jp010731_Final_Year_Report

by Oliver Reynolds

Submission date: 02-May-2017 02:08AM (UTC+0100)
Submission ID: 71591544
File name: Report.pdf (1.13M)
Word count: 20875
Character count: 116612

School of Mathematical, Physical and Computational Sciences
Individual Project – CS3IP16

Predator-Prey Simulation

Oliver Reynolds
21010731

Supervisor: Richard Mitchell
24th April 2017

Abstract

Braitenberg Vehicles are a simple type of wheeled robot equipped with sensors to perceive their surroundings, and these sensors are then connected to the actuators which drive their wheels, resulting in vehicular motion. This project aims to create a 3D graphical simulation that shows how complex behaviours can emerge from relatively simple configurations of sensor-actuator configurations. The predator prey co-evolutionary dynamic is explored whereby simulated vehicles display behaviours evident to living organisms in nature. In doing so, we draw parallels between these sensor-actuator connections such that they come to bear a resemblance to the neuronal structure of living organisms. In implementing this graphical simulation, the OpenGL API is researched and thus utilised to create an engaging real-time display of Braitenberg vehicles moving around their environment. In doing so, we additionally explore a range of different techniques for using OpenGL to render various aspects of the predator-prey robot simulator. The outcome of this project is an extensible framework that provides a means by which enthusiast robotics developers can implement their own Braitenberg Vehicles, and a fully-functional simulator that those interested in learning about the subject domain can run to explore the topic in an interactive environment.

Acknowledgements

I would like to thank Richard Mitchell and William Harwin for their assistance throughout the development of this project.

I would also like to thank my family for their support along the way.

Table of Contents

1. Glossary of Terms and Abbreviations

API – Application Programming Interface

Box2D – Open-Source 2D Physics Engine written in C++ for simulating rigid bodies.

Braitenberg Vehicle – A concept of vehicle design by cyberneticist Valentino Braitenberg in which the motion of a vehicle is controlled by its sensors.

FOV – Field of View

GLSL – OpenGL Shading Language

GPU – Graphics Processing Unit

GUI – Graphical User Interface

NDC – Normalised Device Coordinate

Neuron – Symbolic representation of a connection between a Braitenberg Vehicle's Sensor and an actuator.

OpenGL – Low-Level API to facilitate communication of data to the GPU.

Predator-Prey – A model representing the behaviour of entities within a population, whereby predators rely on prey as their source of food.

Shader – A computer program that runs on the System's Graphics Card, split into sections representing a type of processing to be executed.

VAO – Vertex Array Object

VBO – Vertex Buffer Object

2. Introduction

Braitenberg Vehicles were conceived by Italian Cyberneticist, Valentino Braitenberg in his pivotal text on the subject "Vehicles". In this text, a range of different vehicles are established, each behaving in a slightly unique way, but each behaving in a manner also that mimics emotions seemingly conveyed by living organisms. These vehicles are simple robots in which the wheels are powered by actuators, connected to sensors. The vehicles move per their sensory input and can display distinct types of movement and behaviour by configuring how the sensor-actuator connections are wired.

In this project, we treat these connections as a simplified neuron, whereby an input signal can be amplified, attenuated or even masked by setting a numerical threshold. In creating a simple model of neurons, we then attempt to create a series of vehicles which can be controlled by tweaking their neuronal connections. These controlled vehicles are then able to interact with each other in a completely virtual, but as realistically accurate to real-world forces as possible. In the implemented 3D simulation, Braitenberg vehicles can be created and tweaked with different parameters to affect their movement. Two distinct types of vehicle move around at run-time, representing a Predator-Prey dynamic, whereby a Predator aims to hunt a Prey vehicle, and the Prey aim to flee from Predators.

The primary objective for the product is to serve as a learning tool, by which users can gain an intuition of how complex behaviours can emerge from distinct types of Braitenberg vehicle.

A secondary objective is for the simulation to function as a basic prototyping tool, for those wishing to create Braitenberg vehicles using actual hardware.

We begin by articulating the aims and requirements of this project, and the problems it sets out to solve. It is paramount in this section that the potential stakeholders are clearly identified so that the development of the project can be targeted towards that audience. In addition, the constraints and assumptions made about the project, established in the Project Initiation Document are revisited and discussed.

Next, effort is made to examine existing literature and products that explore some of the concepts this project aims to cover in detail. Naturally Valentino Braitenberg's text is the primary source of interest here, although some existing products that have aimed to simulate Braitenberg's vehicles are analysed. For each technology and tool in this section, we consider how aspects of each could be used to enhance the current solution. In defining these technologies, we then formulate a solution approach for how these requirements can be successfully met. This solution approach briefly outlines some of the key tools and technologies examined that should be used in the actual implementation of the simulator, and is followed by a set of acceptance criteria which outline some of the technical objectives for the project.

