Implementation of Evolutionary Algorithm for Optimising the Generation of Agents to Navigate Randomly Generated Terrain

ABSTRACT

This paper outlines the design and implementation of an evolutionary algorithm as an optimisation technique for constructing virtual agents to navigate a selection of environments. The final product is called EvoSim and can be regarded as population-based stochastic generate-and-test algorithm. It utilises the physics engine Box2D to construct a world where the evolutionary algorithm can be tested on a simulated population of agents. The application was used to generate a selection of agents that satisfied the objective function, each with unique morphologies and strategies. Multiple experiments are conducted on the simulation to test the effects of population size or culling rate on the resulting solutions. The experiments found some interesting results with the overall project considered a success.

ACKNOWLEDGEMENTS

I would like to thank Dr James Anderson for his assistance during the development of this project.

TABLE OF CONTENSE

1 CONTENTS

1.0 INTRODUCTION

Introduction to evolutionary approaches

Many modern inventions have been inspired by the natural designs produced by evolution. Nature provides us with explicit examples of solutions that practically work in the real world which can then be adapted to solve human-centric problems. Evolutionary algorithms take this a step further, by not only borrowing ideas produced from evolution, but taking the optimisation framework used to create successful living organisms.

The history of evolutionary algorithms

The application of evolutionary principles for automated problem solving originated in 1950s and 60s. Some of the original names included, Lawrence J. Fogel, John Henry Holland, Ingo Rechenberg and Hans-Paul Schwefel who each independently formulated their ideas of evolutionary computing. Early work on simulations of evolution using evolutionary algorithms and artificial life was conducted by Nils Aall in the 1960s and further improved by Alex Fraser who made process is areas such as artificial selection. Due to Ingo's contributions to evolutionary algorithms, they became a widely recognised optimisation method, which he himself applied to complex engineering problems. Academic interest in evolutionary approaches has grown through the decades as computational improvements has allowed this more exhaustive technique to be fully utilised.

Innovative technology using evolutionary algorithms

Evolutionary strategies have found new use in many areas of computer science, one of which is machine learning, with researchers such as OpenAI using it as an alternative to reinforcement learning in ATARI games [1] and DeepMind as a mechanism for producing flexible behaviours in simulated environments such as walking, running and jumping [2].

Report overview

The main sections of the report will be the problem articulation, literature review, design, implementation, testing, results, conclusions and future work.

The paper starts by introducing the reader to the problem articulation which is used to outline the problem statement, define who and what the role of the stakeholders are, review the technical specification outlined in the PID and discuss the motivation and inspiration behind the project.

The next section is the literature review, which looks at a brief overview of the relevant aspects of Karl Sims' paper 'Evolving virtual creatures' which was used frequently throughout the development. The Box2D manual and the book titled, 'Introduction to game physics with Box2D', which were heavily used for their implementation approach, is discussed. Only the most pertinent literature was selected to be in the document and contains information on how this literature applies to the project.

The following section is the Design. The design section discusses how the project should be implemented, using design techniques such as UML class diagrams for planning the construction of the program. It will also outline how the world will be created along with a potential design for the construction of agents and reviewing potential evolutionary algorithm approaches.

The implementation section outlines the construction of the backend library Odingine, which is used as a platform to build EvoSim and discusses how the rendering of the simulation is performed. It also reviews the implementation details of constructing the world, the agents, the environment and the evolutionary algorithm used to optimise the agents.

The next section is the testing section, the role of the testing section is to validate and verify the project solution. This section will contain a selection of unit tests used to constantly check the functioning of the program. Performance testing is used to identify the edges cases of the program and stress test the program. Finally, acceptance testing is performed, which is used to verify the acceptance criteria.

The results section will present and discuss the findings of the paper. This includes results concerning the morphology of the agents produced, changing the fitness metric, the effects of population size, the effects of culling size and how well the solutions generalised.

The penultimate section discusses the social, legal, ethical and health issues related to the project. A number of risks are identified with potential solutions suggested as a way to mitigate the effects.

The final section is the conclusions and future work. This section will summarise the entire project and give an overview of the results section. The future work discussed will investigate potential improvement to the project such as ANNs, new selection mechanisms and a method for parallelising the algorithm.

2.0 PROBLEM ARTICULATION

Problem statement

The problem statement outlines the project objectives. These objectives are the desired outcome of the project for the author and act as a requirement list for the project.

The first primary objective is to construct a backend library for the base of the project. Another primary objective is to create/implement a suitably realistic physics simulation that is capable of simulating agents in an environment. This environment will also need to be visualisable. A third primary objective of the project is to implement an evolutionary algorithm that will be used to solve difficult to define problems via optimisation. A further primary objective is to evaluate some well-known mechanisms of evolution. A secondary objective is to generate agents to traverse randomly generated terrain.

Stakeholders

Developer – Alex Small

The responsibility of the developer is to develop a solution that satisfies the objectives outlined in section 2.0 and 4.0. Due to the author being the only developer, it is vital to ensure consistent effort on the project is maintained while making sure all deliverables are on time and of a suitable standard. Sufficient time must be allocated between the program design and documentation and will loosely follow the Gantt chart specified in the PID document.

Project supervisor – Dr James Anderson

The responsibility of the project supervisor is to track the development of the author's project and discuss issues, potential solutions and give feedback in weekly meetings arranged by the supervisor. The project supervisor will offer advice if needed via additional meetings or email responses and will ultimate be responsible for ensuring the objectives specified are met to a sufficient standard.

User – Anyone

The role of the user is to use and experiment with the application. The objective is for the project is to have the user enjoy the product. Users may require some additional knowledge about evolutionary algorithms to understand the details of the program.

Technical specification

The technical specification will outline the technical objectives which will be used throughout the development process to ensure the implementation stays in line with the original objectives discussed in the PID. This technical specification is a slightly alter version of the one seen in the PID.

Platform requirements

- The program is specified to be accessible to both windows and Linux platforms as long as they conform to the other system requirements.

Physics

- A functional physics simulator in which agents will operate.
- The physics engine will be relatively simplistic containing the most essential elements of a physics engine in order to simulate the required conditions to implement the genetic algorithms. This will include but is not limited to. Rigid body collisions, movement using vectors, air/water resistance, rotation of rigid bodies.

Rendering

- Graphics will be used to display the physics simulation. An appropriate rendering mechanism should be used to create a rendering engine that is distinct from the physics engine.
- The rendering engine will use OpenGl and GLSL to create shader programs.
- Agents will be represented with boxes
- The environment will be represented with lines

Agents

- A number of interesting agent morphologies will be produced with their ANN and 'strategy' for solving the presented problems.
- Agents will need to be constructed of multiple independent ridged bodies (components) that are connected.

Evolutionary Algorithm

- The main objective of the evolutionary algorithm will be the emergent behaviour mode which creates an initial population of agents and places them in the physics simulation. The simulation is then run for many generations to see the how the agents adapt to their surroundings.
- A secondary objective for the evolutionary algorithm will be to make agents that solve novel problems such as movement, swimming or jumping.
- Statistical data will be recorded during the simulation to track aspects such as, speciation, the fitness of each agent, the average fitness of the group over time. From this data graphs and statistical models will be constructed to show the data is an informative way.

Input/Output management

- Create an easy to use interface so users can change parameters amount the evolutionary simulation

Maths libraries

- Linear algebra will be a necessity in the project as vectors and matrices will be used in both the physics and rendering engine. Numerous libraries are available which will be utilised to save time.

Project motivation

The primary inspiration behind this project originally comes from the paper, 'Evolving Virtual Creatures' by Karl Sims. The paper written in 1999 and displays some very impressive results. The author has always been fascinated by nature and evolution their entire life and found Karl's paper to be exactly the interesting project that combines both computer science and evolution. While evolutionary strategies have been around for a long time, Sims' implementation is very interesting as it attempts to solve novel problems such as walking.

3.0 LITERATURE REVIEW

The literature review will outline relevant work and material vital for the design and implementation of the evolutionary simulation. All literature has been filtered to ensure that only the most pertinent information is included. Below are a number of very useful books that were used throughout the entire development process but do not feature within the literature review,

Understanding physics engines – Ian Millington

Introduction to Evolutionary Algorithms – Xinjie Yu, Mitsuo Gen

Artificial Life – Christopher G. Langton

Evolving Virtual Creatures - Karl Sims

Introduction to evolving virtual creatures

Karl Sims is well known in the field of computer graphics and artificial life with several well-received publications on the topics. His paper titled 'Evolving Virtual Creatures' was the primary inspiration for this project and provided much of the necessary information to begin designing the agents, environment and evolutionary algorithm. The report describes a novel system for creating virtual agents that interact and operate in a fully three-dimensional physics world. The agent morphology and their neural systems are designed by a genetic algorithm. Karl Sims tests his algorithm with multiple different fitness evaluations to direct the optimisation process towards specific behaviours, such as walking, following or swimming. Directed graphs are used to describe both the agent's morphology and the neural circuitry. The directed graphs define a hyperspace containing an indefinite amount of agent solutions and can be searched using optimisation techniques to find ones which use a successful strategy. Karl's project found some very interesting results and also outlined many implementation steps which became extremely useful for the author [5].

Agent morphology

The phenotype of a creature is constructed of a hierarchy of three-dimensional rigid-bodies. The hierarchy is represented by directed graphs which contain instructions to construct the agent and provides an easy mechanism for adding recursive components. An example of the genotype and corresponding phenotype are presented below [5].

Figure [1]

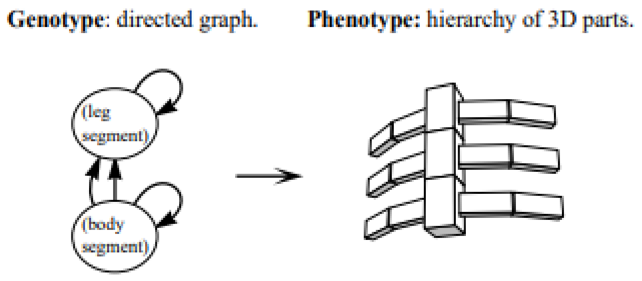


Image taken from 'Evolving Virtual Agents – Karl Sims'

The phenotype hierarchy start point is defined by the root-note, other parts are synthesised by tracing through connections of the directed graph. Graphs can be recurrent, and nodes can connect to themselves. Each node contains information about the corresponding rigid body such as the dimensions. Each node is connected via a selection of joints (much like in nature) that allows the bodies to move in relation to one another depending on the joint type. Each joint contains limits based on the body configuration and joint type. Finally, each node has a list of other nodes that are connected. Joints and body parts are also capable of containing a sensor which measures inputs from the agent and its environment such as accelerometers, photosensors, joint angle sensors or contact sensors [5].

Artificial Neural network (dataflow program)

Agents use an artificial neural network (ANN)-like system with input nodes, activation nodes and output nodes (response nodes) to give control to the agents over their movement. While the implementation is not a typical ANN (less biologically realistic) and more like a dataflow program the diverse amount of functions available allows for interesting behaviours to form. A sub-set of functions available for neurons are, sum, product, divide, sum-threshold, greater-than, sign-of, min, max, abs, etc. Nodes can be connected to form more complex functions. Input values follow the program flow while simultaneously being operated on until they reach an activation function. Effectors are activated by activation functions which stimulate effectors. Effectors are used to respond to the current input states of the agent and in this case are usually rotatable joints [5].

Behaviour Selection

Like many other genetic/evolutionary algorithms this report uses a fitness measure to optimise agents towards a specific goal. The report contains many fitness measures such as success in walking the furthest, following a light or jumping the highest. When an agent is placed within an environment the input sensors begin to activate which stimulates a response from the effectors. After a specified amount of time has elapsed the fitness score of the agent can be measured. Agents with high fitness scores are more likely to survive and reproduce [5].

Saving computation

Some agents are doomed from the beginning due to their poor design such as being motionless, in these cases the agents are removed from the simulation and given a fitness score of 0 [5].

Environment (land/walking)

The report discusses many fitness measures however, the most applicable to the author's report are the 'walking' creatures and therefore will be the only environment discussed. The land environment is given gravity and a static ground plane which exhibits friction on the agents. The report also mentions it can be necessary to prevent creatures from generating high velocities by simply falling over. This can be overcome by first simulating the environment with no friction until a stable centre of mass is established [5].

Creature Evolution

All population based evolutionary strategies use an initial population of genotypes that are randomly generated. This report also has the capabilities of generating agents based on a seed. The survival-ratio determines the number of agents that will survive each epoch of the simulation. The report outlines that a population size of 300 and survivor-ratio of 1/5 was suitable. Those that are culled are replaced by the offspring of the other members, so the population remains constant. An agent's survival rate and reproductive success is proportional to its fitness score. Agents reproduce by crossing over the genomes of two or more agents [5].

Mutating directed graphs

The report outlines several mutation methods to cause random changes in an agent's genome over time.

1. **Node mutation**

   Each node is subject to changes in its internal parameters which code for the size and shape of the node. Floating point values are mutated by adding several random numbers for a Gaussian-like distribution so small adjustments are more likely. Boolean values are simply flipped during mutation [5].

2. **Adding and removing nodes**

   New nodes can be added to an agent's directed graph by creating new node and connecting. These two steps are performed independently, and a new node will only be attached if a connection to it is also mutated. If not, it is later garbage collected. Nodes can be removed by simply removing the connection to it and letting the garbage collection remove the disconnected node. The morphological graphs are capable of variable size so can therefore expand past the limitations of their creation [5].

3. Joint mutations

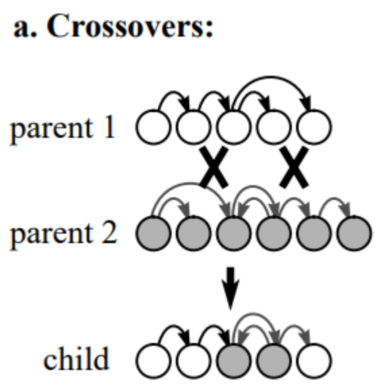
The parameters of each connection are subject to possible mutations, in the same way the node parameters are. Pointers can be moved to randomly selected nodes [5].

Mating directed graphs

Sexual reproduction allows components from more than one parent to be combined into a new offspring. Mixing two agent genomes creates a huge number of potential offspring and permits features to evolve independently [5].

Crossover

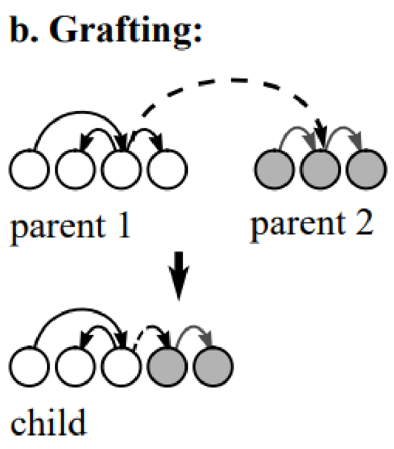
Figure [2] – example of crossover



The nodes of two agents are aligned, one or more crossover points are then defined for the two genomes and the resulting combination of the genomes becomes new offspring. The connections of the nodes are copied with it and simply point to the same relative node location as before. If copied connections point out of bounds, then they are randomly reassigned [5].

Grafts

Figure [3] – example of grafts



The grafting method mates two agent genotypes by connecting a node of one agent to a node of another at random. Newly unconnected nodes of the first agent are removed and the newly connected node of agent two are appended to a new directed graph. Agent genomes are also subject to mutations after mating but is much less likely to occur [5].

Results

The report found that, 'A population of interbreeding agents often converges towards homogeneity, but each separately run evolution produced completely different locomotion strategies that stratify the required behaviour'.

The walking fitness measure produced several simple agents that can 'shuffle' or 'hobble'. Walking styles included, lizard-like gaits, rocking back and forth, pushing/pulling and crawling [5].

Box2D v2.3.0 User Manual and Introduction to game physics with Box2D

An introduction to Box2D

Box2D is an open source 2D physics simulator engine written in C++ and has been used by many reputable game developers. Box2D is a constrained ridged body physics engine which is capable of simulating convex polygons, circles, edges and is able to join objects together with joints. Box2D supports gravity, friction, restitution while also providing the mechanisms for collision detection and collision response[7].

Although the final product does not contain a physics engine created by the author, it is important to understand the underlying processes which control the physics simulation. This section will cover the necessary components from Box2D used to construct EvoSim.

b2Vec2

b2Vec2 is a two-dimensional vector with two 32-bit float member variables corresponding to an x and y position. Vector mathematics and vector calculus are the fundamental building blocks of any physics engine. Vectors are a way of representing position coordinates with respect to the origin in an n-dimensional space. For this system a 2D Cartesian vector space is used. A vector in 2D space is represented by three values, one for each axes x and y.

$$a = \begin{bmatrix} x \\ y \end{bmatrix}$$

a represents a 2D vector. Each vector represents a unique position in the cartesian space making it a one to one relationship[3].

b2World

To establish a Box2D world a b2World pointer is created. This pointer is the primary way to interface with the Box2D physics world and is used to manage memory, the addition and alteration of bodies and fixtures in the world and is used to step over the simulation. The constructor for b2World takes a b2Vec2 parameter which defines the direction and magnitude for the force of gravity within the world [6].

Adding bodies to b2World

Body definitions

Creating a body in Box2D consists of four main steps that all bodies comply with. The first is to define a body definition. The body definition holds the data needed to create and initialise a body.

Body definition types

There are three body types in Box2D, static, kinematic and dynamic. EvoSim is primarily concerned with static and dynamic bodies. Static bodies are used for the terrain while dynamic bodies will be used for the agents [7].

Static type

A static body cannot move during the world simulation and acts like an object that possesses infinite mass. They also do not collide with one another making them suitable for the terrain [7].

Dynamic type

Dynamic bodies are fully simulated. They behave in accordance with the forces being applied to them by the simulation. Dynamic bodies can collide with one another and static bodies. However, for EvoSim dynamic body interaction with other dynamic bodies will be turned off as there is no need for them to collide. This will be done because, it allows multiple agents to be tested in parallel/simultaneously making the program more efficient. Dynamic bodies must always have a mass that is non-zero and non-infinite mass [7].

Body definition position

The bodydef's position can be set on initialisation which is far better for performance than creating the body at the origin and moving it. All agents will be required to start from the same start location, so all body definitions are positioned to the start location on creation. Fixtures and joints are attached relative to a body's origin. The centre of mass is determined from the mass distribution of the attached fixtures [7].

Body creation

The second step is to pass the body definition to b2World's CreateBody function for it to create the body. When added to the word, Box2D copies the data out of the body definition but does not keep the reference to the bodydef pointer, meaning one bodydef can be used to define multiple bodies [7].

Fixture creation

In step three, a b2PolygonShape is created and attached using a fixture definition. Finally, for the four and final step the fixture definition is created by initialising a b2FixtureDef and passing it to the parent body that it should be attached to. Multiple fixtures can be added to a single body this allows agents to have multiple appendages [7].

Fixture density

A fixture's density is defined when the fixture definition is created. All bodies should have the same density. This will allow agents to evolve the height and width properties of their appendages to control their mass. All fixtures attached to a body are used to compute the mass properties of the entire parent body [6].

Centre of mass

The centre of mass of an agent is calculated using the fixtures attached to the agent body. It does this by calculating the distribution of mass between all fixtures [7].

Revolute joint

The revolute joint makes two bodies share a common anchor point, giving them one degree of freedom. The degree of freedom is the relative rotation of the two bodies called the joint angle. Revolute joints are used to connect the limbs of agents and allows rotation around that point. Each new body (limb) is placed at the maximum length of the parent body, specified by its chromosomes, and then connected together by the revolute joint [6] [7].

Joint limits

In order to prevent all of the agent's limbs from endlessly rotating, each joint should be given an upper and lower limit in which the limb can rotate between. Box2D will apply the necessary torque to make this happen.

These values should be relative to the limbs starting angle. Limits are used to ensure a bias in the movement. This will inevitably cause an agent to move in one direction rather than remaining still [7].

Joint torque and speed

Torque is a measure of how much a force is acting on an object causes that object to rotate. Torque is the cross product between the distance vector/momentum arm (distance between the revolute joint and where the force is being applied) and the force vector (The angle between the limb of the agent and the surface it is in contact with). The joint motor will maintain the specified speed unless the required torque exceeds the specified maximum torque member variable. If a body connected to the revolute joint makes, contact with a surface, the contact forces will be proportional to the motor force and torque. In Box2D the maximum torque for a revolute joint should be set to an appropriate amount to prevent the agents from applying to much force to the ground and propelling themselves unrealistically [7].

Friction

The friction force is used to make the objects in the world interact with one another realistically. Box2D uses coulomb friction which means the friction strength is proportional to the normal face. The friction force between two shapes combines the friction parameters of the two colliding parent fixtures. The average friction force is found using the geometric mean.

$$FinalFriction = sqrtf(FrictionA * FrictionB)$$

Because the agent's initial properties are random, their movement direction is determined by the friction and restitution interactions they have with the environment. They have no specific goal to move in any direction other than those properties being selected for during reproduction. Friction and restitution could essentially be used by an agent’s fixture to ‘grip’ the terrain by increasing the fixtures friction member variable to a very high amount [6] [7].

Restitution

Restitution allows bodies to ‘bounce’ and constitutes the elastic response between two fixtures and has a domain between zero and one. A restitution force of zero produces a non-elastic collision and will remain in place after collision. A restitution of one produces a perfect elastic collision with the body maintaining the same force/velocity after the collision. The restitution force between two bodies is calculated by finding the maximum restitution of either body.

$$FinalRestitution = Max(RestitutionA, RestitutionB)$$

When multiple bodies are in contact with one another, Box2D simulates an approximation of the restitution because Box2D is an iterative solver. If the collision velocity between bodies is very small, then the restitution is forced to be close to zero to prevent jittering [6] [7].

