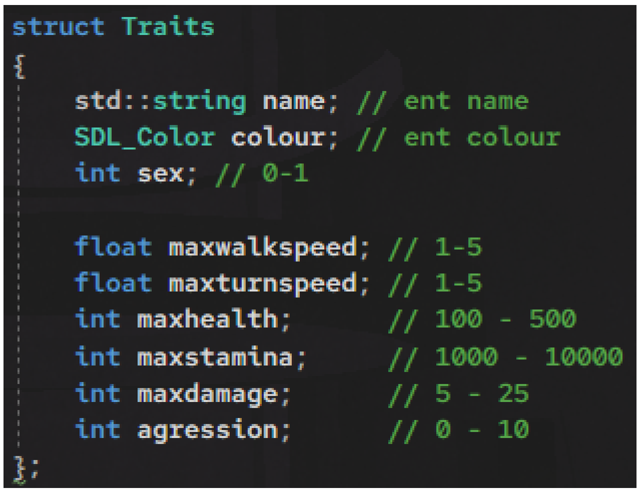
An agent's traits are a set of variables[fig. 9] contained within a struct that makeup an agent's core personal traits (health, aggression, speed etc). These traits are set once at birth based on the agent's genome and random chance, utilising the following methods to best represent its makeup.



```
// get agent area and spikeyness
float area       = agent->GetArea();
float spikeyness = agent->GetSpikyness();
// set agent traits
SDL_Color agentcolour = { r, g, b };
float walkspeed      = (spikeyness / (area * 0.1f)) + 1.0f;
float maxturnspeed   = maths::GetRandomFloat(0.5f, 5.0f);
int maxhealth        = area;
int maxstamina       = area * 10;
int maxdamage        = spikeyness;
agent->SetTraits({ "", agentcolour, sex, walkspeed, maxturn
```

Fig. 9. an Agent's Traits being Set in God.h

These traits are used by the agent to describe its behaviour; such as move speed, damage, and health[fig. 10].



```
struct Traits
{
    std::string name; // ent name
    SDL_Color colour; // ent colour
    int sex; // 0-1

    float maxwalkspeed; // 1-5
    float maxturnspeed; // 1-5
    int maxhealth;      // 100 - 500
    int maxstamina;     // 1000 - 10000
    int maxdamage;      // 5 - 25
    int agression;      // 0 - 10
};
```

Fig. 10. Agent's Traits struct

5. Data & Analysis

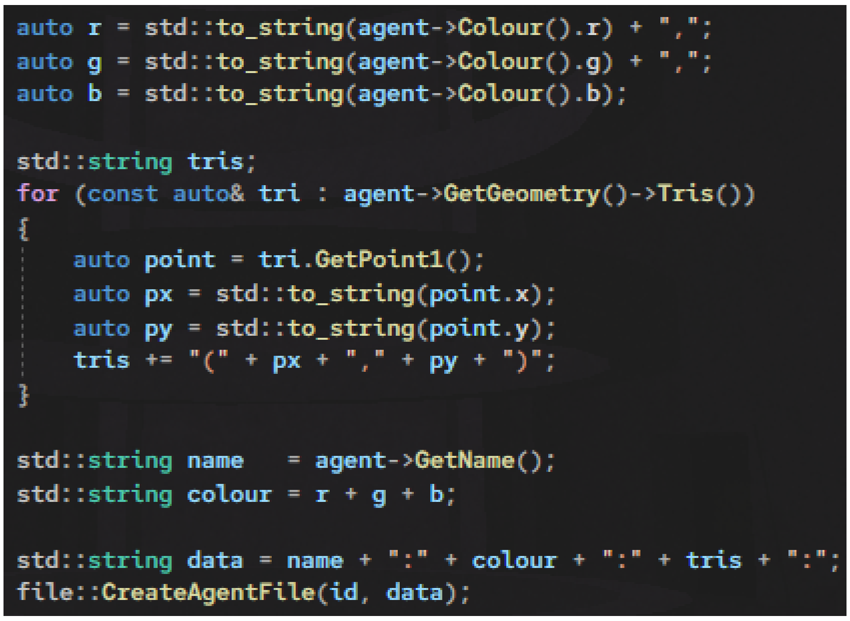
While the simulation offers a real-time 2d interface, this doesn't include all the data known about the simulation (individual agent lifespan, population etc). Therefore data analysis will include the methods of obtaining the data from the simulation, the visualisation of this data, and the information that can be inferred.