The design and implementation of the simulator is then thoroughly explored, whereby details on how underlying technologies and libraries used to create the product will be detailed. The implementation of the simulator is substantial, so significant effort is made to cover as many of the key implementation details as possible here. The discussion of the implementation is split according to the subdomain of the topic being discussed.

After covering the design and implementation of the simulator, the testing process is outlined. Within this testing process, performance testing, acceptance testing and unit testing were carried out to verify and validate the project solution. This section is followed by the Discussion, in which the outcome of the testing phase is analysed, and limitations of the project following its implementation considered.

Potential issues regarding the social, legal, ethical and health and safety implications of the project are briefly discussed. This section acts as a reminder of long-term considerations that need to be considered as the project continues to develop in the future.

Finally, this report ends with a Conclusion section wherein the project objectives are briefly restated and we determine how effective the developed solution was in meeting these objectives. In addition, we consider how the simulation might be improved in future iterations of development.

3. Problem Articulation and Technical Specification

3.1 Problem Statement

Braitenberg Vehicles are a simple type of robot, which emphasises on analogue circuitry in place of a microprocessor. The motion of these robots is driven by direct stimulus from the environment, using sensors connected to comparators, which in turn drive a robot's locomotive system. With Braitenberg Vehicles, motion is determined by the construction of connections between its sensors and its motors. New behaviours can emerge by modifying the construction of these circuits – and more complex configurations resemble a neural network. There is uncertainty however in predicting the behaviour of robots constructed using analogue circuits. This uncertainty becomes more prevalent as more complex networks of circuitry are established – akin to explaining why certain weights are set in an artificial neural network. To overcome these issues, there is therefore a need to develop a simulator that can help a user visualise how instantiated vehicles may behave in a preconfigured environment.

Furthermore, by developing an interactive simulation, robotics engineers will be able to rapidly prototype new robotic hardware, with a stronger understanding of sensor-motor connections required to modify the robot's behaviour. At the same time, suitable constraints will be imposed on the way in which the robots are customised to avoid 'reward hacking' – where the program would offer an unrealistic amount of control, resulting in flawed simulations.

3.2 Stakeholders

Developer – Oliver Reynolds

The developer of this project will be responsible for developing the solution that satisfies the objectives outlined in *3.3* and resolves the problem statement specified in *3.1*. As the sole developer on this project, it will be critical to ensure all deliverables are provided by the necessary deadlines and that the balance of effort between implementation and programming with documentation is maintained.

Project Supervisor – Richard Mitchell

The project supervisor will follow the development of the project and will schedule weekly meetings whereby feedback will be given on the progress of the project. In these weekly meetings, the Project Supervisor will discuss with the Developer of the project, the progress that has been made. The project supervisor will offer advice when needed to ensure the overall quality of the produced solution and will also be responsible for ensuring the project's objectives are met.

User - Robotics Developer

Robotics Developers who want to prototype a design for a Braitenberg vehicle using actual hardware might be interested in using the developed solution to trial interactions between different vehicle behaviours before deployment. The robotics developer will require that the simulation is clear and easy to use, while also being complex enough to develop a range of different behaviours. Further, this user will also want the environment in which the robots operate to be as physically realistic as possible such that Braitenberg designs work as well when implemented using actual hardware.

3.3 Technical Specification

In this section, the technical objectives are stated, which will be used throughout the development of the simulator, to guide the implementation process.

Platform and Support

- The simulation should be able to compile and run on Windows and Linux platforms.

AI

- Implement artificial brains using Braitenberg vehicles and Neural Networks. The vehicles may have some aspect of memory, so they can recall where obstacles were. Allow the artificial brains to be swapped out and configured at run-time.
- Allow weights between sensory input and simulated actuators to be configured by the user, so that it is possible to observe the relation between a vehicle's weights and its behaviour.
- Allow connections between sensory inputs and comparators to be changed by the user, so that is it possible to observe the relation between a vehicle's neural network and its behaviour.

Physics

- Create a physically plausible simulation, in which the vehicles will operate. A framework could be created with functions implementing external forces. A Newtonian approach may be adopted in these forces affecting the acceleration and angular acceleration of the vehicles.
- Position will be determined using Euler Integration using Acceleration and Velocity to improve the accuracy of the simulation.
- Rotation will be determined using Euler Integration using Angular Acceleration and Torque to improve the accuracy of the simulation.
- Friction against wheel modelled using relation between wheel's surface area applied to ground, velocity, and friction constant of the surface.

GUI

- Create a functional interface for the user to interact with the simulation.
- The simulation will feature simple prompts to direct the user to specific features.
- One portion of the GUI will be dedicated to displaying textual information about the current simulation state, to inform the user.