Fixture filtering

Collision filtering prevents the collision of fixtures or bodies between one another. Box2D provides categories and groups which allows the separation of bodies. The agents in the world want to collide with the terrain but not with each other. Agents will be given a category of -1 while the terrain will have a category of 0 [6].

b2EdgeShape

Edge Shapes are line segments. The b2EdgeShape is derived from b2Shape and are useful for creating freeform static environments in Box2D. Edge shapes are composed of two b2Vecs which are connected to form the edge. b2EdgeShapes are not able to collide with one another because they have no volume. This causes ghost collision between connected edge shapes and can be solved by placing ghost vertices between the connection. A chain of B2Edgeshapes is the ideal data structure to form the terrain for EvoSim because, it is relatively simple to randomly generate a multitude of different environments [7].

Fixed timestep

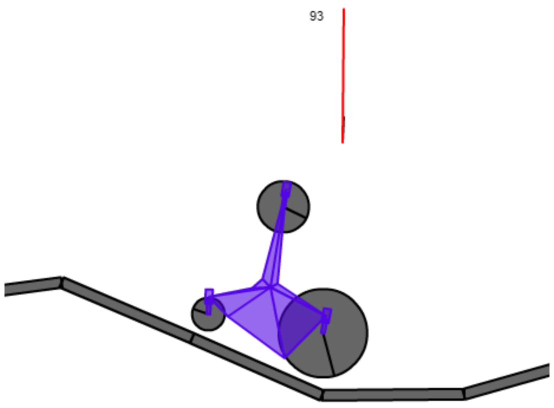
Box2D uses a computational algorithm called an integrator to simulate the physics equations at discrete points of time called a timestep. It important to pick a suitable timestep for the corresponding requirements of the program. A timestep is the quantity the physics simulation steps by or the amount of time passed since the last sample. It is analogues to taking samples of a function, the more samples taken the more accurate the predicted function is or in this case, the simulation [6].

In physics simulations it is common practise to use a fixed timestep of 1/60, meaning that the simulation steps 60 times every second. Because of this, EvoSim should also use a fixed timestep of 1/60. The main advantage of a fixed timestep is consistency. The consistency component comes from the fact that each time the simulation is run, bodies within the world will respond to forces in a nearly identical way. A 1/60 timestep is also accurate enough to capture the agent's motion within the EvoSim simulation.

Existing technologies

BoxCar2D

Figure [4] – image taken from Box2D



BoxCar2D is a program that learns to build a car using a genetic algorithm and also utilises Box2D as a physics engine. It starts with an initial population of 20 randomly generated shapes with wheels and independently tests each design. Like other genetic algorithms the cars that perform the best according to a fitness measure survive and are then more likely to reproduce to form the next generation. The idea is that two successful agents have a chance to produce an even more successful offspring. Repeating this process will eventually produce a car that is suited for the terrain generated.

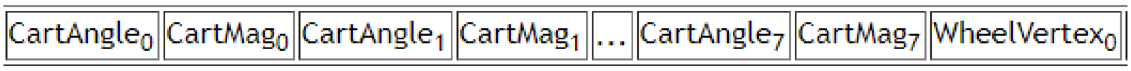
Fortunately, the author of BoxCar2D has provided some details on design and implementation.

Designing cars

Bodies are created from 8 randomly generated vectors that radiate from a single points and internal angles that add up to 2π. Wheels are added by randomly selecting vertices and selecting the wheels axel angle.

Chromosome representation

BoxCar2D uses a real value representation to store agent genomes. Each genome consists of 22 floating point values with varying ranges. An example of a car's genome is displayed below,



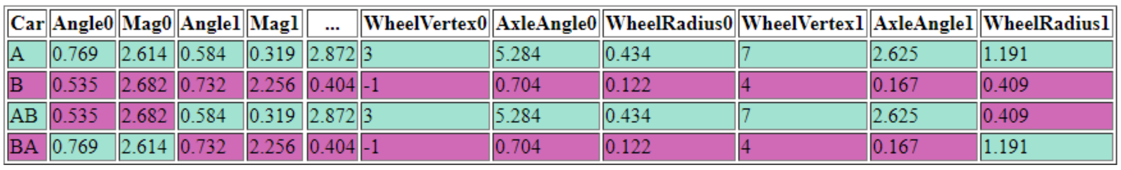
CartAngle$_0$	CartMag$_0$	CartAngle$_1$	CartMag$_1$	...	CartAngle$_7$	CartMag$_7$	WheelVertex$_0$

Selection

BoxCar2D primarily uses a roulette-wheel selection method which finds the sum of all fitness scores and divides each score by the sum to calculate a likelihood probability. These are then summed to create a probability density function which can then be selected from.

Crossover

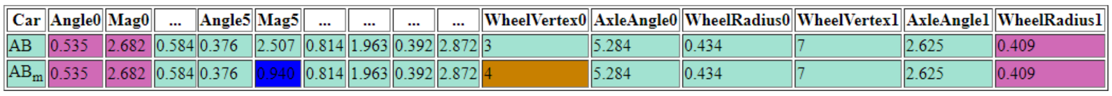
BoxCar2D uses a two-point crossover strategy that produces two offspring (AB and BA) from two parents (A and B). From the image below the two crossover points can be seen,



Car	Angle0	Mag0	Angle1	Mag1	...	WheelVertex0	AxleAngle0	WheelRadius0	WheelVertex1	AxleAngle1	WheelRadius1
A	0.769	2.614	0.584	0.319	2.872	3	5.284	0.434	7	2.625	1.191
B	0.535	2.682	0.732	2.256	0.404	-1	0.704	0.122	4	0.167	0.409
AB	0.535	2.682	0.584	0.319	2.872	3	5.284	0.434	7	2.625	0.409
BA	0.769	2.614	0.732	2.256	0.404	-1	0.704	0.122	4	0.167	1.191

Mutation

Each generation an agent's genome goes through mutation which gives it a chance to alter a random gene within its genome. Mutation can be seen in the blue and yellow,



Car	Angle0	Mag0	...	Angle5	Mag5	...	...	...	...	WheelVertex0	AxleAngle0	WheelRadius0	WheelVertex1	AxleAngle1	WheelRadius1
AB	0.535	2.682	0.584	0.376	2.507	0.814	1.963	0.392	2.872	3	5.284	0.434	7	2.625	0.409
AB$_m$	0.535	2.682	0.584	0.376	0.940	0.814	1.963	0.392	2.872	4	5.284	0.434	7	2.625	0.409

Simulation parameters

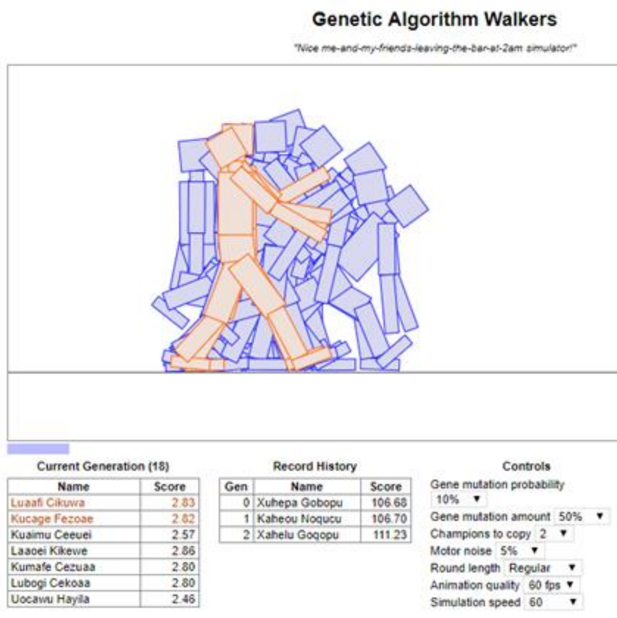
BoxCar2D allows the user to change many internal parameters of the simulation including, the population size, the mutation rate, mutation size, environmental conditions and gravity.

How BoxCar2D is relevant to EvoSim

BoxCar2D is a fun implementation of a genetic/evolutionary algorithm that also uses Box2D as its physics engine, this implies that the engine should also be suitable for the author's project. BoxCar2D also provides a lot of insight into the functioning aspects of the evolutionary algorithm with details such as, how the genome is represented, what variance operators are used and how survivor and mate selection take place. BoxCar2D also represents an achievable benchmark for which to aim for.

Genetic algorithm walkers

Figure [5] – Genetic walkers



Genetic Algorithm Walkers

"Nice me-and-my-friends-leaving-the-bar-at-2am simulator!"

Current Generation (18)

Name	Score
Luaafi Cikuwa	2.83
Kucage Fezoae	2.82
Kuaimu Ceeuei	2.57
Laaoei Kikewe	2.86
Kumafe Cezuaa	2.80
Lubogi Cekoaa	2.80
Uocawu Hayila	2.46

Record History

Gen	Name	Score
0	Xuhepa Gobopu	106.68
1	Kaheou Noqucu	106.70
2	Xahelu Goqopu	111.23

Controls

Gene mutation probability 10%
Gene mutation amount 50%
Champions to copy 2
Motor noise 5%
Round length Regular
Animation quality 60 fps
Simulation speed 60

Genetic algorithm walkers (GAW) much like BoxCar2D uses a genetic algorithm to optimise a human like figure to stay upright and 'walk'. The details of GAW are not given by the author so is much harder to determine how the genetic algorithm functions.

4.0 THE SOLUTION APPROACH

Solution options

Programming language

C++

C++ was created for application and system development. Along with the features of the procedural language, C++ has an added support for object-oriented programming features, exception handling and generic programming. C++ does not have native support of threads which may be useful for parallelising the program. C++ however, is ‘much closer to the hardware’ which makes it ideal for a physics simulation that needs to be as efficient and stable as possible. The author is very familiar with C++ after using it in their first year of education at the university of reading.

Java

Java is also a statistically typed object-oriented programming language. The main advantages to developing with Java is that it is easy to use and a widely available programming language. Java abstracts out some of the concepts seen in C++ such as pointers. This puts less strain on the programmer to remember how to pass variables. Unlike C++, Java does provide native support for threads. The author is also fairly familiar with Java as it was used in both CS2JA17 and CS3NN17.

Physics engine

Build a physics engine

One potential option is to build the physics engine from scratch. This would require a lot of background reading and understanding of linear algebra and calculus. The physics implemented would be the minimum required. For example, ridged body collisions, friction and joints. The benefit of this is that the author can have full understanding of how the physics is operating. The main downside is the initial investment and difficulty.

Box2D

Box2D is a free open source 2-dimensional physics simulator engine written in C++ and published under the zlib license and can be found on GitHub. This gives the author access to the source code. Box2D has a variant known as JBox2D that can be used in Java. There are lot of resources online about Box2D, including an official manual and unofficial book.

Box2D uses continuous collision integration which makes it better at handling faster moving objects. It also contains joint types that are not in other physics engines. Box2D also contains a feature known as sleeping bodies, when they come to rest it can be detected. This will help with detecting agents that are no longer moving. Box2D is light-weight and contains a reduced number of features compared to others available. Although, this is also a benefit in some scenarios.

BulletPhysics

Bullet is a physics engine which simulates collision detection, soft and rigid body dynamics. It has been used in video games as well as for visual effects in movies. The Bullet physics library is free and open-source software subject to the terms of the zlib License. The source code is hosted on GitHub which means that the author has complete access to understand or change aspects of the engine. Bullet has plenty of available resources online to help with the implementation. It also has an extension which is a java wrapper for the C++ engine.

BulletPhysics has the capability of 3D simulations and also has access to a rich set of rigid body and soft body constraints with constraint limits and motors, much more than Box2D.

Rendering

OpenGL

Open Graphics Library is a cross-language, cross-platform application programming interface for rendering 2D and 3D vector graphics. It provides developers with an easy way to create crossplatform games or port a game from one platform to another. OpenGL is supported on most current day systems with modern OpenGL 3.0 released in 2008 containing a majority of important features. OpenGL is generally considered more powerful than alternatives such as DirectX.

Vulkan

Vulkan is a low-overhead, cross-platform 3D graphics and computing API. Compared to OpenGL, Vulkan is intended to offer higher performance and more balanced CPU usage. Other major differences from OpenGL are Vulkan being a considerably lower level API and offering parallel tasking. Vulkan also has the ability to render 2D graphics application. In addition to its lower CPU usage, Vulkan is also able to better distribute work among multiple CPU cores.

Chosen solution

The author has decided to implement the program using C++ due to their familiarity and experience with the language. Another reason for this is because Box2D was selected as the physics engine. Box2D is also written in C++ so it made sense to use the original implementation rather than a wrapper written for Java. Box2D was selected for its light-weight capabilities but still contains all of the desired features required by the program such as revolute joints for movement and physics forces such as friction and restitution. There is also a number of useful features such as sleeping bodies which will be utilised by the project. The rendering pipeline selected for the project is OpenGL. This was mainly due to using windows as the development platform and the plethora of online resources and tutorials to teach and explain how OpenGL works.

Acceptance criteria

The following acceptance criteria are defined to validate the implementation against the technical objectives and the user's needs.

Main criteria

1. The program should use/make a suitably realistic physics engine to simulate the motions of agents in an environment.
2. A backend library should be created to deal with user input, error handling and any other management functionalities.
3. The program should provide a mechanism for visualising the evolutionary algorithm in progress. This will be used to verify the creation of agents in Box2D, validate the evolutionary algorithm and provide an interactive interface for the user.
    a. An input/output manager must be used to detect keyboard and mouse inputs
4. The evolutionary algorithm must generate a variety of agent morphologies that satisfy the requirements of the environment.
    a. These agents should be saveable and loadable to/from file.

Agents criteria

1. The agent generator must create agents from a sufficiently complex agent solution space where many agent morphologies are theoretically possible.
    a. The morphology must be infinitely expansive through evolution
2. Agents should be constructed of lines and/or wheels that use rotating joints in order to manoeuvre through the environment.
3. Agents should use a simple ANN to control their movements
    a. The ANN should be optimised via the EA.
4. An agent's genome must be saveable and loadable to/from a file.

Evolutionary algorithm criteria

1. A variety of fitness measure should be used to demonstrate the versatility of evolutionary algorithms.
2. A number of different environments should be created for agents to optimise towards.
    a. Flat land terrain, Non-flat land terrain, Water
    b. Terrains must be saveable and loadable
3. The evolutionary algorithm must use a variety of crossover and mutation mechanisms.
4. Statistics must be recorded about each epoch which can then be displayed on graphs.

5.0 DESIGN

Specification

Odingine class diagram

Figure [6] – Odingine UML class diagrams

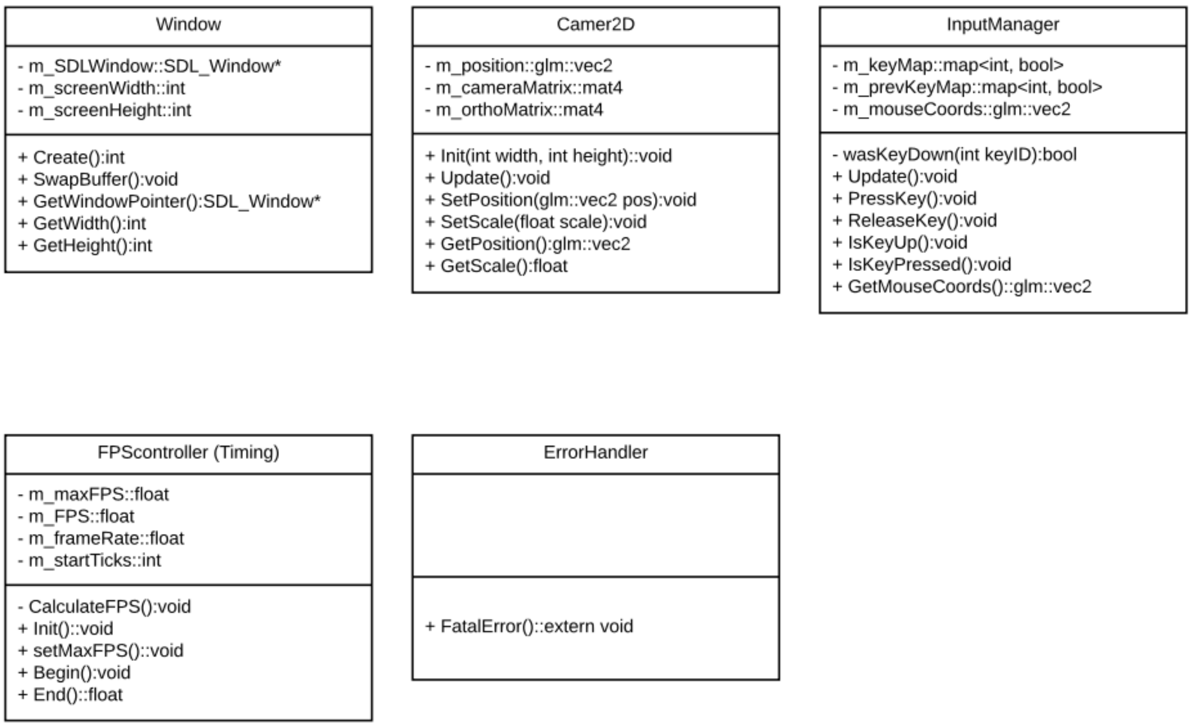


Figure [6] shows the UML class diagrams for the backend library Odingine. Odingine provides window management, IO management, controls the camera and calculates the fps. The idea behind the library is to allow for the quick prototyping of programs. Once the library is set up it can be reused multiple time for separate projects. The library uses an OOP design that utilises features such as encapsulation by providing getter and setter functions to retrieve information from classes. The library is later implemented into EvoSim and the classes can then be used by EvoSim.

EvoSim class diagram

Figure [7] – EvoSim UML class diagrams

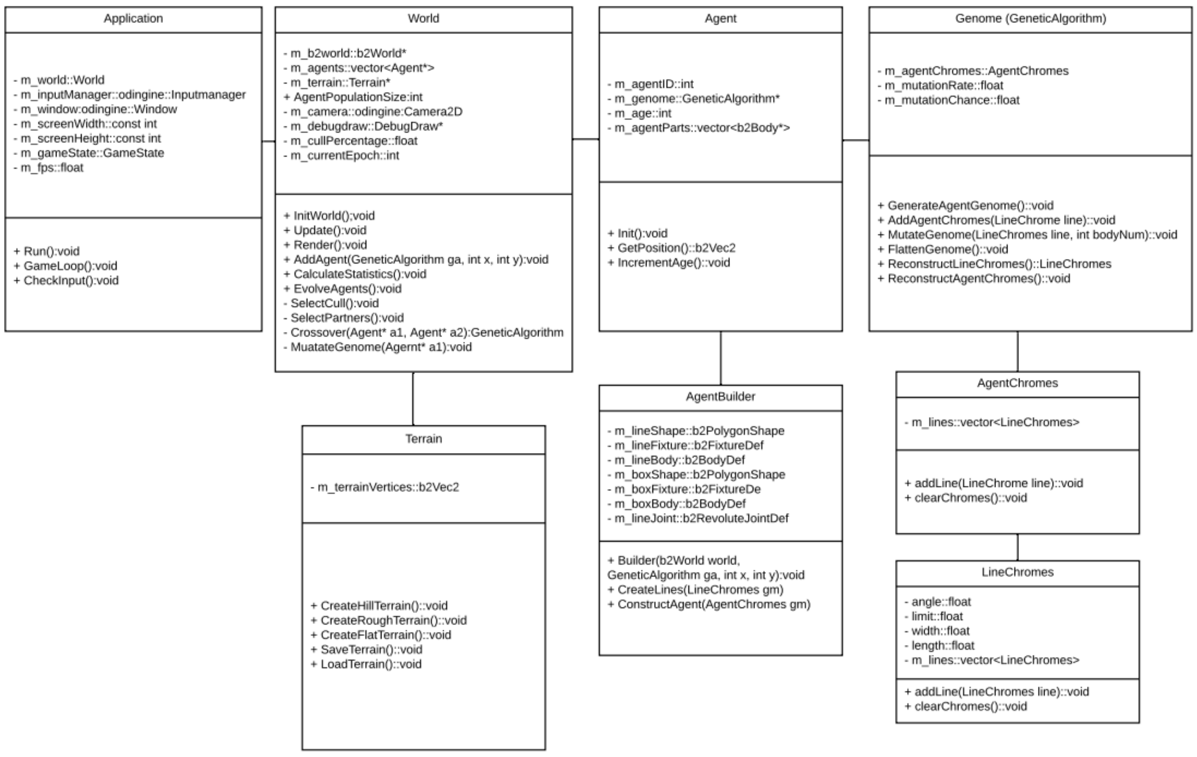

Application
- m_world::World
- m_inputManager::odingine::Inputmanager
- m_window:odingine::Window
- m_screenWidth::const int
- m_screenHeight::const int
- m_gameState::GameState
- m_fps::float
+ Run():void
+ GameLoop():void
+ CheckInput():void
World
- m_b2world::b2World*
- m_agents::vector<Agent*>
- m_terrain::Terrain*
+ AgentPopulationSize:int
- m_camera::odingine:Camera2D
- m_debugdraw::DebugDraw*
- m_cullPercentage::float
- m_currentEpoch::int
+ InitWorld();void
+ Update():void
+ Render():void
+ AddAgent(GeneticAlgorithm ga, int x, int y):void
+ CalculateStatistics():void
+ EvolveAgents():void
- SelectCull():void
- SelectPartners():void
- Crossover(Agent* a1, Agent* a2):GeneticAlgorithm
- MuatateGenome(Agernt* a1):void
Agent
- m_agentID::int
- m_genome::GeneticAlgorithm*
- m_age::int
- m_agentParts::vector<b2Body*>
+ Init():void
+ GetPosition()::b2Vec2
+ IncrementAge()::void
Genome (GeneticAlgorithm)
- m_agentChromes::AgentChromes
- m_mutationRate::float
- m_mutationChance::float
+ GenerateAgentGenome()::void
+ AddAgentChromes(LineChrome line)::void
+ MutateGenome(LineChromes line, int bodyNum)::void
+ FlattenGenome()::void
+ ReconstructLineChromes()::LineChromes
+ ReconstructAgentChromes()::void
Terrain
- m_terrainVertices::b2Vec2
+ CreateHillTerrain()::void
+ CreateRoughTerrain()::void
+ CreateFlatTerrain()::void
+ SaveTerrain()::void
+ LoadTerrain()::void
AgentBuilder
- m_lineShape::b2PolygonShape
- m_lineFixture::b2FixtureDef
- m_lineBody::b2BodyDef
- m_boxShape::b2PolygonShape
- m_boxFixture::b2FixtureDe
- m_boxBody::b2BodyDef
- m_lineJoint::b2RevoluteJointDef
+ Builder(b2World world, GeneticAlgorithm ga, int x, int y):void
+ CreateLines(LineChromes gm)
+ ConstructAgent(AgentChromes gm)
AgentChromes
- m_lines::vector<LineChromes>
+ addLine(LineChrome line)::void
+ clearChromes()::void
LineChromes
- angle::float
- limit::float
- width::float
- length::float
- m_lines::vector<LineChromes>
+ addLine(LineChromes line)::void
+ clearChromes()::void

Figure [7] represents the UML class diagram for the EvoSim application. The lines represent which classes are utilise one another.