5.1 Retrieval

Data is retrieved from the simulation via the parsing *.g* files that are then read by an external program. These files either contain simulation information or agent profiles, both being linked via a common unique agent id.

Format

Within the *AI* namespace of the program there is a section of variables and methods that keep data about the simulation, primarily via listening for the *AgentSpawn*[fig. 11] and *AgentDeath* events. Therefore information is parsed via the 2 main listeners such that every time an agent spawns or dies a main *Logs* file is appended with simulation information (event and time) and an *Agent* file is created to store agent information (name, colour, shape etc). This agent file is named the unique agent *Id*. Further data is parsed upon an agent's death, this updates the log file with the respective event and time, and the unique agent file with death time, children, damage done etc.



```cpp
auto r = std::to_string(agent->Colour().r) + ",";
auto g = std::to_string(agent->Colour().g) + ",";
auto b = std::to_string(agent->Colour().b);

std::string tris;
for (const auto& tri : agent->GetGeometry()->Tris())
{
    auto point = tri.GetPoint1();
    auto px = std::to_string(point.x);
    auto py = std::to_string(point.y);
    tris += "(" + px + "," + py + ")";
}

std::string name   = agent->GetName();
std::string colour = r + g + b;

std::string data = name + ":" + colour + ":" + tris + ":";
file::CreateAgentFile(id, data);
```

Fig. 11. Section of the AgentSpawn Method in ai.cpp

File System

The file system used for saving and loading simulation data are .g files. These files foremost contain a header dictating if the file type is *Logs* or *Agent* then all relevant data is stored using newlines and *:* as separators. The files are saved within the */logs/* directory contained within a simulation folder named the time the simulation starts. The structure consists of a root file *logs.g* and a folder *agents/* containing a file for each individual agent.

5.2 Visualisation

Visualisation of the data is performed by a tool called g-visualiser[11], written in python and powered by tkinter and matplotlib.

Parsing

Data is read from .g files first by popping the header, allowing for the data to be correctly parsed by the relevant methods. Log files are parsed line-by-line with each line containing a significant simulation event (*AgentSpawn AgentDeath*) and the time it occurred. This allows for the creation of graphics that relate to time, for example a population/time graph[fig. 12].

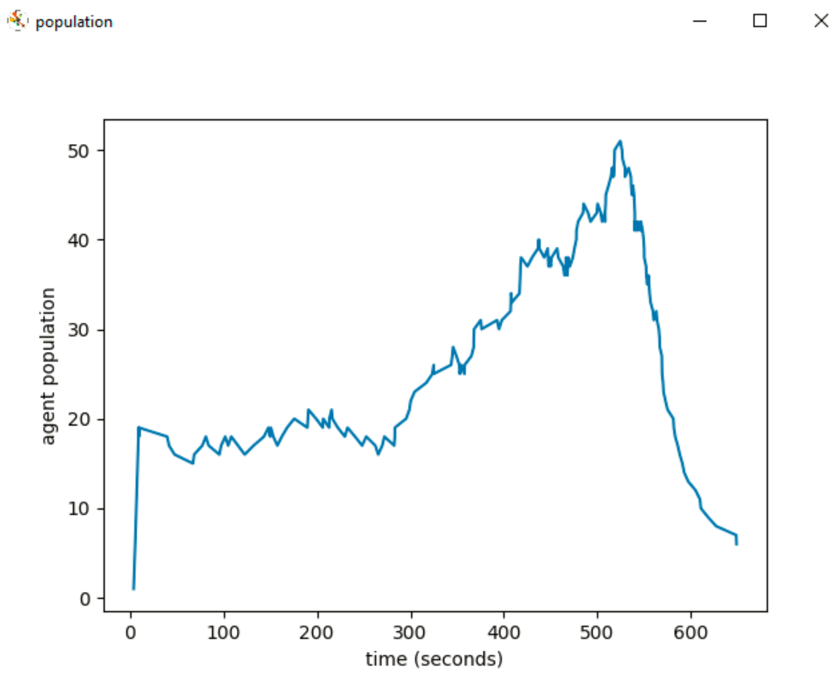


Fig. 12. An Example of a Population/Time Graph

Agent files store information on a specific agent that allows for analysis of the agent's lifespan as well as how it appears visually[fig. 13].