Rendering

- Represent the simulation using graphics implemented using an appropriate means of rendering. The approach taken towards rendering the simulation must be such that it is decoupled from the part of the codebase that deals with the logic behind vehicle movement.
- The sensors will be represented using sectors of a circle, rendered using a Vertex-Geometry-Fragment pipeline of GLSL Shader programs. Alpha blending will be utilised to overlay render of sensor field of view over scene geometry to emphasise sensor-stimulus events.
- The vehicle will be represented using a simple quad, rendered using a Vertex-Geometry-Fragment pipeline of GLSL Shader programs.
- Obstacles in the scene will be represented using arrays of quads set in various positions. A future extension of this would-be user-defined n-gons that still work using the sensor-collision functionality.
- Consideration towards platforms that don't have a modern GPU should be considered.

Maths

- Vectors and Matrix types will be the backbone of the graphical components of the simulation. A small library providing implementations of various operations using these types will speed development time.
- Define 2D, 3D and 4D vector types to aid the implementation of high-end simulation logic.

- Define 4D matrix type to aid the implementation of high-end simulation logic.
- Derive and implement Orthographic 4D matrix routine, with which to provide the simulation a birds-eye view, scaled 1:1 with the viewport resolution of OpenGL context window – this will be used for the construction of an object's final Model-View-Perspective matrix for GL rendering.
- By extension, implementation of Perspective 4D matrix routine, with which to provide a more user-friendly 3D view of the simulation.

3.4 Constraints

The following constraints are recognised to meet the technical objectives as specified in 3.3.

- The area in which a vehicle's sensor can scan will be represented as a triangle primitive.
- Sensor events in which a signal stimulates a vehicle sensor will be initially be discrete in nature, and thus be represented using boolean data type.
- Sensor detection takes place in 2D and only X and Z components of 2D positions are considered.

3.5 Assumptions

The following assumptions are made to meet the technical objectives as specified in 3.3.

- The computer running the simulation supports OpenGL version 3.2 at the very least and is thus capable of displaying the 3D simulation view.
- The user has some familiarity with the concept of Braitenberg Vehicles and will thus be familiar with the concept of changing sensor-neuron connections to influence a vehicle's behaviour.
- Objects in a simulation perceived by vehicle sensors can be represented using a 3D cartesian point.

.

4. Literature Review

In examining existing work related to the domain of Braitenberg Vehicles and Predator Prey robots, we aim to identify how existing research can be built upon to develop an original and relevant solution.

4.1 Braitenberg Vehicles

Braitenberg's work on vehicle behaviours serves as the best place to start this review, in which different types of vehicle behaviours are presented based on sensor-actuator connections [1]. A simple vehicle is initially presented as having a single wheel and a single temperature sensor, that moves when it reaches an area with a higher temperature and potentially comes to rest in a cooler environment [2]. This idea is then developed to present three new types of vehicles with two motors and two sensors on each side, each with a unique configuration of how these sensors and motors are connected. The first vehicle see the motor connected to the wheel on the same side, the second vehicle has its motors connected to sensors on the opposite side, and the third vehicle has sensors connected to both wheels [3]. From these different configurations, it is possible to create apparent behaviours in the way that each vehicle interacts with its environment. This concept is developed further by describing a more complex relationship between sensor and motor whereby sensory input up to a certain strength will result in a maximum speed of a wheel motor, but even stronger signals beyond this threshold lead to the speed of the motor to begin to decrease [4].

Braitenberg's Vehicles serve as a useful base for which the vehicle simulator can be created. The key concept being that vehicles could be made to seem attracted or repelled from an energy source, which could be used to implement the Predator-Prey dynamic, whereby Predators would be attracted to other vehicles emitting a light source and Prey could act in a comparable way to the vehicles that moved away from their surroundings.

The simulator presented in [5] allows Braitenberg Vehicles and Machina Speculatrix vehicles to be created, that exhibit predator-prey behaviours. The world in which the vehicles operate in this simulator feature coloured lights either attached to the robots, or at a static position. In this simulation, the Braitenberg vehicle feature two sensors connected directly to actuators, modelling perception and action whereby sensor detection leads to a wheel driving forwards and influencing the motion of the robot. A useful aspect of this simulation is that it can be controlled using standalone initialisation files, which can be edited at runtime to change the properties of the simulation. William Harwin's simulator serves as a highly useful starting point upon which the proposed predator prey solution can be inspired from, as it successfully implements an accurate form of vehicle locomotion driving by wheel motion. Further, the use of different coloured lights to represent distinct types of energy source is a feature that would be useful to add to the developed solution, to help reinforce and demonstrate some of Braitenberg's more complex vehicle ideas based on the concept of different energy sources. One other noteworthy element of this simulator is that it uses the Fixed-Function Pipeline of the OpenGL API [6] for rendering the vehicles to a windowed framebuffer. This shows that rendering an effective display of the simulation can be achieved by making effective use of geometric primitives and without having to rely on a many external third party libraries that provide high-level rendering functionality. The GLUT library is instead used to provide the utility functionality for creating an OpenGL context and creating a window containing a framebuffer for writing to [7].