The Application class will provide the main loop for the program. From there the entire evolution simulation is controlled, for example deciding when to spawn agents into the world. This will be done when the trial time for an epoch is done. It also initialises the world class and provides functions that implement the InputManager for processing user input and producing the correct response.

The world class will initialise the b2world along with providing functions for incrementing the step within the physics simulation and functions for rendering the agents in the world. The world also provides functions for controlling agents within the world and will perform culling, mate selection, crossover, mutation and initialising the next generation of agents. The world class contains both the terrain information and the currently alive agent information.

The terrain class defines the terrain that will be used in the world. It will use edge shapes to form terrain as specified by the random terrain generators.

The Agent class will hold information used by Box2D to construct an agent along with its genome (GA). Each agent contains one GA and is initialised upon construction. The Agent class also contains the Agent Builder struct which is used for implementing the agent into the Box2D world.

The genome (GA) contains two structs, AgentChromes and LineChromes used for holding the agent genomes. They will encapsulate the structure of an agent in a spanning tree. The GA will provide mechanisms for converting a genome to string and back. It will also perform mutations and crossover on a specific agent.

Designing the agent solution space

Symmetry in Nature

Symmetry can also be found everywhere in nature, from galactic mega structures like spiral galaxies to the 6-fold symmetry of tiny snowflakes and even by the creation of another organism such as honeycomb. It is therefore clear that symmetry is an important component to the universe around us. This connection to symmetry also applies to organisms where it can be seen in almost livings things, including humans which exhibit vertical bilateral symmetry down the middle of their body. The two main forms of symmetry in nature are radial symmetry and reflective symmetry.

Radial symmetry

Figure [8] – example of radial symmetry in a starfish

Radial symmetry is rotational symmetry around a fixed point known as the centre. Radial symmetry can be cyclic or dihedral. Cyclic symmetry is when the internal angle of a circle is sub-divided up into n equal parts. Dihedral symmetry differs from cyclic in that they have reflection symmetry in addition to rotational symmetry. The starfish in Figure[8] has five-way dihedral symmetry as it has five rotations of 72 degrees and five line of symmetry.

Reflective symmetry (bilateral symmetry)

Figure [9] – an example of reflective symmetry

Bilateral symmetry divides an organism into roughly mirror images across a line of reflection. Bilateral organisms are divided into left and right halves which generally have one of each sense organs on each half. Bilateral symmetry can be observed in 99% of living organisms which seems to indicate that it plays some role in the development of organisms. Facial symmetry in humans has even been suggested as an influence in judgements of attractiveness.

Box2D Constraints

Because Box2D is a ridged body physics engine that utilises hyperplane separation theorem it can't handle the collisions of concaved shapes. Therefore, the agents must be constructed out of concaved bodies [6].

Box2D also only allows 8 vertices per polygon by default, this can be changed in settings.h but may cause instability. In order to ensure the stability of the simulation, which is of high priority, agents will be designed without requiring more than 8 vertices per polygon [7].

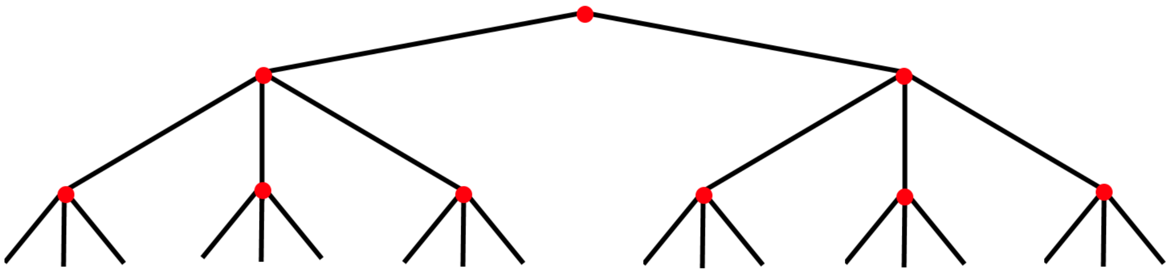
Genotypes and Phenotypes

Genotypes represent the genetic material of the agent. Phenotypes represent the visible traits and characteristics or the agent which depend on the genotype and the environment. An agent's phenotype traits are also composed of the agent's 'form', how the agent is constructed into it final shape or even secondary phenotype traits such as its centre of mass which is a result of the agents form. Natural selection occurs between the phenotypes of an agent and its environment [10].

Agent Morphology

An agent tree structure will start with the root-node and branch out using two sub-trees, known as the left and right halves. Each line on the graph will contain attribute information about each line such as its height and width dimensions, colour, etc. Lines will be connected via joints which will have their own attributes such as, initial angle and angle limits which the joint can rotate between. Lines will always be placed on the end of other lines. Because symmetry is so ubiquitous within living organisms the author thought that it would be interesting to provide a way for agents to form both symmetrically and non-symmetrically which can later be compared.

Figure [10] – maximum spanning tree of agent design



Agent tree structure

Describing the agents as tree structure creates a language which can be used to define a hyperspace of all theoretical agent morphological solutions. The agent generator will be confined to the limitation seen in figure [10]. Figure [10] represents a fully connected agent morphology. The black lines indicate the location of limbs, while the red dot indicate where joints will be. Note that the length, width and angles of these lines are variable.

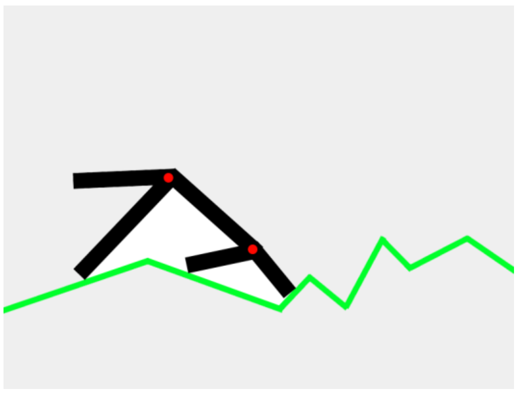
The tree structure also represents the actual physical morphology (phenotype) of the agent. Within its structure it contains the instructions for creating agents and provides a way to easily replicate or save the morphology of an agent.

The tree structure is convenient for its recursive properties which makes it possible to traverse easily using post order-like traversal, perform operations on, save and load existing structures.

While an agent's morphology is restricted by the limits of the agent generator, the evolutionary algorithm will not be so confined and will allow the ability to extend a maximum tree structure seen in figure [] via mutation. The purpose of limiting the agent generator is to prevent over complex agents forming in the initial population. Due to the simplicity of the task it is may be very easy for agents to be too complex, which will ultimately hindering them. Allowing complexity to form through evolution means that if it arises and is efficient it will remain.

While this design for agent morphology is much simpler than the one seen in Karl Sim's 'Evolving virtual agents' it does however use recursive techniques to construct agents and a similar ability to extend the agents morphology over time. Karl Sims' stressed that the agent's solution space must be expandable. It could also be argued that the tree structure is much more suited to the 2D nature of the problem.

Figure [11] – a quick illustration for what an agent might look like



Designing the evolutionary algorithm

Introduction to Evolution by Natural Selection

Evolution by natural selection in essence is best phrased by Darwin himself, "Decent with modification" coined in the origin of species. Darwin was the first to explain natural selection as a mechanism for evolution, even though he did not know how certain traits were passed on (by genes) the principle stays the same. In a population two organisms are not identical because of mutations and sexual reproduction. two organisms will over reproduce, meaning that they will create more offspring than nature can provide for. Due to limited resources (lack of food, mates, etc) this will cause competition or as Darwin phrased it, "struggle for life". There will be a natural selection for fit traits or genes (a metaphorical filter). The fitter or better adapted an organism is to its environment the more survival and reproduction chance it has. this will result in a change in allele frequency of a population (microevolution). not only will a population evolve but also be better adapted to its environment. An accumulation of microevolution can result to macroevolution which will lead to speciation (the formation of new species) [10].

Natural selection is also a blind, unsupervised, backwards looking process, that requires no additional/heuristic knowledge. It is also non-random because it uses reproductive fitness as a measure of success which determines the likelihood of mate selection and survival selection. Finally, NS is non-progressive as the design is suited to the requirements of the environment.

Darwin’s four postulations

The four postulations made by Charles Darwin in the ‘On the Origin of species’ – 1859

1. Individuals within a species are variable.
2. Some of the variations are passed on to offspring.
3. In every generation, more offspring are produced than can survive.
4. The survival and reproduction of individuals are not random: The individuals that survive and go on to reproduce, or who reproduce the most, are those with the most favourable variations. They are naturally selected

Evolutionary algorithms

Throughout history many great inventors and engineers have been inspired by the designs witnessed and formed in nature. Nature can be compared to a grindstone, refining and optimising the design and body plan of organisms to fit their environment. In some cases, nature presents solutions to problems that can be adapted to fit the requirements of human problems, such as flight and the design of the wing or glider. Evolutionary algorithms however, take this a step further by not only being inspired by nature but utilising the mechanism of nature (natural selection) to achieve a similar optimisation component. Evolutionary Algorithms, (EAs) essentially perform optimisation on learning tasks through the use of evolution.

Evolutionary algorithms are population-based solutions which maintains a collection of agents used to perform the optimisation task. EAs are also fitness-orientated because their survival and reproductive success is based on a measure of fitness. Finally, they are variation-driven because the agents will undergo a number of variation operators which explore the solution space by creating new agents with different genomes.

Evolutionary algorithms are a form of soft computing which uses inexact solution methods for computational hard problems such as NP-Complete problems. For hard problems it may not be possible to generate the global optimum in a reasonable amount of time making them more suited to heuristic algorithms. It is clear that the dimensionality of EvoSim is complex and hard to define so it very suited for an EA.

Evolutionary Algorithm Design

Main components of an Evolutionary Algorithm:

1. Agent genotype representation.
2. Initialisation of the initial population.

3. Evaluation of the phenotypes via fitness (The distanced travelled)
4. Survivor mechanisms.
5. Recombination (mate selection mechanism).
6. Definition of the variation operators (mutation and crossover).
7. Technical parameters such as mutation rates and population size.

Figure [12] – Life cycle of an evolutionary algorithm

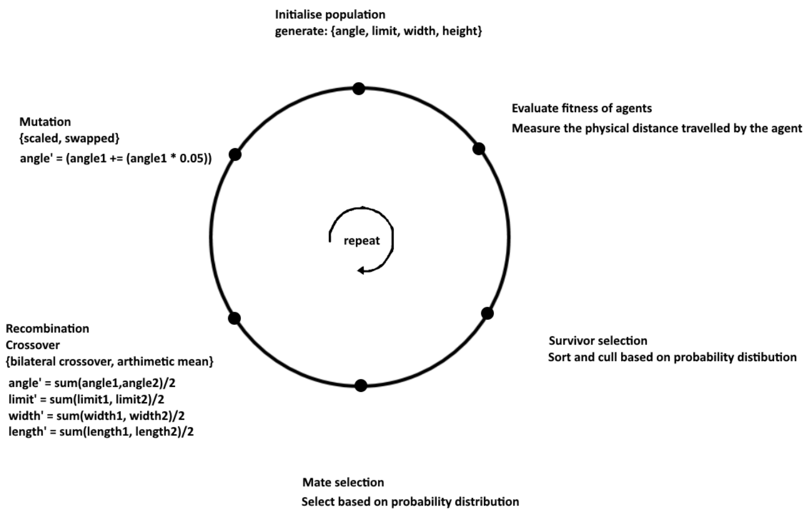

Initialise population
generate: {angle, limit, width, height}
Evaluate fitness of agents
Measure the physical distance travelled by the agent
Survivor selection
Sort and cull based on probability distibution
Mate selection
Select based on probability distribution
Recombination
Crossover
{bilateral crossover, arthimetic mean}
angle' = sum(angle1,angle2)/2
limit' = sum(limit1, limit2)/2
width' = sum(width1, width2)/2
length' = sum(length1, length2)/2
Mutation
{scaled, swapped}
angle' = (angle1 += (angle1 * 0.05))
repeat

Image adapted from Dr Giuseppe Nicosia lecture slides

Agent representation

The agent representation is the method used to encode the agent's genome. There are a number of different methods of implementation which all have benefits within certain problem domains. The three main methods are evaluated for their suitability for this problem. They consist of binary, real value and permutation representation [4].

Binary representation

Binary representation is the simplest and widely used method that represents the genotype in a string of binary values (Genotypic representation). This makes them suitable for problems where the solution space consists of Boolean decision variables for example solving the knapsack problem. Each value (or a set of values) correspond to changes in the phenotypes. One issue with this representation is that different bits have different significance and therefore mutation and crossover operators may have some undesirable effects.

Permutation representation

With some problem spaces the solution is represented in an order of elements. In these cases, the permutation representation is appropriate.

Real value representation

Real value representations use continuous floating-point values to define the genotype (also known as a phenotypic representation) and is the most 'natural' representation. However, the optimisation is limited to the precision of the architecture of the machine (64-bit pression).

The author concluded that the genotypes of the agents would be best as a set of real numbers because, they are generally more appropriate than binary representations for function optimisation. Phenotypic representations are becoming more popular within the domain of evolutionary strategies and tend to optimise quicker than binary representations. Because the genotype is continuous values it makes the optimisation a multimodal problem.

Phenotype translation of genotypes

From the Box2D representation of the agents we know that the four real value properties of each line fixture will be: the initial angle of the revolute joint, the limits of the revolute joint, the width of the limb and its length. In this form the agents genotype is fully expressive because all the genes determine the physical traits of that limb. Each agent is constructed of multiple limbs with the structural shape of the agent determined by the order of limbs and their corresponding connections [8].

Solution space of agents

The solution space of the agents is the range of total possible agent body configurations and attributes values, determined by the random variables assigned during agent creation. These random variables are defined by the ranges of the random number generators. The ranges can be altered by the user to change the solution space of agents. The standard ranges used by EvoSim are discussed in section 6.0.

Initialisation of initial population

The initial population is generation zero and constitutes a random selection of generated agents which are members of the agent solution space. The number of agents generated is equal to the desired population size of the simulations. The population size is determined before the simulation is run. Karl Sims' recommends a population size of around 200 for his simulations [5]. However, the author suspects that a smaller population size will suffice due to complexity of the problem being smaller. It's important to make sure the population size is large enough to ensure a good amount of variance within the population but not so large so that it take too much computation [4]. Agent creation in EvoSim is done as described in section 6.0.

Fitness evaluation

The fitness evaluation is the measure used to calculate the fitness of an agent, known as the fitness function. The fitness function acts as the measure of success for an agent. A high fitness score indicates an agent who is successful. Depending on the selection mechanism, the agents with the higher fitness scores are more likely to survive [8].

Selection mechanism

Survivor selection

Survivor selection determines which agents are culled and which are to be kept for the next generation, which are then used for mating. It's important to ensure that the most fit agents remain in the population without causing the loss of too much genetic diversity. There are many selection criteria including, roulette-wheel selection, stochastic universal sampling or tournament selection [4].

Mate Selection

Mate selection determines which agents mate and which ones don't. The objective is to select two or more agents to be breed together and hopefully produce an agent that is more successful than any of them. The same mechanisms used in survival selection can also be used in mate selection such as the roulette-wheel [4].

Types of selection

Roulette-wheel selection

The roulette-wheel selection mechanism also known as fitness proportionate selection and is analogues to a roulette wheel. An agent's probability of selection is proportional to the fitness score it achieved. This can be calculated by finding one agent score and dividing it by the total fitness of all agents and therefore normalising the scores. This is repeated until all proportions are found essentially creating a probability distribution. When removing the agent's, the inverse probability distribution is used so that those with the lowest fitness are the most likely to be selected. Agents are then selected from the distribution are removed from the algorithm or bred with another agent [4].

Variation operators

The genetic variance of a population defines the entire selection pool from which new offspring can be created. The total possible solutions of offspring are every combination of the existing pool of genes without considering mutations.

It is important to ensure a variation of genes within a population to prevent genetic drift occurring too quickly. With a small amount of variance in the genotypes the agents will not search many global solutions and quickly begin to optimise to a local minimum. The variance of genotypes will continue to decrease which amplifies the problem. It is therefore important to strike a balance in variance and diversity.

There are two popular methodologies for controlling the variation within a population, mutation and sexual reproduction (cross-over).

Mutation

Morphological mutation

Morphological mutations are mutations that change the physical structure or phenotype of the agent. Instead of altering the existing values of the agent, complexity is added or removed from the morphological structure of the agent. This means that agents can increase or decrease in complexity over time [5].

Real-value mutation

The size of the mutation step is usually difficult to choose. The optimal step size depends on the problem considered and can even be changed during the optimisation process. Small steps are generally considered better but generally slower than larger steps. The Breeder Genetic Algorithm proposed is as follows.

$$var' = var \pm range \times delta$$

$$range = 0.5 \times domain\ of\ var$$

$$a(i) = 1\ with\ probability \frac{1}{m}, else\ a(i) = 0; m = 20$$

$$delta = \sum a(i) \times 2^{-i}$$

This mutation algorithm is capable of generating most points in the hypercube defined by individual variables and mutation range. However, it tests nearer to the variable more frequently therefore, small step sizes are more probable than bigger steps. The mutation algorithm can locate the optimum to an accuracy of (range·2 ^-19).

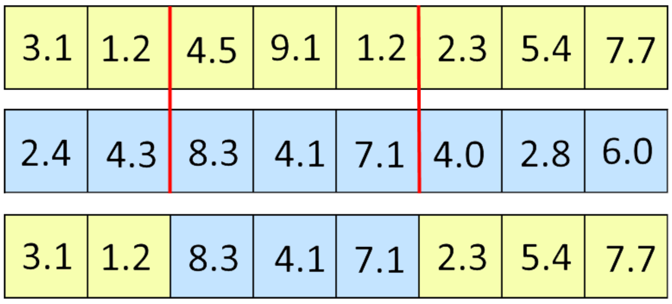
Crossover

Crossover with respect to evolutionary algorithms is one of the primary mechanisms for exchanging genes between agents to form new offspring. A description of common crossover techniques is outlined below.

N-point crossover

N-point crossover is a generalised version of single point crossover and consists of creating n crossover positions within the genome. M number of variables of the agent are selected at random with no duplicates and sorted in order. The values between successive crossover points are swapped between the two agents to produce two new offspring [9].

Figure [13] – an example of n point crossover



3.1	1.2	4.5	9.1	1.2	2.3	5.4	7.7
2.4	4.3	8.3	4.1	7.1	4.0	2.8	6.0
3.1	1.2	8.3	4.1	7.1	2.3	5.4	7.7

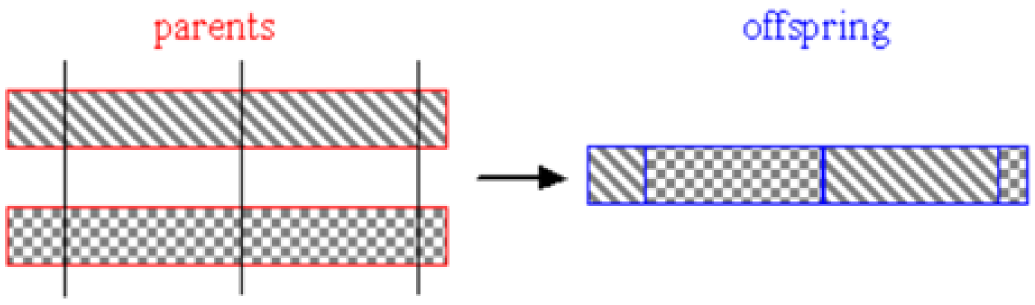
The idea behind n-point crossover is that parts of the chromosome representation that contribute the most towards the fitness of an agent may not necessarily be contained in adjacent values. The disruptive nature of n-point crossover appears to encourage the exploration of the search space, rather than favouring the convergence to highly fit individuals early in the search [4].

Uniform crossover

Single and n-point crossover defines cross points as places between loci where agents are split. Uniform crossover generalises this mechanism to make every locus a potential crossover point. A crossover mask or buffer which is the same length as the attribute values is created at random with the parity of the bits indicating which parent the new offspring should inherit the attribute from. Uniform crossover has been claimed to reduce the bias associated with the length of the attribute values used and the particular coding for a given parameter set [4].