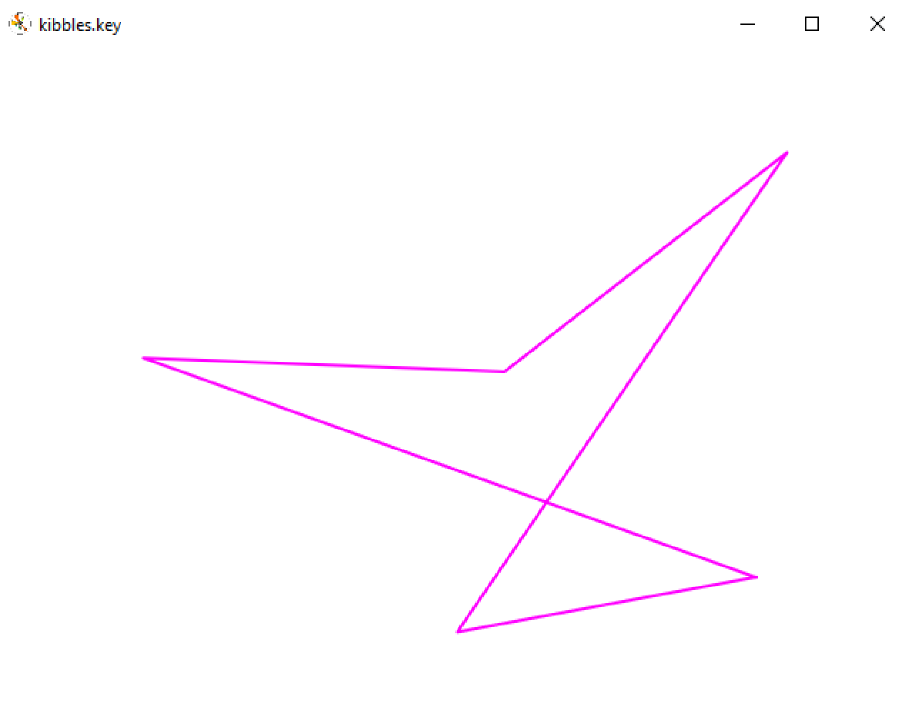


Fig. 13. Visualisation of an Agent

5.3 Findings

The above data visualisation methods describe some very interesting behaviours and outcomes depending on the simulation starting parameters. Outlined are some interesting situations that provided particularly interesting data.

One Clan

The most basic setup tested was spawning 20 agents with the same black colour (0, 0, 0) in the middle of the map along with a single, large foodsource. This led to a meandering portion of time where the population stagnated as the agent's reached sexual maturity, however once this was achieved the population spiked at a quick, consistent rate. This was until the food source ran out, leading to the death of the population. This simulation didn't lead to any notable mutations, though there was an early birth of a more grey agent, though this proved a similar enough shade as to not upset any aggressive agents.

Two Clans

Another more basic setup involves positioning 10 agents of opposing colours 2000 units apart from each other, each with their own large food supply. Compared to the 'One Clan' setup, this allowed for the possibility of interbreeding and fighting! As shown by the population graph[fig. 14], the blue coloured agents get off to an early lead in terms of population, easily having over double that of the red agent's by 300 seconds in. Though red seems to start showing signs of growth after a lull period, up until blue's food supply runs out, which proves catastrophic for their genome. It then seems, around the 800 second mark, the remaining blues may have migrated to the red food, though they are quickly driven out by the reds, who then have a slight spike until they meet a similar fate. Once again this simulation test is more to see how populations interact, and therefore there are no large mutation events, though, there do appear to be new shades of blue (around 500 seconds in) and red (800 seconds) that emerge, though these agents don't appear to pass on their genome.