Next, we explore an implementation of an interactive webpage of a Braitenberg Vehicle, developed by Richard Mitchell as part of the Future Learn MOOC, Begin Robotics [8] [9]. The interactive simulation of a Braitenberg Vehicle can be configured by changing the weights of intermediate neurons between sensor and wheel motor. Four neurons can be configured and are labelled according to the motor and wheel they are connected to: LL, LR, RL, RR. These neurons are used, alongside the value of the left and right motors (LM and RM) to calculate the resulting speed for a vehicle's left or right wheel as so:

Left	LS * LL + RS * LR + LM
Right	RS * RR + LS * RL + RM

Figure 1- Calculating Left and Right wheel speeds *[10]*

One of the effective features of this simulation is that it features exercises for the user, to encourage them to use the simulation in diverse ways, but more importantly to help teach a user unfamiliar with Braitenberg vehicles, how affecting neuron weights can allow for different vehicle behaviour to be created. Exercise tasks might therefore be a viable way of ensuring that the developed simulation helps to educate a non-technical user, while also encouraging a user aware of the technical domain to fully explore the developed solution. Another interesting facet of this simulation is that it is implemented in a web-page and uses JavaScript, allowing for the same simulation to be run across different OS platforms without needing to modify the implementation. Further, the use of JavaScript in this way encourages the use of the Model-View-Controller architectural pattern, whereby the data (Model) and the relevant Controllers can be encapsulated within a loaded JS script and kept separate from the HTML view, thus making it easier to develop the simulation further by creating different HTML views that connect to the Controller. This also avoids defects that might arise from complicated dependencies between code written for rendering, display and backend, overall resulting in a more modular architectural design [11].

4.2 Predator Prey Behaviour

The paper presented in [12] outlines evolutionary strategies to evolve the behaviour of robots. It outlines an experiment conducted to co-evolve predator-prey behaviours. In this experiment, a population of predator-prey robots are created and placed into square arenas. Predators are evaluated using a fitness score, based on how long they took to catch the corresponding prey vehicle. In conducting this experiment and evolving the predator and prey robots, the author could observe different pursuit and evasion strategies and concludes by comparing the instability of evolved vehicle behaviours in the experiment to natural systems because of co-evolutionary dynamics. This paper is of interest to the development of the simulation, as it demonstrates an effective way in which predator prey behaviour can be evolved and improved upon, without directly hard-coding the behaviour of the robot. The result of this is a wider range of vehicular behaviour that changes as the simulation runs over time.

4.3 OpenGL

The OpenGL reference pages serve as useful piece of documentation in explaining how the OpenGL API can be used within a C++ application. The API detailed in these reference pages allows for the rasterization of primitives defined by vertices to be drawn to a frame-buffer and in turned displayed on a user's screen. Within this process, the user can specify Shader programs to handle transformations and lighting of defined geometry. [13] The documentation aims to outline every facet of the API, including explanations of GL commands and how the client-server model representation of the CPU and GPU facilitates the transfer of data between these components.

OpenGL seems suitable for use as a graphics API in this project, given its availability for use on a variety of different Operating Systems [14], potentially making it easier to port to different platforms and thus serves as a more suitable choice than Direct3D, which is currently only supported for Windows based platforms [15].

5. The Solution Approach

In this section, we use the Literature Review and Technical Specification to define potential solution options which could be used to implement the predator-prey simulation. For each option, we identify how that approach might be advantageous in the development of the simulation, but also identify what its shortcomings are. Finally, we use these enumerated options to determine a chosen solution which will be used to create the simulator and which will be detailed further in the Implementation section of this report.

5.1 Solution Options

5.1.1 Rendering and display approach

OpenGL (Desktop-based)

OpenGL as a means by which the simulation is rendered would be effective given its portability across different platforms [14]. One of the key advantages in using a low-level API such as OpenGL is that it offers a degree of flexibility for rendering techniques that might be more difficult to achieve in more abstract high-level tools such as a Game Engine or an external library. Adoption of OpenGL for rendering would also meet an aspect of the technical specification, which is supporting platforms that don't have a modern GPU, or perhaps even integrated support for graphics APIs within the CPU.

Simple and Fast Multimedia Library (SFML) [16] (Desktop-based)

SFML is a library available for use in a variety of different programming languages (including C++), and provides a straightforward way in which the developer can interface with various media, such as graphics, OS windows and audio. The benefit of using such a library for rendering and displaying the graphics of the simulation, is that development time would be saved by not having to 'reinvent the wheel' and creating such interfaces using more low-level APIs. However, the downside of using SFML, is that the same recovered development time may have to be used to become familiar and fluent in usage of the library. In the Technical Specification, it was established that the developed solution should provide a means by which custom Shaders can be written, and SFML provides functionality for uploading such Shaders to an OpenGL context. But the drawback of this approach is that it becomes more difficult to debug Shaders in the case of compilation errors, whereas the OpenGL API provides commands to achieve this. Although SFML may seem like the obvious choice initially, there are certainly pitfalls to its usage that may interfere with development progress.