Figure [14] – example of uniform crossover

1	1	0	0	0	0	1	1	1	1	0



Bilateral cross-over (graph or tree-based crossover)

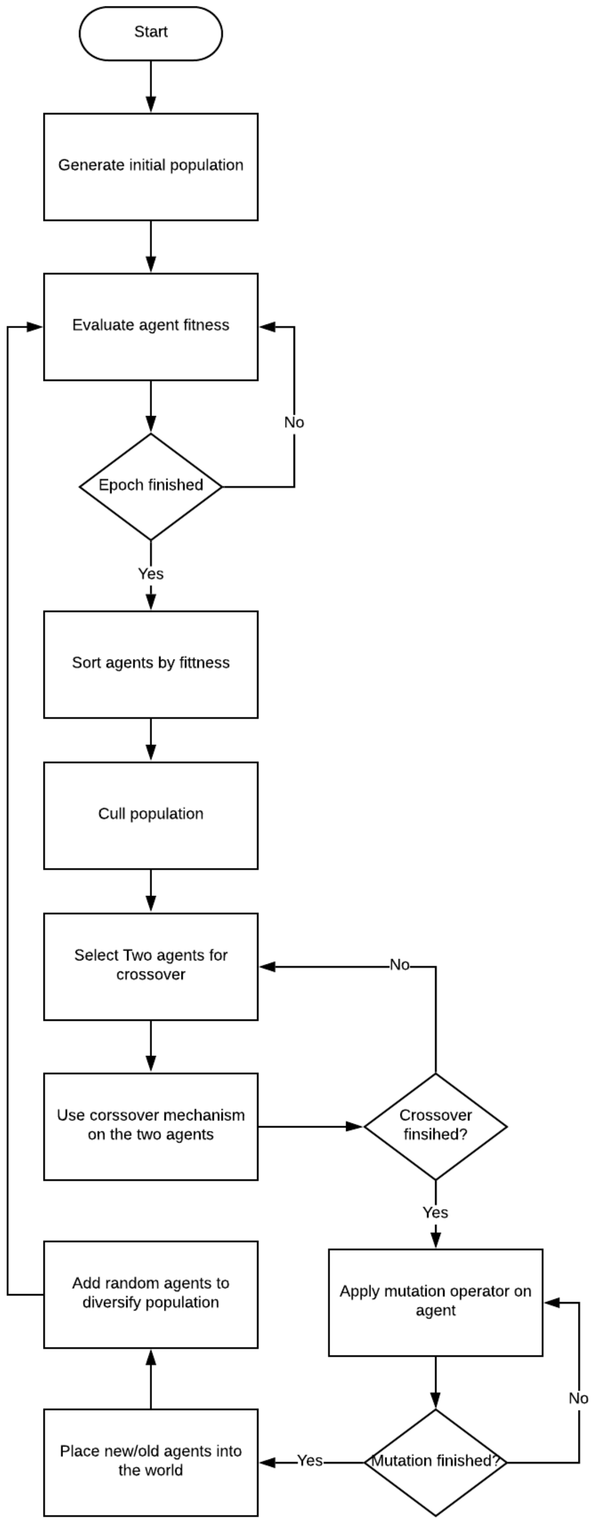
Bilateral cross-over is a method of crossover which takes one sub-tree of an agent and swaps it with another agent's sub-tree. This method is similar to the 'splicing' method seen in Karl Sim's 'evolving virtual creatures' as it is a physical crossover of two agents body compositions [5].

Arithmetic mean crossover

The arithmetic mean crossover (AMC) attempts to find the hypothetical average of two agents. Often this is not possible due to the varying length in an agent's genotype. The AMC works by finding the arithmetic means between two genomes or sub-section of genomes. This approach can also be used by graph based methods by finding the arithmetic mean between two subsections of the graph.

Evolutionary algorithm flowchart

Figure [15] – Flow chart of evolutionary algorithm


Start
Generate initial population
Evaluate agent fitness
Epoch finished
No
Yes
Sort agents by fittness
Cull population
Select Two agents for crossover
Use corssover mechanism on the two agents
Crossover finsihed?
No
Yes
Apply mutation operator on agent
Mutation finished?
No
Yes
Place new/old agents into the world
Add random agents to diversify population

The flow chart seen in figure [] outlines in diagram form, the primary operations and functions of the evolutionary algorithm used in EvoSim.

Advantages of EA’s

Evolutionary algorithms are widely applicable and useful where no good problem specific techniques are available. For example, when used in multimodalities, discontinuities, noisy constraints and multicriteria decision making problems. Evolutionary algorithms make no presumptions with respect to the problem space and their solutions have straightforward interpretations.

Disadvantages of EA’s

Evolutionary algorithms however have no guarantee of finding a global optimum within a finite amount of time. They are also often very computationally expensive as many attempts are often constructed to find a suitable solution. There is also no complete theoretical basis for these algorithms, although progress is being made by X.Yao and others.

5.0 IMPLEMENTATION

Implementing Odingine

Creating a custom library for backend input management and rendering

A custom library was created to provide input management and rendering for any interfaced program. The author named this library Odingine. Odingine interfaces with EvoSim and handles the SDL windows, user input, error handling, fps controller and some basic rendering support. The custom library was created to facilitate rapid prototyping and design for the evolution simulator. It also acts as a robust basis for additional projects which makes the initial investment to start much less. This was useful when the final project design pivoted from the original design.

Simple DirectMedia Layer

Simple DirectMedia Layer also known as SDL is a cross-platform development library designed to provide low level access to audio, keyboard, mouse and graphics hardware via OpenGL. SDL2 is used to create and control the windows that will display the Box2D simulation. SDL2 was chosen for its ability to access low level functions such as the ability to change the windows dimensions and position.

Window class

The windows class contain two main functions. The first being the create function, which is used to initialise an SDL window. Parameters can be passed in for altering the screen dimensions and position flag. The create function then initialises SDL with video and events, creates the window with the specified parameters and creates an OpenGL context. All interfaces with OpenGL are error handled appropriately to mitigate damage. The second method is used to alternate between active screens. While one screen is being displayed on the SDL window the other is being drawn to. This is a well-established technique to avoid flickering when rendering.

Camera

Camera transformations

The camera class handles mapping from world coordinates to the normalised device coordinates used by OpenGL. This makes it much easier to position the camera or agents on the screen compared to using normalised coordinates. The camera works by translating everything in the scene in the opposite direction to the desired direction using matrix transformations. The camera used will only be required to perform translation and scaling matrix transformations as no rotation or sheering is required.

In two dimensions translation is simple, the point p is given as,

$$p = \begin{bmatrix} x \\ y \end{bmatrix}$$

The transformation of p to p' can be expressed in traditional algebra as,

$$p' = \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} tx \\ ty \end{bmatrix}$$

However, OpenGL uses matrices to achieve its transformations. In order to multiply the translation matrix by the vector p it must be converted into a homogenous form. This is done by creating an artificial row on the vector that contains the value '1'. The result is the final position of the vector p'.

$$p' = \begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

The camera is also required to be scalable, so the user can inspect the world closer and further away as desired. The transformation matrix for scaling is given as,

$$p' = \begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

For any transformation, the final value of p' can be found by removing the homogeneous component. Fortunately, the glm library hides the complexity of matrix multiplication with functions used for scaling and translation.

Projection matrix in OpenGL

By default, there are two issues with OpenGL defaults, the first being that objects will appear to have the same size no matter where the camera in positioned. There other is that anything drawn must be between the normalised device coordinates. Scaling in OpenGL can be achieved by virtually enlarging or shrinking the clipping volume using a projection matrix. When the clipping volume is the shape of a box or parrelepiped it is referred to as an orthographic projection. The orthographic projection will not modify the size of any objects drawn to the screen no matter its position making suitable for 2D rendering engines. The orthomatrix used for transformation in OpenGL is as follows,

$$p = \begin{bmatrix} \frac{2}{right - left} & 0 & -\frac{right + left}{right - left} \\ 0 & \frac{2}{top - bottom} & -\frac{top + bottom}{top - bottom} \\ 0 & 0 & 1 \end{bmatrix}$$

The mathematical library glm is used to implement the transformations.

Input Manager

The input manager's purpose is to acquire input from a user, in the case of EvoSim, the only input required is from the keyboard. The input manager uses the SDL poll events function to call SDL pump events which removes the first event in the stack and copies it into the event pointer given. SDL pump events call the operating system's event processing API and transforms platform events into SDL events and pushes them onto the SDL event stack. The SDL event type is then used to work out which type of input was used and places the actual input data into an unordered map. This map can then be inspected by an input check function which residing in the game loop. The map is searched for the desired input and if found returns true. This can be utilised by an if statement to control functions or events based on input.

The current control scheme in EvoSim is as follows:

Action	Input
Move camera right	l
Move camera left	k
Move camera to agent	p
Zoom in	z
Zoom out	x
Render mode	r
Display fps	f
Add one agent	g
Clear all agents	t
Display current fitness time	m
Maximum fitness time	o

Fps Controller

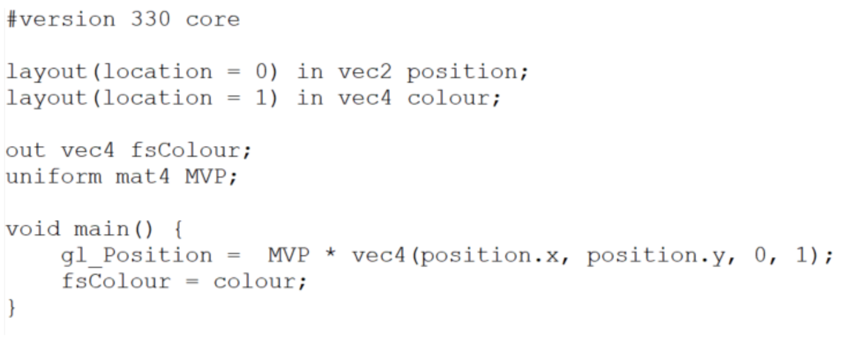
The fps (frames per second) controller class is used to measure the fps within the program during run time and limit it to a specified amount. Its purpose is to stabilise the simulation and make it more consistent. The fps controller contains two functions for measuring the fps, the begin and end function. The game logic is placed between these two methods. The begin function saves an unsigned 32-bit value representing the number of milliseconds since the SDL library was initialised into a start variable. The end function calls another function which first calculates the number of ticks for the most recent frame and added to an array of the previous ten frame times. An average is then calculated for the previous ten frames. The fps is calculated by dividing one thousand by the average frame time to convert it into seconds. If the maximum fps is greater than the current frame tick, then SDL can create a delay which pauses the program temporarily until the desired amount of fps is reached.

Compiling the shaders

The initial step is to call the OpenGL function glCreateProgram to generate a program object. Next, glCreateShader is used to create a vertex shader ID and a fragment shader ID. These in turn are passed with the file location for the corresponding shader. The same process is used to compile each shader type. The shader file is opened, and the contents is read line by line into a file content. A pointer is retrieved from the c_str of the file contents and used to tell OpenGL that file contents is to be used as the contents of the shader program. The shader is then compiled using the shader ID. Error handling is used to prevent fatal crashes when attempting to load and compile the shaders.

Vertex shader

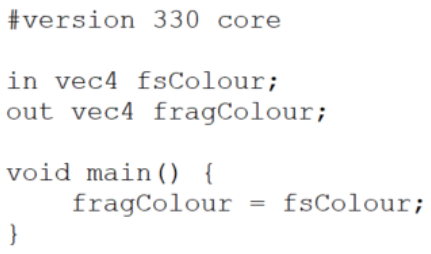
Figure [16] – vertex shader actual GLSL code



```
#version 330 core

layout(location = 0) in vec2 position;
layout(location = 1) in vec4 colour;

out vec4 fsColour;
uniform mat4 MVP;

void main() {
    gl_Position =  MVP * vec4(position.x, position.y, 0, 1);
    fsColour = colour;
}
```

Fragment shader

Figure [17] – Fragment shader actual GLSL code



```
#version 330 core

in vec4 fsColour;
out vec4 fragColour;

void main() {
    fragColour = fsColour;
}
```

Attaching and Linking the shaders

Before the shaders can be used they must be attached to the program generated previously using glAttachShader. Once the shader object is attached to the program object, the shader object is linked using glLinkProgram. Finally, glUseProgram is required to make the program object part of OpenGl's current state.

OpenGl rendering pipeline

Figure [18] – OpenGL rendering pipeline

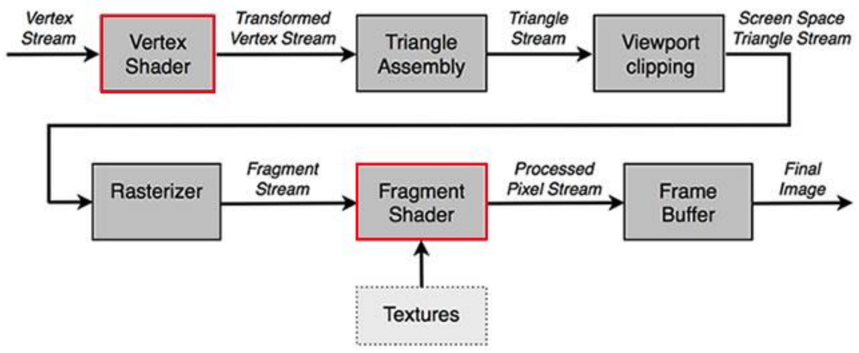


Image provided by adobe – Marco Scabia

Takes rendering data in the form of vertex attribute data and is processed by the vertex shader as individual vertices. The vertex shaders take a single vertex from the vertex stream and generates a single vertex to the output stream. This vertex data is then sent to the rasterizer where single vertex points are connected together and filled in with no colour. For each sample of the pixels covered by the primitive, a single fragment is created. Each pixel in turn is then selected from each fragment and the fragment shader determines its colour until all fragments are completed. Once all the fragments are done the data is sent to the GPU.

Rendering the simulation

Before rendering begins its important to clear the vector information contained within the two primitive renders lineRenderer and fillRenderer. This is because the simulation has changed and therefore so has the vertex information. Next, the DrawDebugData function is called inside of the current b2World.

Draw flags

b2Draw supports five different drawing flag types. However, EvoSim will only be using two of these. The first and most important is the e_shapeBit for defining that shapes will be drawn, e_jointBit for defining that the joints between shapes should be drawn.

ShapeBit

If the shapeBit is active, then the DrawDebugData function will loop through the m_bodyList in b2World which contains the body information added by the agent generator. The transformation information is then retrieved from the body which is used to transform all the connected fixtures to the correct orientation. Next, the fixture list of the current body is looped through to get each individual fixture. Then, the shape type (dynamic, static, etc) is checked. The shape type determines the colour the shape will have when draw. Finally, the b2Fixture and fixture transformation along with the colour are passed to the DrawShape function in b2World.

JointBit

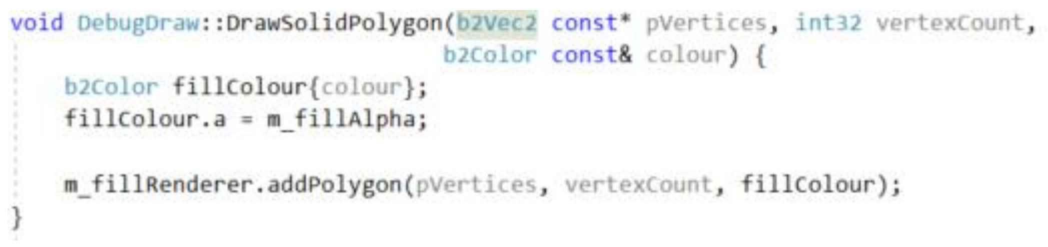
If the jointBit is active, then the DrawDebugData function will loop through the m_jointList which contains the joint information added during the agent generator. Each joint is passed to the DrawJoint function in b2World.

DrawShape

DrawShape takes the fixture, fixture transformation and colour from DrawDebugData. DrawShape supports the construction of four different shape types. e_circle, e_edge, e_chain and e_polygon. EvoSim only utilised edges which are used for the terrain and polygons which are used to construct the agent. Therefore, these will be the only two functions explained. The DrawShape function performs a switch case on the fixture type to determine which it is. If the shape is a polygon, then the total number of vertices in the polygon is retrieved. Each vertex is looped through and the new vertex information is calculated. It is calculated using the previous vertex position and the fixture transformation and then placed into an array of vertices. The new array of vertices along with the total number of vertices in the polygon and its colour are passed to the DrawSolidPolygon function located in DebugDraw.

DrawSolidPolygon

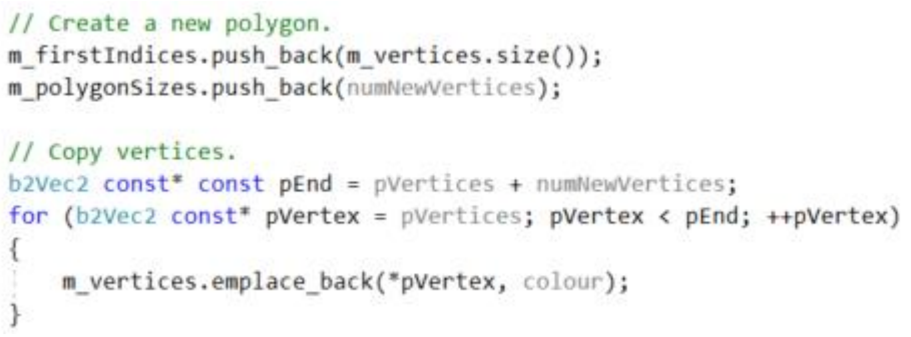
First, the DrawSolidPolygon function creates a b2Colour with alpha using the colour information passed by DrawShape. The addPolygon function of the m_fillRender is passed the array of vertices, the vertex count and colour.



```
void DebugDraw::DrawSolidPolygon(b2Vec2 const* pVertices, int32 vertexCount,
                                 b2Color const& colour) {
    b2Color fillColour{colour};
    fillColour.a = m_fillAlpha;

    m_fillRenderer.addPolygon(pVertices, vertexCount, fillColour);
}
```

AddPolygon

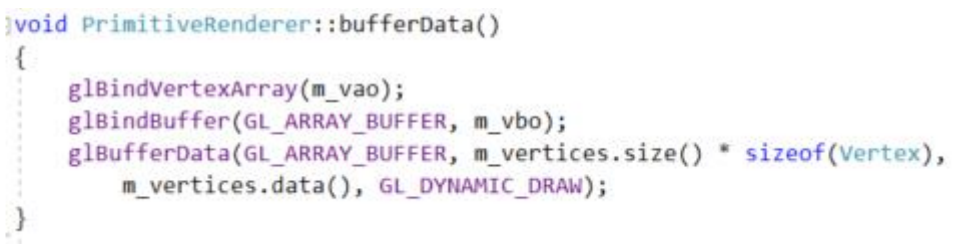
AddPolygon reserves space in m_firstIndices and m_polygonSizes which pushes the size of the new polygon on to a vector. The vertices are then looped through and assigned to the m_vertices vector along with its colour. These three variables are used by shader program and OpenGL to draw the polygons to the screen.



```
// Create a new polygon.
m_firstIndices.push_back(m_vertices.size());
m_polygonSizes.push_back(numNewVertices);

// Copy vertices.
b2Vec2 const* const pEnd = pVertices + numNewVertices;
for (b2Vec2 const* pVertex = pVertices; pVertex < pEnd; ++pVertex)
{
    m_vertices.emplace_back(*pVertex, colour);
}
```

BufferData

Once all of the vertex information has been added to the vector of vertices the BufferData function binds a vertex array object (VAO) to the current OpenGl context and then a vertex buffer object (VBO) is created and bound to the GL_ARRAY_BUFFER type, specifying that the VBO will be storing vertex attributes. glBufferData allocates the stored data for the bound VBO, with the size of the array and its reference pointer defined. The array buffer is also configured to draw dynamic objects as the agents will changing position every frame.



```
void PrimitiveRenderer::bufferData()
{
    glBindVertexArray(m_vao);
    glBindBuffer(GL_ARRAY_BUFFER, m_vbo);
    glBufferData(GL_ARRAY_BUFFER, m_vertices.size() * sizeof(Vertex),
        m_vertices.data(), GL_DYNAMIC_DRAW);
}
```

Implementing EvoSim

The World

The world class is responsible for controlling all of the interactions that happen within the world.

Creating the Box2D world

In order to create a Box2D world a b2World pointer is created in the World class. A value of -9.81 was used for the gravity value to simulate a similar amount of gravity as earth.

Simulating the Box2D world

The update function used by the world call is responsible for controlling the physics steps

```
b2world->Step(timestep, VelocityIterations, PositionIterations);
```

The timestep used by EvoSim is 1/60 which means the physics step will advance 1/60th of a second every step. The timestep also affects how the force of gravity acts on each body. Changing this value may result in different outcomes for the same initial conditions. It is common practise to use a timestep the same as the frame rate the simulation will be render at with 60 fps being the baseline standard for most games or simulations.

The velocity and position iterations effect the way bodies will react when they collide and exclusively relate to collision resolution. When two objects collide, they will usually be found inside one another. Box2D first pushes the bodies apart until they no longer overlap. From here an iterative solver is used to approximate the resulting collision. While this is not perfect, it is consistent enough for EvoSim with an agent's score varying very slightly between simulations. The velocity and position iteration values are an upper limit and if the extra precision is not necessary Box2D will use lower values. The values used by EvoSim are 6 and 2 for velocity and precision iterations respectively. After a number of tests this combination offers a good balance between performance and accuracy.

Rendering the world

The world class also contains the functions used for rendering the Box2D vertex data with the details described in section []. In short, the render requests vertex information from the Box2D world and converts in to a form OpenGl can use to render to the screen.

Creating the terrain

The world class is also responsible for initialising the terrain within the world. The terrain method allows the user to select which terrain type the simulation will use. The details of terrain generation can be found in section 6.0.

Generating a generation of agents

The world class is also used for generating the initial population of agents. It does this by populating an array with agent genomes (GeneticAlgorithm class) that are random generated as described in section 6.0. The number of genomes added is the equivalent to the population size. After each agent has completed its tests an iterator is used to select the next genome which is then constructed into the agent and tested. The buffer of agents is also used by new and existing agents on future generations after crossover.

Evolutionary algorithm

It is also the responsibility of the world class to perform the evolutionary operations on the agents. This is due to the fact the agent information is contained within the world. The details of the evolution algorithm can be found in section [].