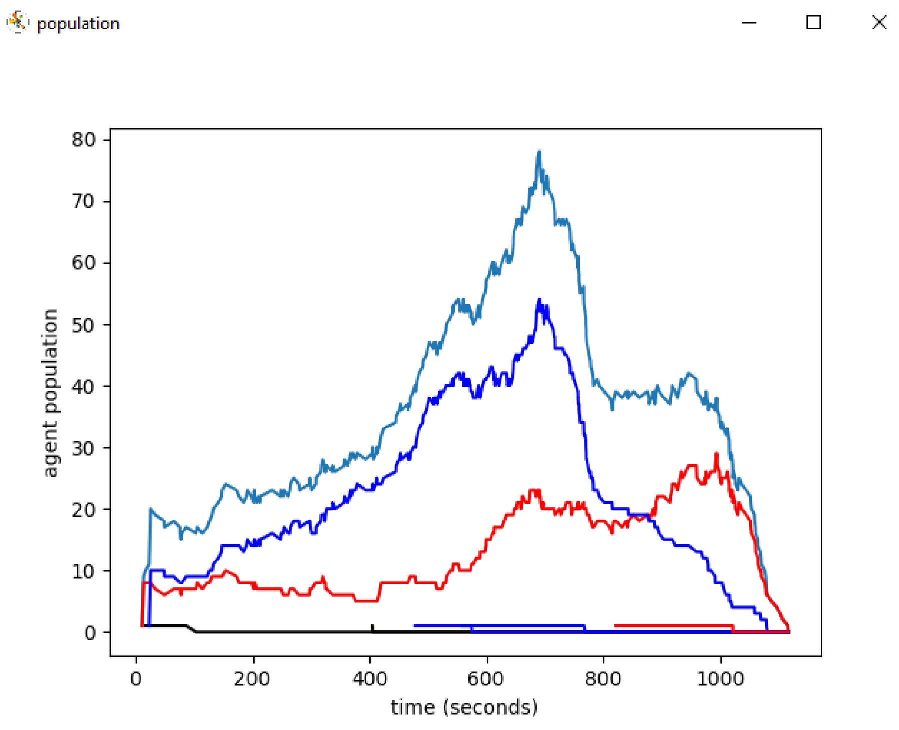


Fig. 14. Population/Time Graph Showing the Population Colours

Complex Testing

More complex simulations were run in an attempt to get an 'evolution event' to occur, in which a new gene that is introduced becomes dominant, though this wasn't achieved within the limited time-frame. An event such as this is possible, however, and given enough time and the right circumstances it will occur, though there are elements that could be added to the simulation (talked about in the next chapter) that may increase the odds.

6. Conclusion

This project has achieved its goal of being able to produce interesting simulations of agents that are able to breed and create offspring that are then able to further mutate, as to observe how genomes evolve. Though there are a number of key observations that showcase how the simulation may be further improved.

6.1 Key Observations

Observations made after running 100s of simulations that offer insight into agent mannerisms and how to best improve the simulation.

Agents

Foremost there seems to be a META strategy emerging in terms of agent build that is the most successful when it comes to the vast majority of simulations, which is to be as average as possible. This is due to agents that are too small and fast tend to quickly explore too far, and due to their small size don't have enough stamina to return to their food. Contrastingly, agents that are too big are too slow to move toward a mate, and therefore don't breed as often as those who are quicker. Furthermore it seems that those agents who do find a mate have children are far more likely to produce offspring themselves compared to older agents, this often leads to some amount of inbreeding and the leading genome dominating all further agents.

Mutations

Mutations within the simulation tend to only offer small changes, much like those found in the real world. Furthermore due to the short simulation time and rarity of meaningful mutations no large evolution event has been recorded yet, though theoretically this should be possible. Further changes that can be made to the program to increase the odds of an event such as this occurring are explored more below, though the smaller mutations do offer insight into what these events may be, specifically related to the colour of a clan changing over time. This is because agents that have been recorded to have slight alterations to their colour aren't targeted due to still being of a similar shade to the large clan, and therefore may breed further to, over a large period of time, create a new family of a different colour.

6.2 Further Work

The engine[10] was worked on alongside the simulation[12], therefore some aspects of the software are tunnel visioned toward a specific goal, rather than being more general. Due to this many aspects of the software could be expanded or improved upon for a better user experience, as well as to accommodate the further simulation of more complex genomes.

Genomes

Admittedly the genomes are more simple than originally planned, as chromosomes were initially meant to be implemented (specifically some variation on the x and y sex chromosomes) though this was never completed. Therefore an update to the sequencer handles genes from different chromosomes, as well as more genes in general, may prove to further the simulation's accuracy to real life.

Environments

An idea explored early in development was the implementation of biomes to simulate different environments that would have specific effects on agents, for example a cold biome would favour larger agents as they would be able to keep themselves ‘warm’. This is an idea that would benefit the current state of the program significantly as it would encourage diversity.