HTML & WebGL (Browser-based)

One of the critical advantages in adopting a more web-oriented solution is that the developed solution would work on any user's system, providing they could run a web browser that supported JavaScript. As explored in the Literature Review, the MVC architectural pattern would be naturally used for this approach. However, there are some limitations to consider. First, it is possible to implement the MVC architectural patterns in most programming environments, across a variety of different paradigms. Although the web-based approach encourages this style of architecture strongly, it by no means provides options for other architectures to be used. On the contrary, development within a C++ OpenGL environment would allow for MVC architecture to ensure modularity of code and ease of extensibility, and more. Second, WebGL although very versatile as a browser based OpenGL stand-in is fundamentally a subset of the OpenGL specification and other technical limitations exist as outlined in [17].

OpenGL ES (Mobile-based)

Another alternative is to develop a more mobile oriented implementation. The reason this is a viable option is the ability for modern smartphones to render impressive 3D graphics using OpenGL Embedded Systems, coupled with the widespread adoption of affordable smartphones across the world. In [18], we see that 1452 are currently capable of running OpenGL ES version 2.0, which would be more than capable of rendering the primitives required for the simulation. There are some downsides however. Targeting a legacy version of a subset of an API would require considerable development effort to maintain versions of the simulation that uses newer versions of OpenGL ES (such as 3.2+) and dividing that much attention towards maintaining both would naturally impact development of more critical aspects of the simulator such as the motion of the Braitenberg Vehicles. Further, the development process across mobile devices is currently fragmented into smaller ecosystems supported by the likes of Android Studio [19], IOS's XCode [20], and Visual Studio for Windows Phone devices [21]. Developing a mobile-based solution would require dividing the target audience twice – once for the OpenGL ES version support on the user's device and again for the device the user owns. It should be recognised however that this solution is feasible given the rendering criteria set out in the Technical Specification.

5.1.2 Braitenberg Vehicle implementation approach

Vehicle Movement

An initial idea for implementing vehicle locomotion involved keeping track of each vehicle's direction vector, and then normalising it to create a vector representing the vehicle's next offset in a subsequent frame of rendering. Turning left and right during motion would be implemented by then rotating this vector clockwise or anticlockwise depending on the difference in speeds between the two actuator driven wheels. A proof of concept was developed to test this implementation's effectiveness, by creating a 2D top-down view of Braitenberg Vehicles with two sensors, that turn in different directions using this movement strategy, upon sensor detection. In practice, this style of motion seemed convincing when watching the prototype simulation run. However, this style of movement doesn't consider physical properties of the simulation world such a mass, friction, velocity and torque, which is a key requirement in this project's Technical Specification. It would be possible to incorporate these into the simulation by implementing the necessary formulae to use such properties and affect the motion of the vehicle, however this might require significant development effort potentially detracting from other key areas of the Specification that need to be considered as a priority.

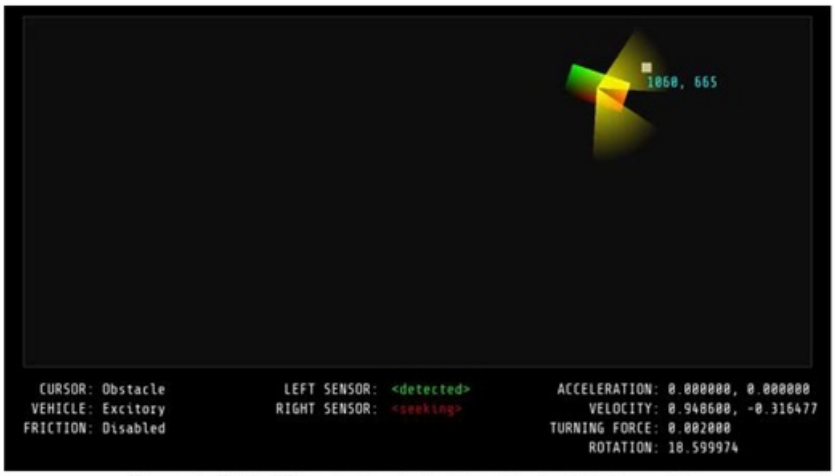


Figure 2- Early 2D prototype to test simulated vehicular motion

An alternative is to instead use an external Physics library that provides an interface by which physical properties stored within the main simulation portion of the codebase can be used as arguments to functions that return necessary motion values by which the vehicles can be moved. One such library, Box2D is available in C++ and allows for the creation of rigid-body and fixture types which could be connected to represent the wheels and body of a Braitenberg Vehicle. Box2D has some features that would be useful within the scope of the defined Technical Specification. One of these is automatic collision of bodies within a simulated Box2D thread, so long as they are initialised and attached to a Box2D World object [22].

Vehicle sensor object detection

Three different approaches are considered for the implementation of vehicles being able to detect external energy sources.