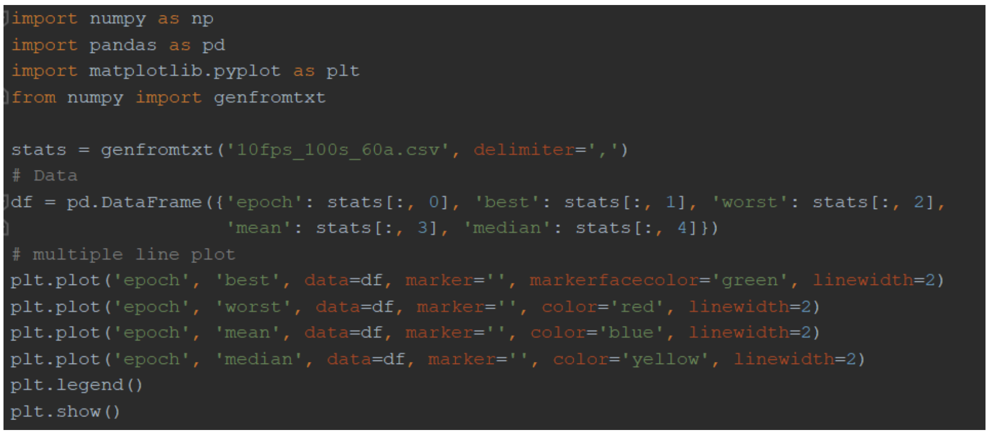
Log bodies

The log bodies function is used to display the fixtures currently active within the Box2D world. It works by retrieving the body list from Box2D and iterating through it while also collecting he position and angle of each fixture. This information is then displayed to the user. The log is used to keep track of which bodies exist within the world which is useful for error handling and testing the validity of agent generation and agent evolution.

Calculating Simulation Statistics

Once an epoch has completed statistics about the entire population of agents are calculated. This includes the mean, median, range, variance and the standard deviation. The values are outputted into a csv file which can then be plotted on a time series graph using the Pandas, MatPlotLib and numpy libraries in python.

Figure [19] – Python code used to plot graphs



```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from numpy import genfromtxt

stats = genfromtxt('10fps_100s_60a.csv', delimiter=',')
# Data
df = pd.DataFrame({'epoch': stats[:, 0], 'best': stats[:, 1], 'worst': stats[:, 2],
                   'mean': stats[:, 3], 'median': stats[:, 4]})
# multiple line plot
plt.plot('epoch', 'best', data=df, marker='', markerfacecolor='green', linewidth=2)
plt.plot('epoch', 'worst', data=df, marker='', color='red', linewidth=2)
plt.plot('epoch', 'mean', data=df, marker='', color='blue', linewidth=2)
plt.plot('epoch', 'median', data=df, marker='', color='yellow', linewidth=2)
plt.legend()
plt.show()
```

Implementing Terrain

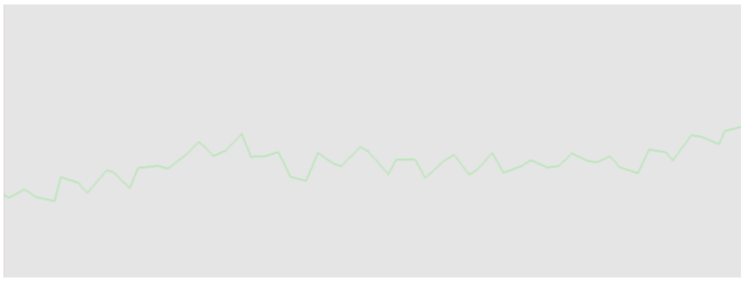
Within EvoSim there are currently three different terrain types for agents to navigate. The three terrain types are flat, rough and hill. Each is designed to be different enough so as to produce distinctly different agent phenotypes which are best adapted to each terrain type.

The terrain is held in place using a b2Body and is constructed of multiple b2EdgeShapes connected together. The chain of edges is created using left and right vertices which are connected together to create one edge. The edges are then connected by making the new left vertex equal to the previous right vertex. This guarantees that all edges will be joined together [6].

All terrain types are randomly generated at the start of each test and saved to a text file. Saving the terrain map allows the same environment to be tested multiple times with differing parameters.

Rough terrain

Figure [20] – Rough terrain

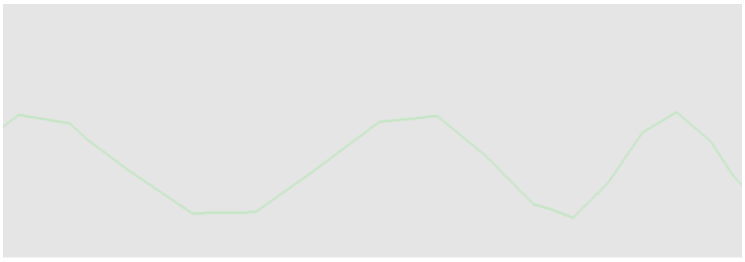


The rough terrain is created using a combination of two sine waves to create the erratic oscillations seen in figure[20]. The pseudocode for generating the rough terrain is as follows. Right and Left are b2Vec2's containing an x and y coordinate.

```
For i = 0 to 100:
    Right = (Left.x + Float(2.0, 3.0) + Sine(i * 3) , Left.y + Float(0.5, 6.0) + Sine(i * 2))
    B2EdgeShape = (left, right)
    Left = Right
```

Hills terrain

Figure [21] – Hill terrain

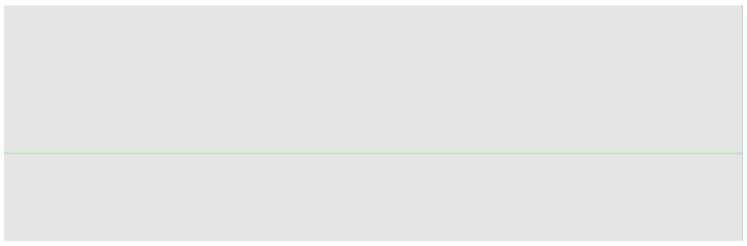


The Hills terrain is created using a combination of a sine and cosine wave to create the slow oscillations seen in figure[21]. The pseudocode for generating the hills terrain is as follows.

```
For i = 0 to 1000:
    Right = (Left.x + 3.0 + Sine(i / 8) , Left.y + 3.0 + Cos(i / 3))
    B2EdgeShape = (left, right)
    Left = Right
```

Flat terrain

Figure [22] – Flat terrain

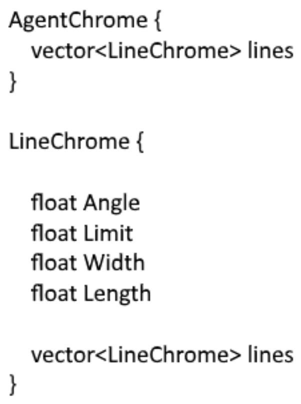


Implementing Agents

Gene structure/representation

Each agent possesses a set of AgentChromes which is effectively holds the entire genetic material or genotype of the agent. AgentChromes contains a single vector of LineChromes which is used to store the limbs of the agent. A LineChrome holds information about one limb such as, the initial angle, joint limits, its width and length. The LineChromes also hold additional information about the structure of the agent in the form of another vector of LineChromes. The genome structure can be seen below,

Figure [23] – pseudo code for agent data structure



```
AgentChrome {
   vector<LineChrome> lines
}

LineChrome {

   float Angle
   float Limit
   float Width
   float Length

   vector<LineChrome> lines
}
```

LineChromes has the ability of recursively adding LineChromes to itself, giving it the ability to create large and complex connected spanning tree structures. The structure of the genome explicitly defines the phenotype (outward appearance) and the form of the agent.

The final genome is stored in the LineChrome vector of AgentChromes. This split is where the line of symmetry appears on the agent and are referred to the right and left side of the agent.

Because bilateral symmetry is so ubiquitous in living organisms the author decided to implement it by using the LineChromes vector in AgentChromes as the 'centre' point of the agent. From this centre point the degrees of symmetry can be determined by the number of sub-trees in AgentChromes. Adding two LineChromes will give the agent bilateral symmetry. Currently EvoSim only supports bilateral symmetry. However, the symmetry is theoretically scalable to any amount of symmetry but may make the agents too complex after around five degrees of symmetry.

Generating agent genomes

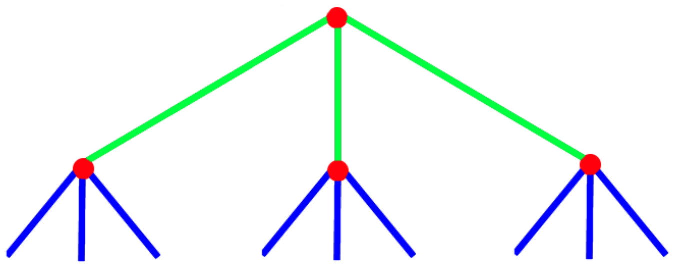
An agent genome (GeneticAlgorithm class) is the information used by an agent to construct itself. The genome defines the values of each line and the connectivity of lines.

Each line attribute is generated using the following ranges. Each random float is generated from a normal distribution.

```
Angle = float(2.0f, 3.0f) * 2 * M_PI;
Limit = float(2.0f, 3.0f) * 0.05 * M_PI;
Width = float(3.0f, 4.0f) * 0.2f;
Length = float(0.5f, 3.0f) * 5 + 1;
```

The generator constructs the agent’s left and right half independently.

Figure [24] – maximum spanning tree of a single sub-tree of an agent’s genome



The first sub-tree is passed to the genome generator where three LineChromes are initialised with attribute values (seen in green). Each one of these lines then randomly selects a number from a discrete distribution which determines the number of additional limbs (seen in blue) that line should have. The discrete distribution follows that, one appendage is ten times more likely than three and five times more likely than two. The new appendages are placed in the LineChrome vectors of the corresponding parent lines. The green limbs also use the discrete distribution to decide the number of limbs attached to the primary LineChrome. Once complete the final sub-tree is returned and pushed onto the AgentChrome and the process is repeat for the next sub-tree.

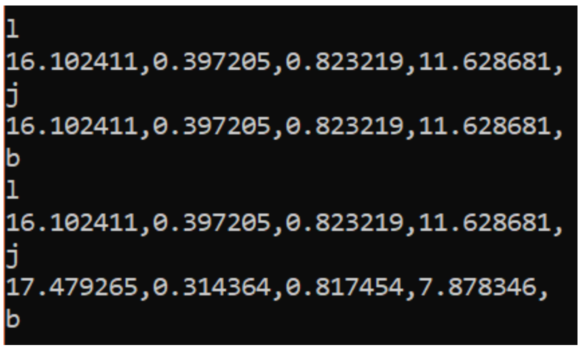
Many implementations for agent body configurations were tried, this method strikes a good balance in complexity meaning that the agent body plans are not too simple to be motionless and not too complex to be uncontrollable.

Outputting genome into a string representation

However, with the agents in this form it makes it very difficult to alter their genome or save their structure to file that can be loaded later. A parser was created to traverse the agent in post-order (reverse polish) while outputting the attribute values of the line and if it is a leaf line or a join line. A leaf line is a line with zero connections to other lines and outputs a 'leaf line' instruction (l) while a join line is a line that contains one or more lines and outputs a 'join' instruction (j). Each LineChrome line (sub-tree) within AgentChromes is traversed separately with the results being combined at the end to form the final agent structure.

The final output is the string representation of the agent genome and can be used to store the agent's genome automatically when the simulation is running, meaning the simulation can be left passively optimising. This representation also allows for operations to be performed on the genome such as more complex crossover operations and mutations. Below is the final structure of the output genome.

Figure [25] – an example of an agent genome



```
l
16.102411,0.397205,0.823219,11.628681,
j
16.102411,0.397205,0.823219,11.628681,
b
l
16.102411,0.397205,0.823219,11.628681,
j
17.479265,0.314364,0.817454,7.878346,
b
```

Reconstructing agents from the string representation

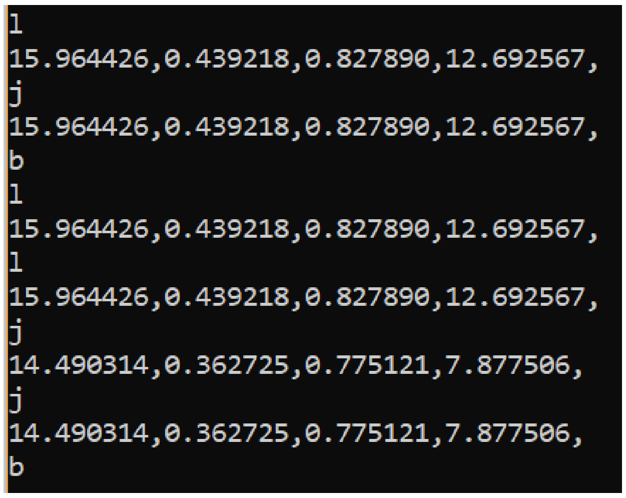
The agent code contains three separate instructions:

B – B standards for body and is used to indicate where the symmetry within the agent is. Each full genome is split by b with tree traversal happening on each half and then being recombined.

L – L stands for leaf line and is used to indicate a leaf node within the agent tree. A leaf line instruction tells the re-constructor to push the current line to the stack. A line counter is kept, storing the amount of lines attached to the next join.

J – J stands for join instruction and is used to indicate when a joint is required in the agent structure. The join instruction pops lines from the stack equivalent to the line counter (number of child lines) and places them into the vector of LineChromes of a new line. This line is then pushed back onto the stack.

Figure [26] – example of an agent genomes

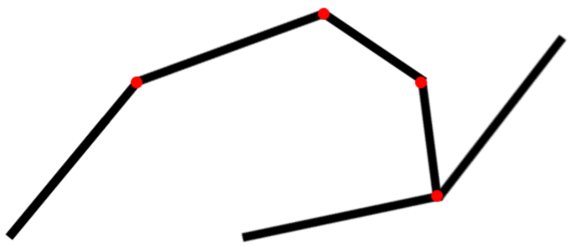


```
l
15.964426,0.439218,0.827890,12.692567,
j
15.964426,0.439218,0.827890,12.692567,
b
l
15.964426,0.439218,0.827890,12.692567,
l
15.964426,0.439218,0.827890,12.692567,
j
14.490314,0.362725,0.775121,7.877506,
j
14.490314,0.362725,0.775121,7.877506,
b
```

The ReconstructAgentChromosomes() function is used to parse the string representation of each LineChrome sub-tree and return a single LineChrome value which contains the sub-tree structure within it. When both trees have been parsed the are both added to the vector in AgentChromes to form the entire genome of the agent. The GA blueprint can then be used to recreate an agent.

The string representation of the agent genome seen in figure[] can be translated into:

Figure[27] – this physical morphology of the agent from figure [26]



Constructing the agent in Box2D

The variable required to construct the again in the Box2D world:

Agent variables

```
vector<b2Body*> agentParts;

b2PolygonShape lineShape;
b2FixtureDef lineFixture;
b2BodyDef lineBody;

b2RevoluteJointDef lineJoint;

b2PolygonShape boxShape;
b2FixtureDef boxFixture;
b2BodyDef boxBody;
```

Initialising variables

Before implementing the agents into the Box2D world the components need to be initialised. Fixtures are initialised with their shape, density and friction. Body are defined as dynamic. Shapes are given their dimensions and the revolute joints are initialised with their maximum torque force. Below are the variables used to contrast an agent within EvoSim.

linefixture - Shape, density, friction, filter

lineBody – body type e.g. dynamic

boxShape – set as box and give very small amount

boxFixture – shape, density, friction, filter

boxBody – bodytype

boxBody – set the position

lineJoint – enable all motors with max 6000 torque

Constructing the agent

Constructing the agent tree

In order to implement the agent into the Box2D world it must be unpacked recursively from its data structure, much like in the agent genome to string function seen in section 5.0.

Figure [28] – pseudo code for constructing the agents in Box2D

```
CreateLine(LineChromes gm)
    lineShape.set(gm.length, gm.width)
    body = world.CreateBody(lineBody)
    Add fixture to body
    push body to agent parts vector
    for (sg : gm.lines)
        line = CreateLines(sg)
        set angle of line
        transform line by gm.length and gm.angle

        set anchor A and B to opposite ends of line
        connect body to anchor
        connect line to anchor
        set upper and lower angle
        set reference angle
        set speed of joint and add to the world

ConstructAgent()
    clear agent parts vector
    body = world.CreateBody(lineBody)
    Add fixture to body
    push body to agent parts vector

    For (sg : gm.lines)
        line = CreateLine(sg)
        Add the values for the last line (ditto from above)
    return body
```

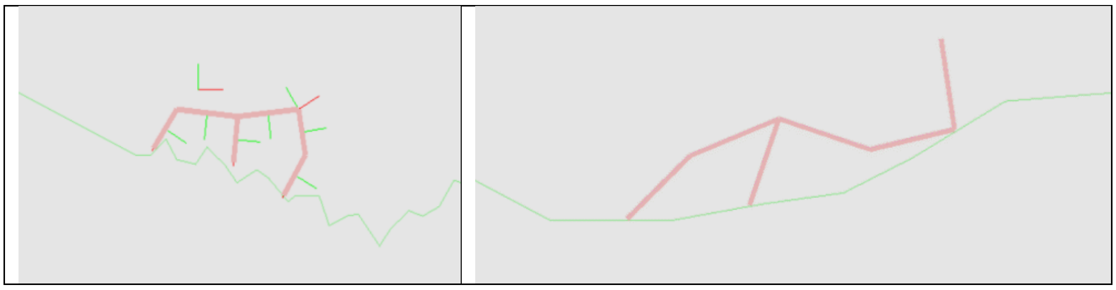
The agent generator starts by clearing the vector used for temporarily holding the b2Body pointers that hold the body information about an agent. The boxBody body definition is then passed to the CreateBody function found in b2World which returns a b2Body pointer. The boxFixture fixture is then attached to the new body using the CreateFixture function found in b2Body. The body is then pushed onto the agent body parts vector. Next, the CreateLines recursive function is called which passes the current LineChromes value sg. A for loop is used to iterate through all of the lines attached to the current LineChrome. For each line the CreateLine function is recursively called until all lines have been found and constructed into the agent. The function essentially performs a post-order traversal that builds the agent lines from the bottom up [6] [7].

Constructing one line

Each time a new LineChrome is encounter these actions are performed:

The line dimensions, height and width are added to the line shape to give the agent it physical form. The revolute joint anchor points A and B are then placed at either end of the new body. This is calculated by multiplying the length of the new line by both 0.5 and -0.5 which gives the position of each end. The current body and line are then attached as bodyA and bodyB respectively. This is how Box2D knows two objects are attached to one another. Next, the lower and upper bounds are found for the revolute joint. This is done by taking the reference angle generated and making the lower bound the reference angle minus the limit and the upper bound the reference angle plus the limit. Finally, the motor speed for the revolute joint is defined. All agents/joints have a fixed rotation speed and strength. This is to prevent the algorithm from optimising just these characteristics which will most definitely have a large impact on the score. This was done because the author is more interested in the optimisation of the morphology, movement technique or style [7].

Figure [29] – example agents constructed



Implementing Evolutionary Algorithm

Initial population

The initial population of agents is constructed via the agent generator which creates the same number of agents specified by the maximum population size. The first generation of agents will be subject to an initial filtering which removes agents that are definitely not suitable for the required task. In this case this constitutes agents whose are not moving right or motionless. Having an increased population size often improves the quality of the final result. However, a small population size often converges much quicker, but the local minima found is often not as good. This is due to a lack of population diversity caused by premature stagnation [4].

Filtering the initial population (Heuristic based fitness)

Because the fitness measure for an agent is ‘Those that travel the furthest to the right are the most fit’ the author can assume that any agents which are not travelling in the positive x direction can be discarded. This is calculated by measuring the agents x position five seconds into the simulation and if it is less than 0 then it is removed, this is known as the position threshold. Generally, this is a very successful strategy, from observations made by the author it was noted that agents that start moving towards the left generally continue to move in that direction and are therefore not appropriate for the fitness measure.

The other scenario in which it is desirable to remove agents is when they are completely motionless or just continue to move on the spot and make no progress. This is unsurprisingly a much more difficult problem than the previous filtering technique. The current implementation uses a buffer to record an agent's position over the last five seconds. After the initial five seconds, the five position values are used to calculate the variance of the samples. The author found that removing an agent when their variance reaches the variance threshold of 0.1 or below was suitable. However, this method did not work for all cases consistently, with expectations such as agents which are moving some limbs but not changing their centre of position to update their variance [5].

Fitness evaluation

The fitness evaluations used were very straight forward. Firstly, the agents must pass the initial trial of the heuristic fitness evaluation of the first generation. The current implementation with a variance threshold of 0.1 and a position threshold of less than 0 suggests that 50-80% of agents do not make it passed the first generation depending on the environment presented. The gives the initial population a good set of initial genes that are more likely to be successful than random selection [4].

The primary fitness evaluation is the distance travelled by an agent in the positive x dimension in a specified amount of time known as the trial time. If an agent finishes its simulation, its score is taken and added to a vector of completed agents. When this list is filled with the number of agents specified by the population size, the current generation/epoch is complete. The agents are then sorted using a simple bubble sort to order them by their fitness score (distance travelled in the positive x direction). Once a generation is complete, statistics can be calculated on the population and recorded to a csv file.

Survivor mechanisms

The primary survivor selection mechanism follows a discrete distribution specified by the user. The current distribution follows a linear distribution which means the agents fitness score in linearly proportional to its survival rate. Firstly, a random number is selected from the user defined distribution. The value is used by a mapping function to index into the list of agents and remove the specified agent [4].

Another culling mechanism is to ensure the removal of agents who have existed for too long within the population. Removing older agents after a specified amount of time generally improves the optimisation process. The author found a life limit of 20 generations to be sufficient.

The implementation method of selection allows for any arbitrary discrete probability distribution which means that it could be made very complex and involve many selection mechanisms.