Camera

The camera system went through a large set of changes and tests, with the final system being very rudimentary and not including a lot of functionality that was planned at the start. For example the zoom system does not feature true zoom ins/outs on specific points of the map, something that could offer a more extensive look at the simulation plane. This would require a large rewrite of the current system, though some elements are already included, including the handling of multiple switchable camera systems and dynamic zooming via rendering interpolation.

User Input

The user has a few tools at their disposal for creating gameobjects, though once again a more extensive system was planned that would allow for the creation and customisation of entire agents. This would allow for the user to introduce new populations that aren’t limited to a pre-set group of colours and random traits. This would require a GUI of its own, partly why this was never implemented.

6.3 Conclusions

This project offers a set of tools and interfaces to simulate artificial bodies that contain basic structures that mutate over time. These mutations may affect an agent in ways that diversifies its methods of survival and the average genome of the simulation, furthermore these mutations may compound to form generations that are completely different to their distant relatives. However there is room for expansion and improvement on this base, particularly regarding methods that may increase the biodiversity of a simulation, allowing for a wider range of specialised agents to form, as well as further expansion on the software that both facilitates and visualises the simulation.

Acknowledgments

I owe thanks foremost to all whose prior work has helped me achieve this project; specifically Terry Brown for offering to answer questions about his work, and the authors of open source libraries I have used in my code. Special thanks to Niel Buckley for proofreading multiple drafts and steering me in the right direction, and Leigh Travis for approving and supervising this paper.

References

[1] Foster, P.L. (1993) Adaptive mutation: THE USES OF ADVERSITY, Annual review of microbiology. Available at: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2989722/ (Accessed: 09 May 2023).

[2] Jefferson, D. (2001) (PDF) Antfarm: Towards simulated evolution - researchgate. Available at: https://www.researchgate.net/publication/2551611_AntFarm_Towards_Simulated_Evolution (Accessed: 09 May 2023).

[3] How many species on earth? About 8.7 million, new estimate says (2011) ScienceDaily. Available at: https://www.sciencedaily.com/releases/2011/08/110823180459.htm (Accessed: 09 May 2023).

[4] Berthold, E. (2023) The origins of life on Earth, Curious. Available at: https://www.science.org.au/curious/space-time/origins-life-earth (Accessed: 09 May 2023).

[5] Niu, C.-H. et al. (2003) The code within the codons, Biosystems. Available at: https://www.sciencedirect.com/science/article/abs/pii/0303264789900592 (Accessed: 09 May 2023).

[6] Brown, T.A. (2002) Genomes 2. New York: Wiley. Available at: https://www.ncbi.nlm.nih.gov/books/NBK21128 (Accessed: January 30, 2023).

[6.1] Brown, T.A. (2002) “Chapter 15.2 - Acquisition Of New Genes,” in Genomes 2. New York: Wiley, pp. 465–475.

[6.2] Brown, T.A. (2002) “Chapter 14.1 - Mutations,” in Genomes 2. New York: Wiley, pp. 418–433.

[7] Mutation rate (2014) Mutation Rate - an overview | ScienceDirect Topics. Available at: https://www.sciencedirect.com/topics/medicine-and-dentistry/mutation-rate (Accessed: January 30, 2023).

[8] SDL (1998) Simple DirectMedia Layer - Homepage. Available at: https://www.libsdl.org/ (Accessed: 09 May 2023).

[9] Gelldur (2017) Gelldur/EventBus: A Lightweight and very fast event bus / event framework for C++17, GitHub. Available at: https://github.com/gelldur/EventBus (Accessed: 09 May 2023).

[10] Faulkner, T. (2022) TRVSF/tri-engine: game engine, GitHub. Available at: https://github.com/TrvsF/tri-engine (Accessed: 11 May 2023).

[11] Faulkner, T. (2023) TRVSF/G-Visualiser: For visualising data from G-Sim Simulations, GitHub. Available at: https://github.com/TrvsF/g-visualiser (Accessed: 11 May 2023).

[12] Faulkner, T. (2023) TRVSF/G-SIM: simulating real-world evolution using triangles, GitHub. Available at: https://github.com/TrvsF/g-sim (Accessed: 11 May 2023).

[13] Riemersma, T. (2012) Colour metric. Available at: https://www.compuphase.com/cmetric.htm (Accessed: 11 May 2023).