Calculating whether a point intersects a triangle (representing the sensor of a vehicle) could be achieved by using Barycentric Coordinates. Barycentric Coordinates define a 3-tuple of normalised values that can describe the position of any point within a triangle, where each of the vertices are labelled (1,0,0), (0,1,0) and (0,0,1) respectively [23]. A candidate point represented using Cartesian Coordinates (x, z) is converted into a barycentric 3-tuple (r1, r2, r3) using the technique demonstrated in [24] as shown in **Figure 3**.

```
float div = ((b.y - c.y) * (a.x - c.x) + (c.x - b.x) * (a.y - c.y));

float r1 = ((b.y - c.y) * (p.x - c.x) + (c.x - b.x) * (p.y - c.y)) / div;
float r2 = ((c.y - a.y) * (p.x - c.x) + (a.x - c.x) * (p.y - c.y)) / div;
float r3 = 1.f - x - y;
```

Figure 3- Cartesian to Barycentric conversion

If each component of the calculated barycentric coordinate for cartesian input point P is between 0 and 1, then P lies within triangle ABC and the triangle intersection function returns true.

Another approach involving sensor-object detection is similar but perhaps more closely represents the circular spread of a sensor. The method involves determining whether a 2D point P intersects the sector of a circle, defined using its origin-point O, radius R, and two vectors A and B - where each of A and B are cast from O. Determining intersection of a point P with this sector require 3 properties to hold [25]:

1. The Euclidean distance between point O and point P must be less than R.
2. P must be anticlockwise of vector B.
3. P must be clockwise of vector A

These rules can then be described in relation to the problem domain of a vehicle's sensor to formulate a C++ function to accurately perform the check. The first rule ensures that the distance from the centre of the circle (the source point of the vehicle's sensor) to the point to be checked in the intersection function must be less than the radius of the circle (the range of sensor).
The other two rules ensure that the point being checked is bounded between the 2 arms representing the extents of a vehicle's sensor.

Finally, Box2D also has support for computing intersections, which can be applied to fixtures by setting its member attribute 'isSensor' to true. This allows for polygonal geometry to be defined using a fixture, but prevents intersections with other fixtures in Box2D simulation.

Typically, changing the attributes to prevent fixture collisions in the Box2D library prevents the relating collision callback functions to be triggered by Box2D, however, by using sensors these callbacks are still invoked and can thus be used for processing of detected bodies intersecting the containing fixture [26].

5.2 Chosen Solution

OpenGL is chosen as the choice of rendering method for the developed implementation due to the flexibility it provides in being able to easily render a range of graphics. The other approaches are technically feasible to use and other similar a similar advantage, but are more restrictive in the platforms and environments they would work in.

Vehicle movement in the simulation is handled using external library Box2D and will consider physical forces to ensure movement is as realistic as possible. This should allow for more convincing vehicle movement and removes the need for a bespoke collision detection solution to be developed, resulting in increased development time for other areas of the project.

Object detection via a vehicle's sensors will be mimicked using the barycentric conversion of cartesian coordinates, to determine whether another vehicle's position lies within the triangle defined by a vehicle's sensor attributes. This approach is favourable over the others explored in this section due to its simplicity and ease-of-implementation, while also being computationally cheap.

5.3 Acceptance Criteria

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

- The 3D scene should suffice for the user to understand how the vehicles are behaving and should provide a simple graphical representation of the artificial brain simulation.
- Vehicles powered by Braitenberg brains should be clearly identifiable and it should be possible to distinguish these from other vehicles.
- The Braitenberg vehicles should act per their Predator-Prey class designation.
- The GUI must be clear and easy to use.
- Each button should be intuitive and correctly convey its function.
- A standalone C++ library with utility functions for physics calculations.
- It should not rely on any aspect of the main simulation codebase. Ideally, it should be fully unit tested, and these tests packaged with the library as a reference for the user

6. Design & Implementation

The design and implementation section of this report embody the overall development effort made towards developing the predator-prey simulator. In the design section, an emphasis on interface decisions is adopted, while the implementation section deals more with how the predator-prey simulator's backend logic works.

6.1. Design

In this section, details about how the look-and-feel of the simulator are discussed and functionality explained.

6.1.1 GUI Elements

Allowing the user to interact with the simulation using a user-interface that is intuitive to use, is specified within the Technical Specification and should thus be designed such that little documentation or instruction is required to interact with it.

Interactive Buttons

A series of buttons are included in the render of the simulation, that the user can click on to interact with the simulation, shown in ***Figure 4***.