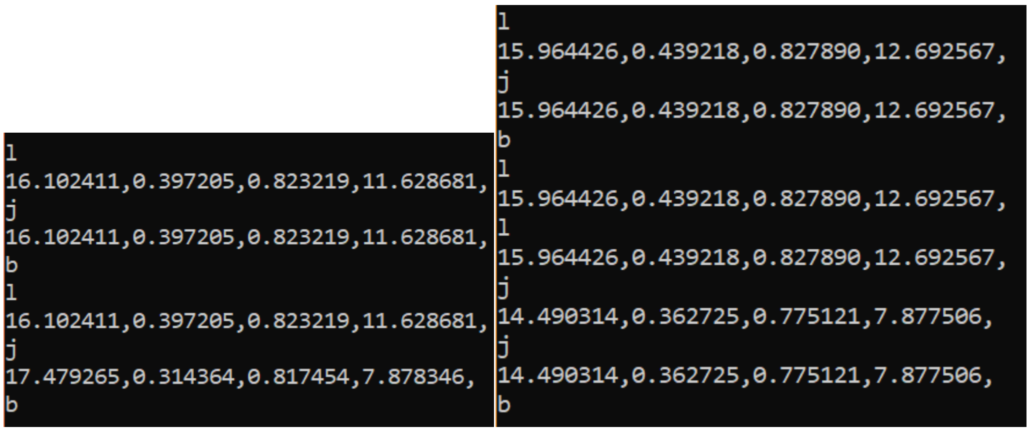
Crossover/Recombination

Bilateral crossover

The bilateral crossover is very simple to implement due to the data structures used to hold the agent information. As explained in section[], the AgentChromes struct is used at the midpoint of all agents and acts as their line of symmetry. Bilateral crossover is performed by swapping the LineChrome values, stored in AgentChromes, of one agent with the LineChrome values of another. Once crossover is complete the new genome and be constructed by the agent generator. This method is similar to the 'splicing' method seen in Karl Sim's 'evolving virtual creatures' as it is a physical crossover of two agents body composition [5].

Arithmetic mean crossover

Figure [30] – Agent a1 and a2

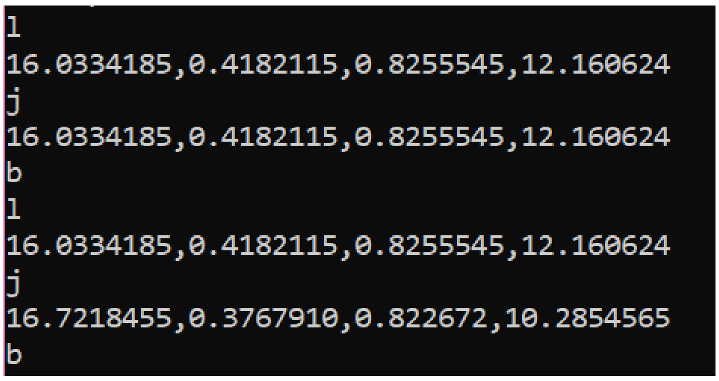


```
l
16.102411,0.397205,0.823219,11.628681,
j
16.102411,0.397205,0.823219,11.628681,
b
l
16.102411,0.397205,0.823219,11.628681,
j
17.479265,0.314364,0.817454,7.878346,
b
```

```
l
15.964426,0.439218,0.827890,12.692567,
j
15.964426,0.439218,0.827890,12.692567,
b
l
15.964426,0.439218,0.827890,12.692567,
l
15.964426,0.439218,0.827890,12.692567,
j
14.490314,0.362725,0.775121,7.877506,
j
14.490314,0.362725,0.775121,7.877506,
b
```

The arithmetic mean crossover method utilises the string representation of an agent's genome. Firstly, one the of agent's genomes is selected for its morphological structure, this is known as the primary parent and will become the new offspring agent's morphological structure. Next the genomes are partitioned by their respective left and right sub-tress. The left and right sub-trees are then compared respectively. When the parser encounters an instruction the instruction of the primary parent is copied to the new genome. When line data is encountered the arithmetic mean of the two corresponding attributes of the left and right sub-trees are calculated for four of the line attributes and added to the new genome. The new agent's left and right sub-trees are then concatenated to produce the final result.

Figure [31] -



```
l
16.0334185,0.4182115,0.8255545,12.160624
j
16.0334185,0.4182115,0.8255545,12.160624
b
l
16.0334185,0.4182115,0.8255545,12.160624
j
16.7218455,0.3767910,0.822672,10.2854565
b
```

The resulting offspring from agent a1 and a2 seen in figure[] is represented above. Notice the structure is the same as agent a1 but the line attribute values contain the mean of both a1 and a2. The new agent is then added to the list of current agents and will be constructed during the next epoch.

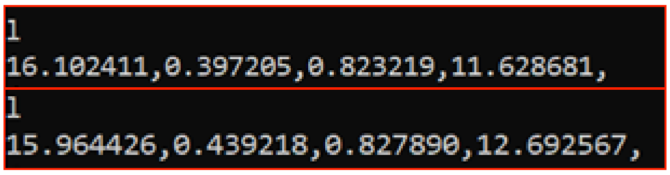
Uniform crossover

The uniform crossover operation is also performed on the string representation of the agent genome much like the arithmetic mean. For each new line instruction encountered a uniform buffer the same size as the number of attributes, which in this case is four. The buffer is filled with randomly generated Boolean values and is used to determine which agent passes on their genetic material. A true value will take the attribute from parent one while a false value will take it from parent two.

Figure [32] – Example of the uniform buffer

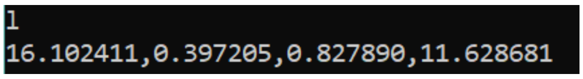
1	1	0	1

Figure [33] – The two genomes segments to be crossed over



```
l
16.102411,0.397205,0.823219,11.628681,
l
15.964426,0.439218,0.827890,12.692567,
```

Figure [34] – The resulting genome



```
l
16.102411,0.397205,0.827890,11.628681
```

Notice how the new genome has taken the first, second and last attribute values from the first agent and just the third from the second agent.

Mutation

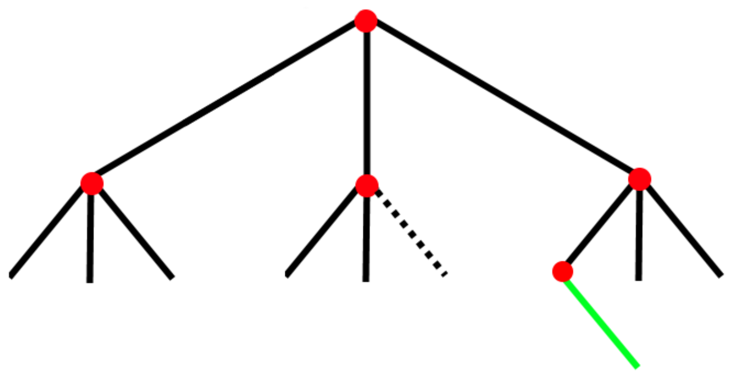
Scaled mutation

Scaled mutations are mutations that scales the attribute values of lines by a predefined mutation rate.

Morphological mutations

Morphological mutations are mutations that change the phenotype or physical structure of the agent directly. EvoSim includes both adding and subtracting lines from agent genomes as a mutation mechanism. First, either the left or right sub-tree is selected for mutation,

Figure [35] – diagram of potential morphological mutation



The mutation algorithm starts by checking if the current LineChrome is a leaf node within a tree. If it is not, then the vector of LineChromes is inspected for its size. This size value is used to generate a random integer which selects a LineChrome to recursively follow. The function is recursively called until a leaf node has been found. Once found, the line has a 50% chance to have an addition line appended to the end (green) or a 50% for the current line to be deleted entirely (dotted). This allows the complexity of the agent to change with the complexity of the problem space.

Initialise new population

After recombination the agent population should be identical to the initial population size. Both the new offspring and old surviving agents are added into the list of agents to be tested in the next epoch of the simulation.

Technical parameters

Population size is the quantity of agents tested in each epoch and remains constant throughout the simulation.

Trial time is the amount of time each agent is tested for.

Cull rate is the percentage of the population that is removed each epoch.

Max age is the maximum life of any agent.

Mutation chance is the probability that a mutation will occur

Mutation rate is the amount that genes are mutated by given mutation occurs

7.0 TESTING

Unit testing

Odingine

Test	**Outcome**	
SDL		
Create a blank SDL window	Window is created with specified background colour	🟩
SDL buffers swap correctly	Buffers prevent jittering when used	🟩
Input manager		
Press key	Key press is activated	🟩
Release key	Release key is activated	🟩
Is key down	Is Key down is activated	🟩
Was key down	Was Key down is activated	🟩
Get/Set mouse coordinates	Mouse coordinates are set and gotten correctly	🟩
Camera		
Setting position	Position of camera is moved to the correct location	🟩
Setting scale	Scale of camera moves to the correct scale	🟩
Error handling		
Error thrown when wrapped	Errors are correctly thrown with corresponding error code	🟩
Fps controller		
Correct fps calculations	Fps is calculated correctly	🟩

EvoSim

UI		
Move left and right	Camera can be moved left and right by 5 meters	🟩
Zoom in and out	Camera can be moved in and out	🟩
Track agent	The agent is not continuously tracked by the camera. (Solved)	🟥
Render/un-Render simulation	The simulation can be rendered and un-rendered during simulation	🟩
Display fps	The fps is correctly displayed in the console	🟩
Add/Remove agents	Agents can be added and removed	🟩
Display time	The current trial time second is successfully displayed in the console	🟩
Maximise trial time	The trial time is set to a maximum (Sandbox mode)	🟩

Box2D		
create a Box2D world	The Box2D world is successfully initialised with correct parameters	🟩
Initialise camera in world	The camera is correctly initialised into the world and positioned at 0,0	🟩
Log agent bodies	All the bodies in the world are displayed correctly in the log	🟩
Flat terrain created	The flat terrain is spawned into the b2world	🟩
Rough terrain created	The rough terrain is spawned into the b2world	🟩
Hills terrain created	The hills terrain is spawned into the b2world	🟩
Water terrain created	The water terrain is not implemented	🟥
Save terrain	The terrain can be saved successfully	🟩
Load terrain	The terrain can be loaded successfully	🟩
World renders successfully	The world renders correctly to SDL window	🟩

Agents		
Generate agent genome	An agent's genome is created successfully	🟩
Create agent in Box2D	Genome added to agent which is added to the world	🟩
Tree to string genome conversion	Tree structure to string successful	🟩
String to tree genome conversion	String to tree structure successful Tested on same tree structure as above and generated correctly	🟩
Output string genome to console	Genome displayed to the user	🟩
Save string representation of genome	String saved to file	🟩
Load string representation of genome	String loaded from file	🟩

Rendering		
Initialise shaders	Shaders initialised correctly	🟩

Evolutionary algorithm		
Create initial population of agents	Initial population created and tested correctly	🟩
Sort ages correctly	Agents sorted by fitness metric correctly	🟩
Survival selection distribution check	Distribution of survival selection test matches desired distribution	🟩
Mate selection distribution check	Distribution of mate selection test matches desired distribution	🟩
Bilateral crossover correct output	Agents bilaterally crossed over and output to screen	🟩
Mean crossover correct output	Mean found of two agents and output to screen	🟩
Uniform crossover correct output	Uniform crossover applied correctly output to screen	🟩

Morphological mutation	Morphological mutation observed in simulation	🟩
Real value mutation	Real value mutation output new genome	🟩
Initialisation of new population	New population added to vector correctly	🟩
Correct calculations of epoch statistics	Epoch stats checked by hand	🟩

Performance testing

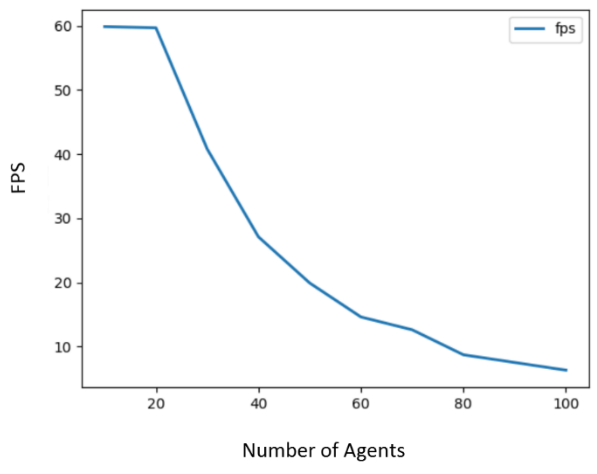
It is important to perform performance testing on the program to test the capable limits of the software. This includes the amount of CPU usage is used by the program during run time and the stability of the Box2D simulation. This is to ensure that the program does not consume to many resources before being distributed to the shareholders. It is important to ensure a high fps to maintain stability within the simulation.

Tests were conducted by generating agents into the world in increments of 10 while the fps and CPU performance were measured. This information could be used to either limit amount of resources (number of agents) accessible to the user. For CPU testing Visual studio performance profiling tools are used.

Box2D stability test

Fps is maxed at 60 fps

Agent quantity	**Average fps over 30 seconds**
10	**59.9**
20	**59.7**
30	**40.8**
40	**27.1**
50	**19.9**
60	**14.6**
70	**12.6**
80	**8.7**
90	**7.5**
100	**6.3**

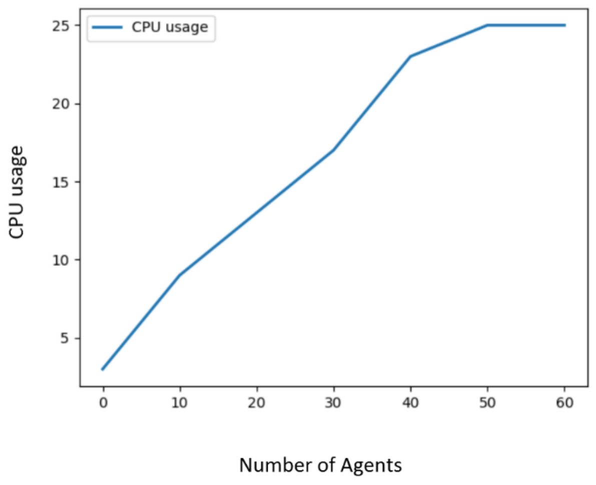


It appears that the simulation can remain stable at 60 fps with less than 20 agents. However, after this the performance drops substantially with 30 agents to ~40 fps while also fluctuating fairly frequently. The performance continues to drop as agents are increased, until it begins to plateau at 5 fps with 100 agents.

This data suggests that the algorithm could be parallelised and remain stable with less than 20 agents. This means tests can be processed in batch, with the results of each set of 20 agents accumulated into one final results array. This allows population sizes to remain scalable but also potentially increase the test rate of agents by 20 times.

CPU performance

Agent quantity	CPU usage
0	3.1
10	9.2
20	13.1
30	16.9
40	22.7
50	25.0
60	25.0



The program is limited to running on one core and therefore is limited to 25% of the CPU. The CPU performance scales linearly with the number of agents in the simulation and maxes out the single core with 50 agents. The program is extremely stable when agents are test in linear and users should never experience any performance drops. This evidence also suggest that 20 agents is a reasonable maximum number of agents to be simulated at once.

Acceptance testing

The acceptance testing section verifies whether the development solution fits the criteria outlined by the requirement specification in section []. The objective of these tests is to equate the functionality of the simulation with the existing criteria specification.

Acceptance Criteria	Result
Main criteria	
The program should use/make a suitably realistic physics engine to simulate the motions of agents in an environment.	The author ended up using Box2D for the physics simulation which reduce the complexity of the project but allowed the author to concentrate on the evolutionary algorithm which was a much more realistic goal.
A backend library should be created to deal with user input, error handling and any other management functionalities.	The final backend library was a success and provided the useful features described.
The program should provide a mechanism for visualising the evolutionary algorithm in progress. This will be used to verify the creation of agents in Box2D, validate the evolutionary algorithm and provide an interactive interface for the user.	SDL and OpenGL were used along with the modified b2Draw class (DebugDraw) to render the agents and environment to the screen. The GUI allowed the author to check the agent creation and resulting offspring from

	two agents and also track the progress of the algorithm.
The evolutionary algorithm must generate a variety of agent morphologies that satisfy the requirements of the environment.	A number of interesting agent morphologies were discovered through optimisation. Some seemed intuitive while others took a very unexpected approach.
Agent criteria	
The agent generator must create agents from a sufficiently complex agent solution space where many agent morphologies are theoretically possible.	The agent generator used a tree structure to form the agents. While total amount of morphological complexity was not that high, the task does not require it to be and often agent that were too complex were removed from the population quickly. There was also plenty of optimisation that could take place on the attributes of each line.
The morphology must be infinitely expansive through evolution	The morphology is technically infinitely expanding but the agents tended not to use the additional complexity. Perhaps with different tasks it would be utilised.
Agents should be constructed of lines and/or wheels that use rotating joints in order to manoeuvre through the environment.	Lines were used however wheels were dropped during development as they were generally to good.
Agents should use a simple ANN to control their movements	ANN's were not added in the final program due to time limitations however this is a future development.
The ANN should be optimised via the EA.	N/A
An agent's genome must be saveable and loadable to/from a file.	An agent's genome can successfully be converted to a string that can be saved and reloaded.
Evolutionary algorithm criteria	
A variety of fitness measures should be used to demonstrate the versatility of evolutionary algorithms.	Unfortunately, only the one fitness measure (which agent can progress furthest to the right) was used due to time limitations. However, future tests would not be difficult as the infrastructure is in place.
A number of different environments should be created for agents to optimise towards. E.g. Flat land terrain, Non-flat land terrain, Water	A number of different terrain types were generated and tested on the agents. It was found different terrains required different morphologies.

	However, the water terrain was not included.
Terrains must be saveable and loadable	It is possible to save and load terrains from file.
The evolutionary algorithm must use a variety of crossover and mutation mechanisms.	Several crossover and mutation mechanism were implemented into the final design and compared in the results.
Statistics must be recorded about each epoch which can then be displayed on graphs.	Statistics are recorded each epoch creating a time series of data. This data is then plotted in python for the visual representation.

8.0 RESULTS

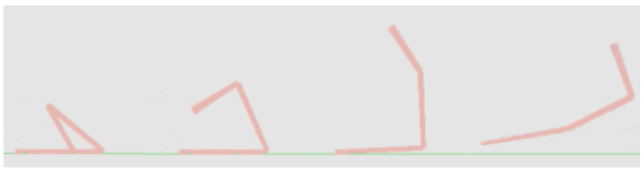
How agents adapted to their environment

Example morphologies

This section will compare the common morphologies of agents and how they differ between environments.

Thrower agent

Figure [36] – thrower agent



This thrower morphology traverses the terrain by launching itself into the air. It appears to use the heavier front limbs to change its centre of mass and propel itself through the air.

Figure [37] – The optimisation graph for figure [36]

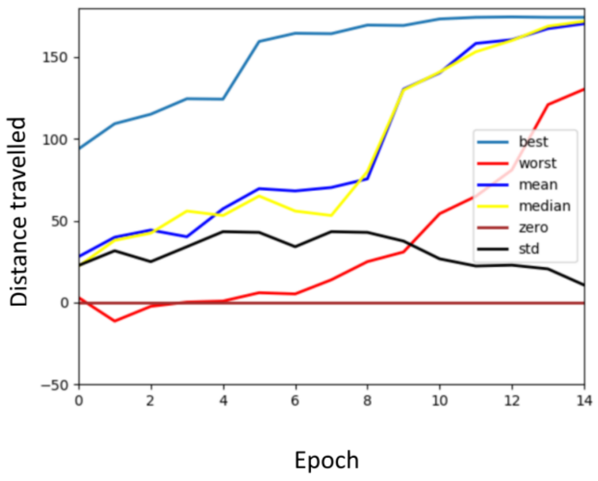
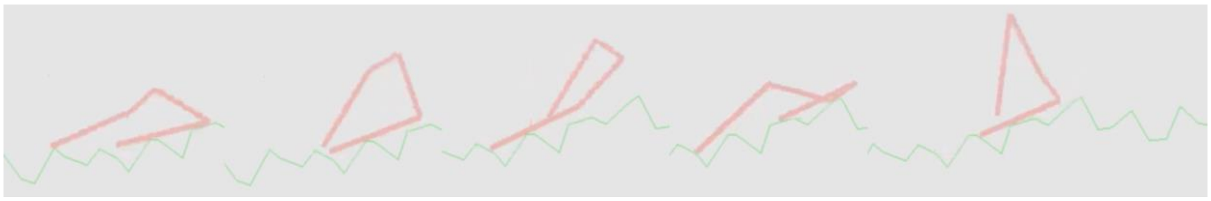


Figure [37] shows a time series of the range, mean, median and standard deviation across epochs. This graph represents to optimisation process for the thrower agent. The best agent represents the agent that achieved the highest fitness score in an epoch. The gradient of the best agent is larger during the initial epochs, demonstrating that the solution space is large, and many solutions are being tested. During epoch five the simulation finds a solution than previous epochs. This increase in fitness gave the agent a more likely chance of being selected for crossover, resulting in more offspring from this agent. A couple of epochs later the mean and median begin to converge on the best agent, plateauing around 175 meters at epoch fourteen. This suggests that the agent solution produced in epoch five was very successful and outcompeted other agent types to dominate the gene pool. After epoch six, further small optimisations were made on the best agent that improved it very slightly.

Crawler agent

Figure [38] – crawler agent



This crawler morphology takes advantage of the ridges and depressions in the terrain with its long appendages which are slightly offset from one another. This offset allows the limbs to get caught in the depressions and uses them as ‘steps’ to propel the other half of its body onto the next step.

Figure [39] – optimisation graph for figure [28]

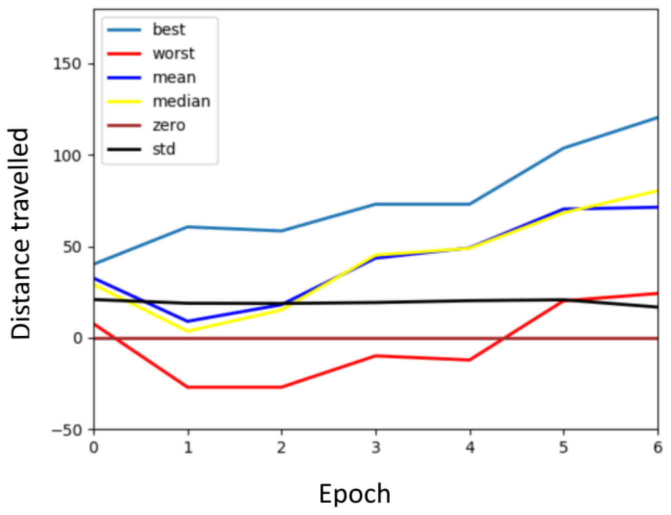
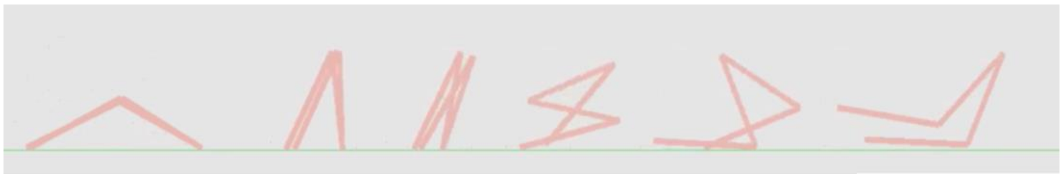


Figure [] shows the optimisation process for a population of agents on rough terrain. The final solution optimised after only six epochs and is three times better than the first. The first thing to notice is the initial average fitness of the population is much lower than those optimised on the flat terrain. The author suspects this is because the rough terrain is much harder to initially traverse with many ridges to get stuck between. This may be indicated by the score of the worst agent being much lower than in figure [].

Bilateral Symmetry

Bilateral agents were tested by enforcing the duplication of one half of the agent and overwriting the other half. These agents are then an exact copy down the middle and now contains a line of symmetry.

Figure [40] – Bilateral agent



The evidence was not clear that bilateral agents performed any better than non-bilateral ones, however they were still able to produce effective and efficient solutions. Because the symmetrical halves are identical they tended to work together using the approach. This can be seen in figure [40] above.

Although not explicitly a bilateral agent, notice how the crawler agent in figure [38] adopted an almost symmetrical morphology with a slight offset to one of the upper limbs. The author wonders if this has any meaningful significance.

Convergent evolution

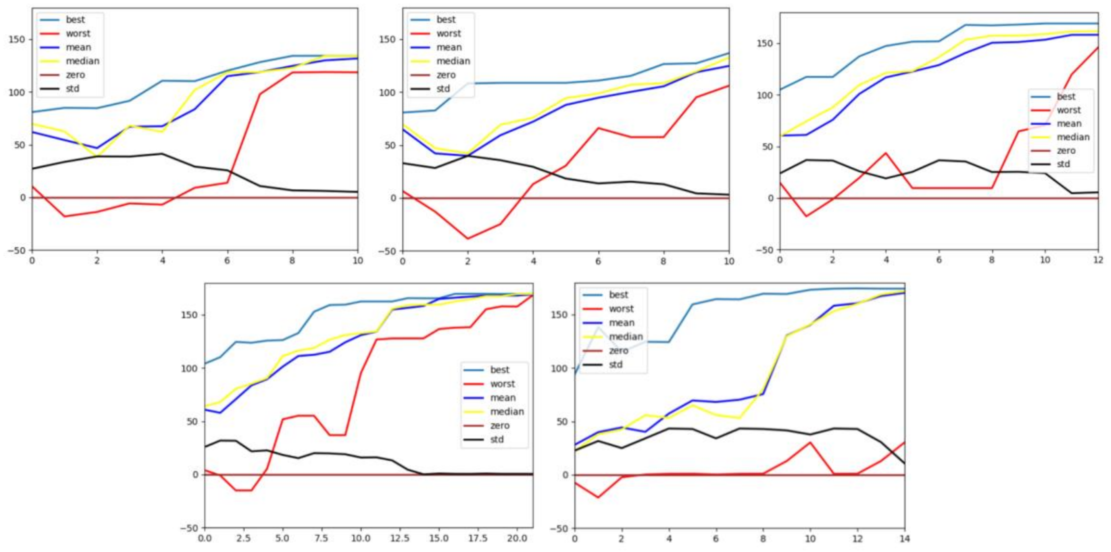
Convergent evolution is the independent evolution of similar features or morphologies in species of separate lineages. It creates analogous structures that possess similar form or function but were not present in the last common ancestor between the two organisms. A famous example of convergent evolution is the evolution of the wing, which has been independently evolved by insects, birds and pterosaurs. This occurs due organisms experiencing similar selective pressures, similar advantages to exploiting the environment, constraints enforced by nature and physics on body shape/structure. When adapting to similar niches, organisms will experience selection for a lot of the same traits and converge on similar phenotypes [14].

Within EvoSim convergent evolution was witnessed a number of times over many simulations. This was observed when the same environments were retested with a different initial population. For the flat terrain the Thrower morphology was relatively common, along with a four-limbed agent which used its two front limbs to drag itself across the floor. These two solutions were very effective which might suggest why they appears so often. In Karl Sims' paper, he finds that a population of agents does converge towards homogeneity however observed no convergent evolution as he described each simulation to produce very different solutions [5].

The effects of population size

The following tests were conducted using a 50% cull rate on Flat terrain. Flat terrain was chosen as it controls for the most extraneous variables and provides the exact same environment for them to optimise towards.

Figure [41] – The effect of population size on convergence



The best score achieved by the agents is not relevant due to the fact that each simulation has different variance in population and therefore can never be directly compared. However, one observation that can be made is that, there appears to be a general trend for the variance of the population score to converge quicker the smaller the population is. The variance of the population score is not the same as the variance of the population genome. However, from visual observations it is clear that once the score variance of the population approaches zero, many of the agents share the same morphological structure. This suggests that the variance or diversity of the agent morphology genes eventually reach homogeneity and implies genetic drift has occurred within the population. It also appears that as the variance of the population score decreases the best agent score begins to plateau. While these observations do not confirm the that an increased population increases the time for convergence there does appear to be a trend. It was also discovered, when evolving a population of agents, it is important to strike a balance in the variance of these genes to ensure that the population does not converge too quickly.

Changing the fitness metric

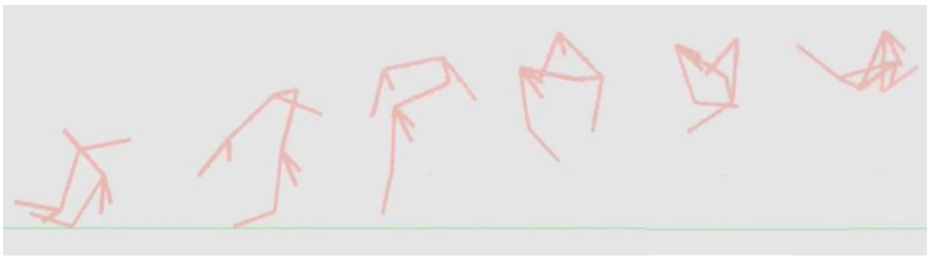
One of the primary benefits of evolutionary strategies are their ability to adapt their solutions based on the fitness metric. The main fitness metric used in EvoSim is evaluating an agent's positive x position. This however, could be changed to be the positive y position. This would cause the optimisation process to produce agents which attempt to 'jump' rather than move horizontally. Alternatively, agents could be optimised to maintain the highest possible y value with the fittest metric being a combination of maintaining the y value (measure the variance of the agent's movement) while also attempting to make it as large as possible. This fitness metric may encourage agents to develop stability.

Jumping agents

The fitness evaluation was altered to optimise for agents using their highest y value achieved within the trial time as the fitness metric.

When optimising to achieve jumping-like agents the maximum torque of the revolute joints was increased to 10000. This gives their limbs more action potential and a more spring-like movement as they propel themselves off the environment. A torque of 6000 used in the walking tests resulted in underwhelming results and was often dominated by agents who optimised for physical size rather than jumping.

Figure [42] – Jumping agent



After the changes were made to the revolute joints the optimisation was attempted again and produced the agent seen in figure [42]. The agent appears to have a long body with multiple smaller appendages attached. First, the agent begins to coil up and then quickly expand, using its long limb to propel itself into the air. Notice how the agent straightens out taking full advantage of the entire length of the limb. Next the agent begins to retract the limb and coil up in mid-air. The author suspects this is because retracting the limb drastically improves the centre of position calculation made which is used for the fitness score of the agent.

The effects of cull percentage

Figure [43] -

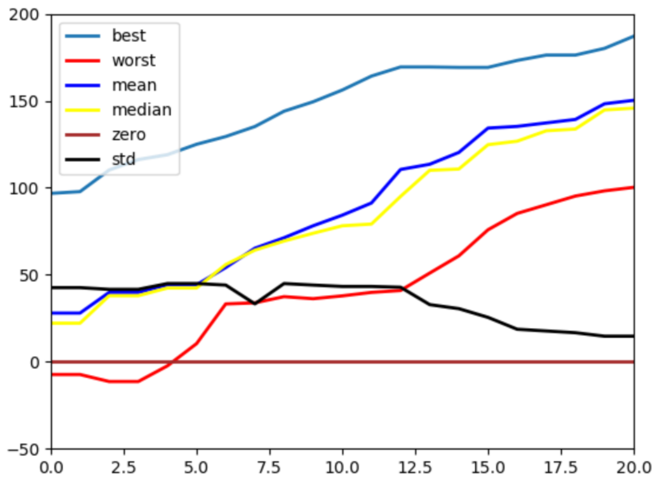


Figure [43] shows the optimisation process from an example of an agent population with a cull percentage of 0.2 with a population size of 60. The graph shows that, compared to the tests conducted with a cull percentage of 0.5 the optimisation process is much slower. This is consistent with what the author theorised as there will be less new agents tested each epoch. Notice how the gradient for the best agent score is much lower compared to tests with a 0.5 cull rate. Also notice how the optimal agent produce is better than previous tests. However, it is not known if this is due to the cull rate change. While this result cannot be considered valid in isolation many tests were conducted and found similar results each test.

Generalisation of optimised agents

Post-trial time generalisation

These tests are constructed to test the generalisability of the agent solutions. One way of measuring the agent generalisability is to measure the performance of an agent after the initial trial time, which the agent is not optimised for. For reference, most of the simulations used a trial time of 30 seconds.

Agents optimised for the flat terrain generalised well after the trial time expired. This is theorised to be because the environment is very simplistic. Once an agent develops a mechanism for propelling itself forward those same actions work for the entire length of the environment. However, it is much harder for agents to find a general navigation solution when the terrain in full of ridges and holes for the agents to get stuck in. For an agent optimised on rough terrain, after the 30 seconds agents would often make mistakes but on the whole still travel in the correct direction.

Environmental generalisation

Agents optimised in more complex environments (rough terrain) generally generalised better to simpler environments (flat terrain) than those optimised in simpler environments did to more complex ones. The first thing to note is that the fitness metric was the same for both agents and therefore both already had a bias moving in the correct direction. The flat terrain offers no resistance to the agent whereas the rough one does. The agent optimised for the rough terrain uses a technique that takes advantages of the ridge and depressions in the environment. While much lower its technique is still capable of navigating the flat terrain that offers no resistance. The agent optimised for the flat terrain, while it does travel in the correct direction, could not account for the ridges.

9.0 DISCUSSION AND REFLECTIONS

When reflecting on the entire project, the author believes that the entire independent project went moderately despite the project being an overall success. This I because due to a few issues the author experienced during the development process.

The first of which is setting more achievable goals. Like all students, the author saw the dissertation as an opportunity to demonstrate their ability and passion for computer science. However, this led to unrealistic expectations of oneself and resulted in a lot time wasted on perusing these goals. This included the partial development of the physics engine. A lot of time was spent reading 'Understanding physics engines' by Ian Millington. While this did provide some insight into how Box2D works, the knowledge was much more in-depth and consisted of implementation details. This initial development went as far as producing a 3D rendering engine which can be seen in the appendix.

The author is happy with the final product but found with the limited time constraints, they did not implement all of the features they had desired. This can be seen in section 11 which contains a large segment of future improvements. The future work has been outlined in detail as these concepts have been considered for a long time but were unfortunately not implemented.

The author also feels that they did not produce the product to a finished standard. This was mainly due to the fact the author found it far more interesting to experiment and test the evolutionary algorithm and simulation under new conditions rather than making the UI cleaner or adding a menu and splash screen.

The author is pleased with overcoming many of the struggles that occurred during the development process. They are most happy with the results gathered from the evolutionary algorithm experiments, the construction of the agents using a spanning tree along with the parser created to perform crossover and mutations, and the creation of the backend library which will definitely be used in future projects.

10.0 SOCIAL, LEGAL, ETHICAL ISSUES AND HEALTH CONCERNS

When developing the project initiation document, a risk assessment was carried out to identify any potential health and safety risks. This section will examine the potential risks in more detail and relate it to social, legal and ethical issue that may arise.

The first hazard is the risk of eye strain or damage due to the blue light from the LCD screen. This light can have disruptive effects on sleep which could ultimately result in fatigue. One way to mitigate this is to use windows inbuild ‘Night light’ feature which removed the harmful blue light.

Another potential hazard is working alone, late hours or extended periods of time that may be see short-term benefits but could potentially cause stress or fatigue if consistently performed. One way of avoiding this to set strict working hours during the day with a maximum of 10 hours worked. During these 10 hours, 15 minutes break will be taken every two hours to prevent tiredness.

One more hazard associated with the development of the project is the risk involved with using and transporting a laptop on university campus. The laptop will be the primary device used to develop the program. The risk involved is the potential threat of either losing or having it stolen. Caution must be taken when transporting the laptop. One mitigation to this is to utilise a cloud storage service such GitHub or Google Drive to create a back up of the progress and any related work.

There is also a concern from a legal perspective, with the misuse or theft of the project from GitHub. To prevent this, the GitHub repository will be made private which removes it from searches and ensuring only the developer has access to it.

11.0 CONCLUSION AND FUTURE IMPROVEMENTS

Conclusion

The development of the project was an overall success in producing a frame work to implement an evolutionary algorithm as an optimisation technique. This was decided as the project acceptance criteria was almost complete fulfilled along with some minor additions that were incorporated during implementation.

The first primary objective fulfilled was creating the backend library for EvoSim. The objective was considered a success as the library provided all of the required functionality that was needed.

We found that it is possible for the simulation to generate many different successful solutions to the desired problems. This was demonstrated by multiple agents finding unique solutions to navigating flat terrain, navigating rough terrain and jumping the highest.

The simulation was also used to test some mechanisms of evolution such as natural selection which was seen through the process of agent selection, convergent evolution which was observed after running numerous simulations and detecting a pattern in morphological structure depending on the environment. And finally, genetic drift which was seen when the diversity in genetic material of a population converges towards homogeneity.

The primary objective that stipulates that a physics simulation is to be created can be considered unfulfilled as the final product did not contain a physics engine created by the author but instead uses a popular and well document physics library.

During project development, the author made an effort to stay in line with the objectives of the project. This is also represented in the design phase. The original design plans for the evolutionary algorithm and agent design were successfully implemented according to the design, making the implementation considerably easier. This shows consistency in the execution of the project and shows that the author had a clear targeted plan of events.

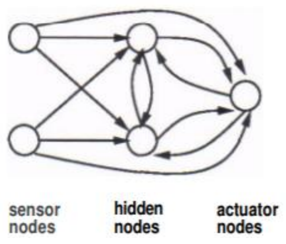
Future improvements

Artificial neural networks as behaviour mechanisms

Research

In ‘Evolving Virtual Creatures’ Karl Sims represents an agent’s brain with something that resembles a dataflow computer program. Micheil van de Panne takes a slightly different approach with Sensor-Actuator Networks (SANs) which are comparable to feed forward networks. The topology of the SAN is given bellow,

Figure [44] – Topology of SAN



The network consists of nodes and unidirectional weighted connections. The weights of the connections can take on values in a fixed range. In our implementation we choose integer values in the range [-2,2] [9].

Figure [45] – Function of a SAN node for hidden and actuator nodes

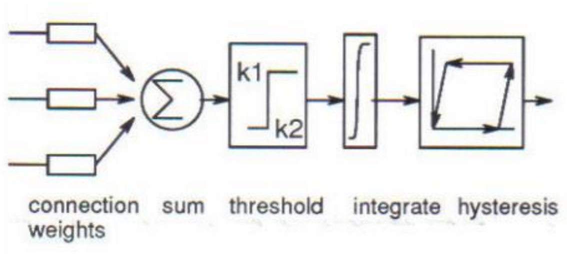


Figure [45] shows the hidden and actuator nodes function. A node sums the weighted inputs and outputs and fires if the sum is positive. This is a function similar to those performed in neural networks. It is important, however, for the controller to be a dynamical system on its own [9].

Implementing in EvoSim

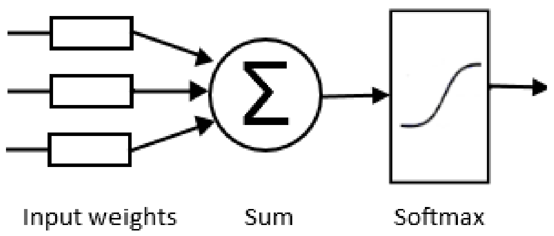
Each limb would contain a ‘gripping’ fixture which contains a sensor. Fixtures have the ability to also become sensors using the isSensor() member function. A contact listener is used to receives a signal from the sensor fixture when contact is made with a surface. These signals could be used as inputs neurons for the ANN. Another agent sensor that could be utilised as inputs could be the current direction of rotation for a specific revolute joint. These two would work well due to the fact they are both binary decisions and therefore normalised.

The output neurons for the sensor fixtures would control the amount of friction exerted between the ‘gripping’ limb and the ground. One way to achieve the grabbing effect could be simulated by increasing the friction on the fixture to very high number creating a large amount of resistance and therefore holding it in place. The revolute rotation angle could also be changed giving the agent much more precise control over how its limbs move.

ANN architecture

The ANN would require a very large number of input neurons. The number of input neurons would be equal to the number of fixture sensors (leaf limbs) plus the number of joints between limbs. This would be in the range of ~(4 – 20) depending on the size of the agent. The network will contain one hidden layer with the number of hidden neurons equal to the input and use softmax activation functions.

Figure [46] – Hidden layer neuron



The number of output neurons will be equal to the number of inputs and will utilise the sigmoidal activation function to allow binary outputs from the network. Each output will correspond to a fixture member function that controls friction or a revolute joint movement direction. If an output neuron is 1 then friction is turned on or the joint changed direction. If an output neuron is 0 then all states remain the same.

The weights of the network would also use an evolutionary approach to optimisation with the weights of the network being altered through the same processes of crossover and mutation. Another method for training the network could be to generate a relatively optimised agent through the evolutionary algorithm and then train it in the environment using the binary cross entropy loss.

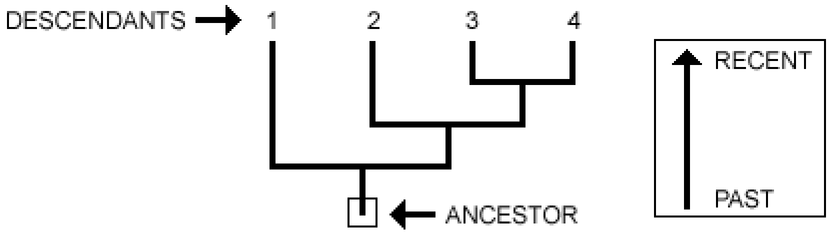
Implementing Speciation

Another interesting concept the author would like to investigate is the effects of speciation on a population of agents.

Defining a species

A species is generally considered to be a group of individuals that interbreed in nature which therefore makes a species the biggest gene pool possible under natural conditions. The definition for a species is difficult to defined in practise as the categorical boundaries of species put in place by humans does not always capture the many ways organisms can diverge from another. An example of this are organisms that reproduce asexually or hybrid animals like mules and ligers.

Figure [47] - speciation



Mechanism for assessing genomes

Phylogeny reconstruction offers a mechanism to describe evolutionary relationships in terms of relative recency of common ancestry. The relationships are represented as a branching diagram or tree joined by nodes and leading to terminals. The method for comparing the diagrams of different organisms involves following the strict consensus parsimony tree and the alignment or incongruence of topologies [13].

Implementing gene comparison in EvoSim

Fortunately, due to the simplicity of genomes used in EvoSim, a technique for assessing the differences in genomes is much easier than phylogeny reconstruction but the sentiment is still the same. This could be done in EvoSim by creating a similarity score. The similarity score would be composed of two components, morphological similarity and genetic similarity. Morphological similarity will be assessed by comparing the physical structure of the agent, for instance, how many limbs does the agent have and are they in a similar configuration. Genetic similarity will be assessed by creating an n dimensional vector space with n being the number of limb attributes. Once the vector space is created the Pythagorean or Mahalanobis distance can be calculated between the two vectors to find how close they are. These two components will be used to construct the similarity score and will be normalised between 0 and 1. Agents would then select agents who fit between a range, for instance 0.5 – 0.8. The reason for capping it at 0.8 is to prevent consanguinity and the resulting genetic drift.

Mate selection with speciation

Calculating genetic difference could provide interesting mechanisms for mate selection. If each agent could possess the ability to assess another agent’s performance based on their genetic material, then it would possible for the agents to define which agents are suitable for mate selection and which aren’t. This is a much more complex mechanism for determining which agents get to breed compared to a simple pin wheel distribution based on fitness. The addition of speciation may provide the ability for the population to simultaneously search many potential solutions.

When an organism undergoes speciation, it can no longer breed with members of another species (generally). This therefore suggests that organisms base, at least in some part, their mate selection preferences on the similarity of their own phenotype or genotype to the other organisms. This implies, one mechanism for determining mate suitability could be the genetic similarity of organisms. However, this could potentially increase the speed at which genetic drift occurs as agents each time similar agents breed the variance in genetic material decreases because the population size stays the same. One method for slowing this down would be to prevent agents which are too similar from breeding. This could be compared to in-breeding in nature which causes extreme genetic drift. Another requirement would be to ensure an increased population size, for example around 500, to ensure there are enough agents similar in form.