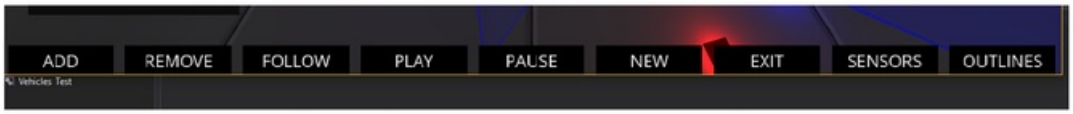


Figure 4- Buttons used to interface with the simulator

The functionality for each is outlined below:

- ADD – Another vehicle is added to the simulation. The increased population is persisted upon creation of a new simulation.
- REMOVE – From the population of vehicles, the last vehicle added is removed. For a simulation-run in which no vehicles were added, the 10^{th} vehicle from the population is removed. The reduced population is persisted upon creation of a new simulation.
- FOLLOW – Makes the camera switch between following a vehicle or viewing an overflow of the current simulation.
- PLAY – Starts the simulation if currently in a paused state. This button needs to be pressed to begin the simulation upon launching the program.
- PAUSE – Pauses the simulation if currently running. The paused simulation state will halt movement of vehicles and updating of timers used to decrease their energy levels as the simulation runs.
- NEW – Creates a new simulation and initialises a new population of vehicles. When creating a new simulation, the location of the new population of vehicles is randomised, as is the direction of each of the vehicles.
- EXIT – Puts the simulation into an exit state, which will eventually propagate to the managing GLFW windowing framework, prompting the program's main loop to finish and the program's termination.
- SENSORS – Affects the rendering of the simulation by disabling the ability to view the scope of each vehicle's sensors as a blue or red filled triangle per predator-prey designation.
- OUTLINES – Affects the rendering of the simulation by disabling the ability to view the outline of each vehicle's sensors as a blue or red triangle perimeter per predator-prey designation.

Simulator Information Display

A semi-transparent window was always developed as a GUI element for the simulator, shown in ***Figure 5***. The purpose of this display widget is to display real-time information about the current predator-prey simulator as it runs.

Figure 5- Simulator Information Display

The labels within this information display are outlined below:

- Generation – Tracks how many generations of predator-prey vehicles have been repopulated since the application was launched. A new generation is created in the application by either pressing the NEW button, or by waiting for the Inactivity Tracker to reach 0.
- Population – Tracks how many vehicles there are in the current population.
- Predator/Prey – Displays the ratio of vehicles allocated to either the predator or prey class.
- Energy Levels – Displays each vehicle's remaining energy. The energy of each vehicle gradually declines throughout its lifespan. Predator vehicles can increase this level by successfully colliding with a Prey vehicle. When a vehicle's energy level reaches 0, it will be removed from the simulation and a vehicle of the opposite predator-prey designation added.
- Inactivity Tracker – This timer decreases when all vehicles in the current simulation have not moved from their current position. When the timer reaches 0, a new simulation is generated and the generation counter is incremented.

Camera Movement

Camera movement is dealt with by adjusting the vectors that compose the view matrix used in OpenGL rendering in the developed simulator. The camera position defaults to an overhead view of the simulation, and its position of the simulation can be adjusted using the following controls:

- ARROW LEFT – Rotates the camera around the midpoint of the centre of the simulation clockwise.
- ARROW RIGHT – Rotates the camera around the midpoint of the centre of the simulation anticlockwise.
- ARROW UP – Increases the altitude of the camera, allowing for more elevated view of the simulation.
- ARROW DOWN – Decreases the altitude of the camera, allowing for closer view of simulated vehicles.

6.2. Implementation

The underlying implementation and logic for the simulator is outlined in this section. Much of the content here focusses on how vehicular movement using the Box2D library is achieved, as well as the overall rendering of the simulation.

6.2.1. 3D Simulation View using OpenGL

Rendering the simulation is achieved using the OpenGL API. In this section, some of the key development tasks required to use this API are detailed.

Representation and transfer of Vertex Data

The simulation view requires support for being able to render 2D and 3D geometric primitives. The vehicles in the simulation are rendered using basic cube meshes, the environment using quads and the vehicle sensors using triangles. Vertex data is defined CPU-side into static floating point arrays and is transferred to the GPU for use by Shader programs. To facilitate this, a series of Renderer classes are defined, each acting as a wrapper for drawing a primitive. Each renderer class provides an init method, which handles the creation of Vertex Array Objects and bound Vertex Buffer Objects, which are used to access geometry sent to GPU memory. In ***Figure 6*** a VAO is generated and bound to current OpenGL context and then a VBO is created and bound to the GL_ARRAY_BUFFER type., specifying that the VBO will be storing Vertex Attributes. The following GL command, glBufferData allocates the stored data for that bound VBO, whereby the size of the local array and its reference pointer is specified.

```
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);

glBufferData(GL_ARRAY_BUFFER, sizeof(utils::mesh::cube_vertices_normals),
&utils::mesh::cube_vertices_normals, GL_DYNAMIC_DRAW);
```