Parallelising the algorithm

The last additional feature discussed is the addition of paralleling the algorithm. The current implementation runs each agent in serial with a trial time of 30 seconds. This was chosen for consistency and stability within the simulation. However, one run of the program with an agent population of 100 and run for 10 epochs would take over 8 hours. It is clear that the algorithm could benefit from parallelisation.

Initially the author attempted to run an entire epoch in one time-trial. A population of 60 agents were created and simultaneously tested. This produce very inconsistent results and is supported by evidence gained from testing the algorithm in section []. However, due to the accuracy requirements of this program even testing a handful of agents at once can produce undesirable inconsistencies. This is because, with the addition of more agents the rendering loop takes longer and therefore causes a longer delay.

One method for solving this is to fix the time step using a method outlined in a well-known web article by Glenn Fielder titled, 'Fix Your Timestep'. An accumulator is used to store the time spent by the renderer, then the simulation is stepped by a fixed amount until they sync as best as possible. A small amount of time might be left over which causes visual stuttering. Glenn Fielder suggests that an entity's transform should be interpolated between the previous and current physics state based on how much time is left in the accumulator.

The author was successfully able to implement the fixed timestep alteration and achieved more consistent behaviour in the simulation but was not able to implement the additional interpolation to solve the visual stuttering. The stuttering was very extreme and made the simulation unwatchable and was therefore removed. The author would like to reattempt the implementation with more time available.

REFERENCES

[1] Evolution strategies as a scalable alternative to reinforcement learning - OpenAI

[2] Producing flexible behaviours in simulated environments - Deepmind

[3] Understanding physics engines – Ian Millington

[4] Introduction to Evolutionary Algorithms – Xinjie Yu, Mitsuo Gen

[5] 'Evolving Virtual Creatures' – Karl Sims

[6] Box2D 2.3.0 Manual

[7] Introduction to game physics with Box2D

[8] Artificial Life – Christopher G. Langton

[9 Sensor-Actuator Networks - Micheil van de Panne

[10] The Selfish Gene – Richard Dawkins

[11] A Study of Genetic Algorithm and Crossover Techniques - Ashima Malik

[12] Genetic Algorithm Parameters Effect on the Optimal Structural Design Search - Z. El Maskaoui1

[13] A step by step guide to phylogeny reconstruction C. Jill Harrison and Jane A. Langdale

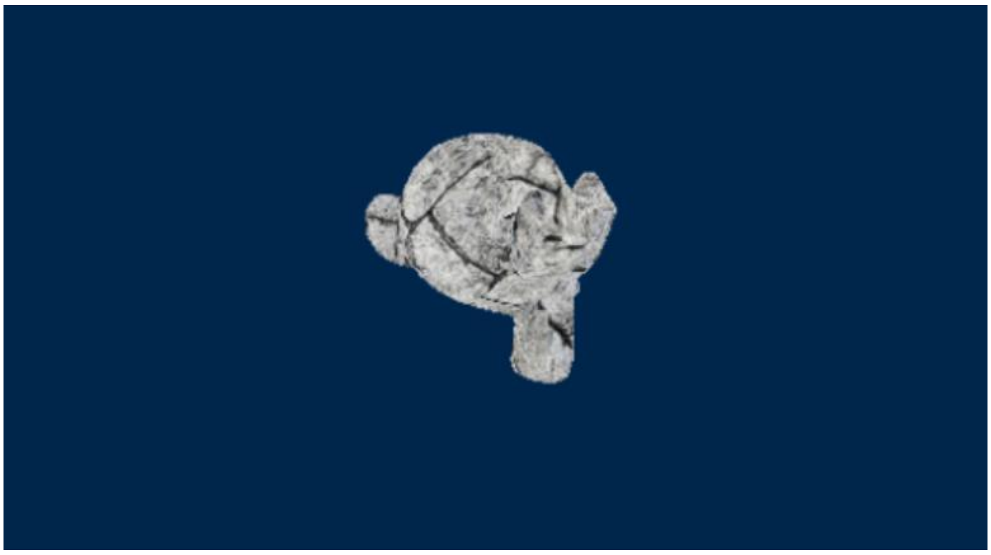
[14] https://en.wikipedia.org/wiki/Convergent_evolution

[15] https://en.wikipedia.org/wiki/Hyperplane_separation_theorem

[16] Mühlenbein, H. and Schlierkamp-Voosen, D.: Predictive Models for the Breeder Genetic Algorithm: I. Continuous Parameter Optimization. Evolutionary Computation,

APPENDICES

Learning OpenGL



During the development process it was vital for the author to learn how OpenGL worked. Online resources and tutorials were used to develop a 3D rendering engine in C++. This was when development was still orientated around producing a 3D physics engine for the simulation.

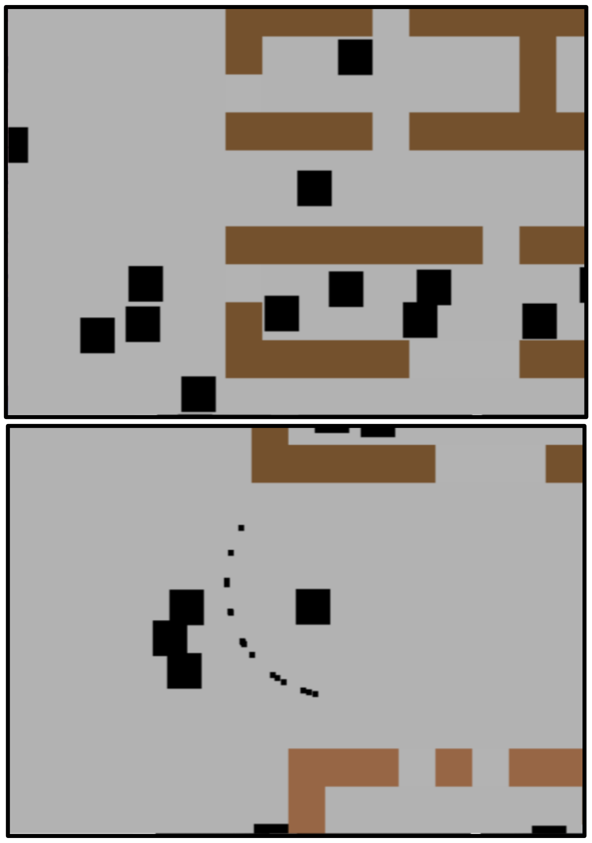
The source code can be found on GitHub: https://github.com/AlexanderSmall

Overview > Physics-engine

Game side-project

Test game created using the O

dingine library:



Objective of the game is stay alive as long as possible while remaining untouched from the NPCS. If the player touches another NPC then the game is over. Players gain score by killing NPCs. The world contains collision with walls and other NPCs. The game allows player to attack NPCs using two weapon types that have different attack animations.

The source code can be found on GitHub: https://github.com/AlexanderSmall

Overview > EvoSim > evolutionsim1 > ZombieGame

End of Report

Individual Project (CS3IP16)

Department of Computer Science
University of Reading

Project Initiation Document

PID Sign-Off

Student No.	**25012484**
Student Name	**Alexander Small**
Email	**xf012484@reading.ac.uk**
Degree programme (BSc CS/BSc IT)	**BSc**
Supervisor Name	**James Anderson**
Supervisor Signature	
Date	**02/10/18**

SECTION 1 – General Information

Project Identification

1.1	**Project ID** (as in handbook)
	450
1.2	**Project Title**
	Simulating evolution by natural selection in a 3D physics engine
1.3	**Briefly describe the main purpose of the project in no more than 25 words**
	Develop a physics/rendering engine that simulates ‘organisms’ in a 3D environment and evolves them via genetic algorithms. Will find unique and novel solutions to problems such as moving and swimming.

Student Identification

1.4	**Student Name(s), Course, Email address(s)** e.g. Anne Other, BSc CS, a.other@student.reading.ac.uk
	Alexander Small

Supervisor Identification

1.5	**Primary Supervisor Name, Email address** e.g. Prof Anne Other, a.other@reading.ac.uk
	Dr James Anderson, j.anderson@reading.ac.uk
1.6	**Secondary Supervisor Name, Email address** Only fill in this section if a secondary supervisor has been assigned to your project

Company Partner (only complete if there is a company involved)

1.7	**Company Name**
1.8	**Company Address**
1.9	**Name, email and phone number of Company Supervisor or Primary Contact**

SECTION 2 – Project Description

2.1 **Summarise the background research for the project in about 400 words. You must include references in this section but don't count them in the word count.**

Background research began by planning what I wanted to achieve from the project and then roughly outlining the sub steps that would be required to complete it. It was decided I would investigate evolution in an artificial setting using many agents interacting simultaneously as selective pressure. Agents will have a simplistic 'nervous system' or 'brain' constructed using an ANN (Artificial neural network), allowing them to make decisions. The ANN and physical body will 'evolve' using genetic algorithms increasing in complexity over time and solving novel problems presented by the user.

During research it became clear that the first thing that would need to be created is a physics engine. In order to gain a high-level understanding reference [1] was used. Creating a semi-realistic environment for the agents to interact with allows for more physically plausible motion and interactions between agents [3]. In order to create the engine an understanding of linear algebra (predominantly vector mathematics) and Newtonian physics is required. Therefore, I will be revising these topics to refresh my understanding [2]. A rendering engine will also need to be made to display the agents and environment in a user-friendly way and to make visual observations that can be used to test hypothesises devised by the user.

While researching, it was decided that C++ with OpenGL would be the best language to use to create the physics/rendering engine. C++ was chosen for its object orientated programming and OpenGL for its graphical capabilities.

reference [4] gives a great background of artificial life with explanations and examples of genetic algorithms for example manipulating genotypes using mutation and crossover to simulate the effects of evolution by natural selection. [5] Gives a concise method for constructing an agent's genotype (for the body) using directed graphs which become the agent's phenotype. This method makes it easy to create the initial population of agents and altering existing morphologies during 'breeding' of agents.

[6] Gives information on how to modify the agent's ANN with the use of genetic algorithms. It also provides an in-depth guide on how to construct the ANN and how it's outputs can be used to control the behaviour of the agent.

[1] http://buildnewgames.com/gamephysics/
[2] Edexcel AS and A Level Modular Mathematics: Core Mathematics 4 (Chapter 5)
[3] Game Physics Engine Development: How to Build a Robust Commercial-Grade Physics Engine for your Game – Ian Millington
[4] Artificial Life – Christopher G. Langton
[5] Evolving Virtual Creatures - Karl Sims
[6] Building Artificial Nervous Systems with Genetically Programmed Neural Network Modules - Hugo de Garis

2.2 **Summarise the project objectives and outputs in about 400 words.**
These objectives and outputs should appear as tasks, milestones and deliverables in your project plan. In general, an objective is something you can do and an output is something you produce – one leads to the other.

Outputs

- A number of interesting agent morphologies, their ANN and 'strategy' for solving the presented problems.
- A functional physics simulator.
- Statistical graphs analysing the successfulness of the agents and the genetic algorithm.

Objectives

The physics engine will be relatively simplistic containing the most essential elements of a physics engine in order to simulate the genetic algorithms. This will include but is not limited to. Rigid body collisions, movement using vectors, air/water resistance, rotation of rigid bodies with 6 degrees of freedom.

Agents will need to be constructed of multiple independent ridged bodies (components) that are connected using 'joints. Each joint will contain a sensor that calculates that angle between the two components. Components will also be able to detect collision with the environment, itself and other agents. These sensors will be the inputs to an agents ANN. The ANN will then calculate an output based on the weights between neurons. These outputs will control the agents every behaviour for example their movement. Over time the genetic algorithm will produce more successful agents by altering the weights of the ANN, this would be considered innate 'learning' similar to a simplistic insect's nervous system. ANN alteration may also be made using backpropagation. This would be considered traditional learning an adaptive behaviour that changes during the course of the agent's 'life' that aids survival.

The main objective of the simulation will be the emergent behaviour mode which simulates a thousand initial agents and places them in the physics simulation. The simulation is than run for many generations to see the how the agents adapt to their surroundings.

A secondary objective for the simulation will be to solve novel problems such as movement, swimming or jumping which can be artificially selected for by the user by altering the test for agent fitness.

Statistical data will be recorded during the simulation to track aspects such as, speciation, the fitness of each agent, the average fitness of the group over time. From this data graphs and statistical models will be constructed to show the data is an informative way.

2.3 **Initial project specification - list key features and functions of your finished project.**
Remember that a specification should not usually propose the solution. For example, your project may require open source datasets so add that to the specification but don't state how that data-link will be achieved – that comes later.

	The project will be considered successful if, • the physics simulations appear to be realistic. • Agents are able to solve novel tasks such as walking, swimming, collect rewards, communicating with one another, ect. • Agents adaptively learn and increase in complexity as the simulation is run
2.4	**Describe the social, legal and ethical issues that apply to your project. Does your project require ethical approval? (If your project requires a questionnaire/interview for conducting research and/or collecting data, you will need to apply for an ethical approval)**
	Legal There is no legal issue within my project. For instance, no individual's personal data or information is required. **Ethical** The project does not require any ethical approval.
2.5	**Identify and lists the items you expect to need to purchase for your project. Specify the cost (include VAT and shipping if known) of each item as well as the supplier.** e.g. item 1 name, supplier, cost
	N/A
2.6	**State whether you need access to specific resources within the department or the University e.g. special devices and workshop**
	Potentially university GPU or more powerful hardware to run simulation quicker/longer.

SECTION 3 – Project Plan

3.1	**Project Plan** Split your project work into sections/categories/phases and add tasks for each of these sections. It is likely that the high-level objectives you identified in section 2.2 become sections here. The outputs from section 2.2 should appear in the Outputs column here. Remember to include tasks for your project presentation, project demos, producing your poster, and writing up your report.

Task No.	**Task description**	**Effort (weeks)**	**Outputs**
1			
1.1	**Background Research**	3	…
1.2	Reading reference [3]	2	
	Relearning linear algebra	1	
2	**Analysis and design**		
2.1	Designing physics engine	4	...
2.2	Designing genetic algorithm	3	…
	Designing agent nervous system	1	
3	**Develop prototype**		
3.1	Building physic engine prototype	4	…
3.2	Creating hypothesis	1	…
4	**Testing, evaluation/validation**		
4.1	unit testing	1	…
4.2	Testing hypothesis	1	…
			…
5	**Assessments**		
5.1	write-up project report	2	Project Report
5.2	produce poster	0.5	Poster
	…		
TOTAL	**Sum of total effort in weeks**		

SECTION 4 - Time Plan for the proposed Project work

For each task identified in 3.1, please *shade* the weeks when you'll be working on that task. You should also mark target milestones, outputs and key decision points. To shade a cell in MS Word, move the mouse to the top left of cell until the curser becomes an arrow pointing up, left click to select the cell and then right click and select 'borders and shading'. Under the shading tab pick an appropriate grey colour and click ok.

	START DATE: ../../…. <enter the project start date here>												
	Project Weeks												
Project stage	0-3	3-6	6-9	9-12	12-15	15-18	18-21	21-24	24-27	27-30	30-33	33-36	36-39
1 Background Research	▨	▨	▨										
2 Analysis/Design	▨	▨	▨	▨									
3 Develop prototype.			▨	▨	▨	▨	▨	▨	▨				
4 Testing, evaluation/validation			▨	▨	▨	▨	▨	▨	▨				
5 Assessments													

RISK ASSESSMENT FORM

Assessment Reference No.		**Area or activity assessed:**	
Assessment date			
Persons who may be affected by the activity (i.e. are at risk)	**Alexander Small**		

SECTION 1: Identify Hazards - *Consider the activity or work area and identify if any of the hazards listed below are significant (tick the boxes that apply).*

1.	Fall of person (from work at height)		**6.**	Lighting levels		**11.**	Use of portable tools / equipment		**16.**	Vehicles / driving at work		**21.**	Hazardous fumes, chemicals, dust		**26.**	Occupational stress	
2.	Fall of objects		**7.**	Heating & ventilation		**12.**	Fixed machinery or lifting equipment		**17.**	Outdoor work / extreme weather		**22.**	Hazardous biological agent		**27.**	Violence to staff / verbal assault	
3.	Slips, Trips & Housekeeping		**8.**	Layout , storage, space, obstructions		**13.**	Pressure vessels		**18.**	Fieldtrips / field work		**23.**	Confined space / asphyxiation risk		**28.**	Work with animals	
4.	Manual handling operations		**9.**	Welfare facilities		**14.**	Noise or Vibration		**19.**	Radiation sources		**24.**	Condition of Buildings & glazing		**29.**	Lone working / work out of hours	
5.	Display screen equipment	✓	**10.**	Electrical Equipment	✓	**15.**	Fire hazards & flammable material		**20.**	Work with lasers		**25.**	Food preparation		**30.**	Other(s) - specify	

SECTION 2: Risk Controls - *For each hazard identified in Section 1, complete Section 2.*

Hazard No.	Hazard Description	Existing controls to reduce risk	**Risk Level** (tick one)			Further action needed to reduce risks *(provide timescales and initials of person responsible)*
			High	Med	Low	
5	User will be using monitor screen for several hours at a time	Take regular breaks for 15 minutes each hour			✓	
10	Electrical equipment as the potential to break or become hazardous	Check quality of electrical devices			✓	
Name of Assessor(s)			**SIGNED**			
Review date						

Health and Safety Risk Assessments – continuation sheet

Assessment Reference No	
Continuation sheet number:	

SECTION 2 continued: Risk Controls

Hazard No.	Hazard Description	Existing controls to reduce risk	**Risk Level** (tick one)			Further action needed to reduce risks *(provide timescales and initials of person responsible for action)*
			High	Med	Low	

Name of Assessor(s)		**SIGNED**
Review date		

Log Book

Week	Summary	Planed actions
01/10/18	This week I started researching information surrounding physics engines and evolutionary algorithms for the project initiation document. I am currently reading the book, 'Understanding physics engines' – by Ian Millington. I read chapters pertaining to linear algebra, vectors, forces such as friction and springs	Continue reading more chapters of the book. Look into OpenGL tutorials
08/10/18	Watched videos on OpenGL looked at documentation. Began implementing rendering engine. Had meeting with supervisor to discuss PID.	Finish tutorial series on OpenGL Looking into Ian Millington's code Looking into topics discussed in meeting
15/10/18	Test Ian's physics simulation and looked into implementing fireworks in a test project.	Look into implementing more forces into the engine. Complete more OpenGL tutorials
22/10/18	This week was used to work on other coursework.	Continue working on last week's actions
29/10/18	Struggling to implement the required features for the physics engine. Looked into alternatives for physics engines. Completed more OpenGL tutorials. 3D render has been uploaded to GitHub	Make more progress on the 3D render Explore more options for physics engine.
05/11/18	3D render is complete Looked into how to begin attaching the physics engine to the render.	complete 3D render
12/11/18	This week was used to complete other coursework.	Continue working on last week's actions
19/11/18	Finished the 3D which is now fully working in isolations. Rendered sphere and monkey head to screen. Struggling to implement physics engine. Time is passing quick may need to look into other option for physics engine.	Research into Box2D and BulletPhysics.
26/11/18	Implemented the Box2D testbest and played around with its features. Thought that it would suffice for the project.	Look at implementing back end library for the project. Look into SDL and SMFL for I/O. Use OpenGl knowledge for basic rendering.
03/12/18	This week was used to work on other coursework	
10/12/18	This week was used to work on other coursework	
17/12/18	Created and design backend library using UML. Started creating the backend library for the evolutionary algorithm	Work building camera, fps controller
24/12/18	Time was spent on Christmas festivities	

31/01/19	Time was spent on Christmas festivities	
07/01/19	Build camera, fps controller and error handler	Create basic rendering and implement small project to demonstrate backend library.
14/01/19	Keep working on back end library and addition project	Continue working on additional project
21/01/19	Completed game project to test back-end library and is no uploaded to GitHub.	Explore Box2D more.
28/01/19	Read Box2D manual and start reading 'introduction to game physics with box2D'	Look into implementing Box2D with back end library
04/01/19	Started writing report. Filled out lit review and problem articulation. Implemented Box2D physics engine into backend library. Struggling to render an object to the screen	Continue working on rendering the Box2D world to the screen
11/02/19	Continued writing report, created the main objectives of the project. Began designing the agents for the simulations. Thinking about using a graph or tree structure for them.	Look into terrain design and evolutionary algorithms.
18/02/19	Began designing the terrain generations. Box2D contains edge shapes useful for terrain. Successfully render box2D world to the screen. Can now created objects and seem them in world.	Implement the terrain into the world.
25/02/19	Implemented terrain using sine and cosine waves. Continued reading, 'An introduction to evolutionary algorithms' wrote implementation for terrain	Implement agents into the world. Continue reading about EAs
04/03/19	Implemented agents from the design sheet, used tree structure to construct them. Looked into how to save the agent tree structure to file. Continued reading, 'An introduction to evolutionary algorithms' Wrote implementation for agents	Look into design the agent solution space. What is a good random selection of agents to generate.
11/03/19	Found a good solution space for the agents. Agents will generally simpler than complex in morphology. Write introduction to report and abstract	Implement genome to string method.
18/03/19	Implemented genome to string and back conversion. Agents can now be saved to file.	Implement crossover and mutation mechanisms
25/03/19	Implemented crossover (uniform, arithmetic mean) and mutation (morphological and real-value) Finished evolutionary algorithm and agents can now be tested. Wrote implementation for genome to string conversion	Collect results from the simulation. Experiment with metrics.

01/04/19	Gathered some interesting results from the simulation. Simulation produce sufficient agent quite quickly. Tested how population size effects Created testing criteria for the program. looked at performance testing, unit tests and acceptance tests.	Look into fixed time step for simulation Write testing section for the dissertation Write results sections for the dissertation.
08/04/19	Gathered more results such as changing the fitness criteria and looking at generalisability of agents.	Write conclusions and future work for the dissertation.
15/04/19	Finished writing main sections of the report. Create power	Prepare for presentations
22/04/19	Complete presentation on dissertation. Presentation went okay.	Wrap up project report and code. Proof read document. Revise for exams.