Figure 6- Generating a Vertex Array Buffer and allocating vertex data for a cube

Sending data to the GPU can be a costly operation, depending on the amount of data being sent, so these array commands are kept to init functions that only get called once as the simulation starts. VAO and VBO references remain active for the lifetime of the program, and are only deleted once it is closed. To use the generated VAOs for each renderer, they just need to be rebound before issuing any GL drawing command, as shown in ***Figure 7***:

```
glBindVertexArray(vao);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
shader.use();
glDrawArrays(GL_TRIANGLES, 0, 36);
shader.release();
glBindVertexArray(0);
```

Figure 7. Drawing process for the Cube Renderer

Defining Vertex and Fragment Shaders

Two types of Shader stage had to be accounted for before rendering anything, and are defined using GLSL. The Vertex Shader operates on Vertex Attribute Data stored in a VAO as part of the Vertex Rendering process. Its input is a vertex from the stream of vertices it reads from the bound VBO per the defined Vertex Specification. The defined Vertex Shaders in this case transform input vertices to different projection spaces and output a transformed vertex. The Fragment Shader operates on input fragments sent from the rasteriser, as well as per-vertex outputs from the previous Vertex Shader stage.

A series of Shaders were written to handle simple use-cases like transforming object geometry and applying basic diffuse lighting. To facilitate the use of these Shaders, and to help abstract the

complexity of sending the Shader information to the GPU via OpenGL commands, a Shader class was formulated, which takes 2 directories indicating location on disk of a Vertex and Fragment Shader as its arguments. Before Shaders can be used by the active OpenGL context, they must be first compiled and attached to a program object. A helper method was implemented to ease this process:

```
void Shader::compile(GLuint shader, const char* src) {
	GLint status;
	GLchar infoLog[512];
	glShaderSource(shader, 1, &src, nullptr);
	glCompileShader(shader);
	glGetShaderiv(shader, GL_COMPILE_STATUS, &status);
	if (!status) {
		glGetShaderInfoLog(shader, 512, nullptr, infoLog);
		std::cout << infoLog << std::endl;
	}
}
```

Figure 8 - Shader Compilation

The method takes a handle to the Shader and a character array representing the Shader source to be compiled and assigns this source to the handle using the glShaderSource function. The glCompileShader function then compiles any attached source bound to the Shader object. In development, errors were made in defining each Shader's source code, so the compilation status is queried using glGetShaderInfoLog and sent to standard output.

Once the vertex and fragment shaders are compiled, they are linked together to create a program object. This is implemented in a similar way to the implementation of the shader compilation function:

```
void Shader::link() {
	GLint status;
	GLchar infoLog[512];
	glLinkProgram(program);
       glGetProgramiv(program, GL_LINK_STATUS, &status);
	if (!status) {
		glGetProgramInfoLog(program, 512, nullptr, infoLog);
		std::cout << infoLog << std::endl;
	}
}
```

Figure 9 - Shader Linking

Once the program is created, it can be made active at any point by calling the glUseProgram command and passing a GLuint handle to the Shader's program object. A Shader must always be active to render anything within the OpenGL context.

Transforming vertices using different coordinate systems

One key Vertex Shader required to implement the 3D view of the simulation is one that transforms each input vertex into Projection Space. One of the most commonly used vertex Shaders written for this project is outlined below:

```
#version 450

layout(location = 0) in vec3 position;
layout(location = 1) in vec3 normal;

uniform mat4 model;
uniform mat4 projection;
uniform mat4 view;

out vec3 normal_out;
```

```
out vec3 fragpos_out;

void main() {
        gl_Position = projection * view * model * vec4(position, 1.0);

        // Generate normal matrix, as using non-uniform scale
        normal_out = mat3(transpose(inverse(model))) * normal;

        fragpos_out = vec3(model * vec4(position, 1.0));
};
```

Figure 10 - Vertex Shader for calculating projection transformation

The first line is the version directive, which specifies which version of GLSL the Shader is being written in, which acts as a pre-processor command to the GLSL compiler. The next 2 lines are layout qualifiers and specify how input data gets read from the currently bound VAO. In this case vertex positions are read from attribute index 0 and vertex normal are read from attribute index 1. In the C++ code, these attribute indices are set up as so:

```
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), 0);

glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float)));
```

Figure 11 – Setup of vertex attribute arrays in C++

glEnableVertexAttribArray specifies that the 0^{th} and 1^{st} vertex attribute indices are to be enabled. glVertexAttribPointer then specifies the actual data to be used as vertex attribute data, and we specify the stride of offset from the beginning of the vertex array data. In this scenario, 6 * sizeof(float) specifies offset as 6 floating point values before another vertex attribute is read, while for vertex attribute array 1, 3 * sizeof(float) indicates the offset distance from the beginning of the array for normal attributes. This allows us to only require a single VBO to store all vertex and normal data in, whereby vertex attribute arrays allow us to interleave this data, by specifying relevant offsets.

In the Vertex Shader, three uniform type mat4 variables are then declared. In GLSL, uniform types are a form of input variable that is defined from the CPU and sent via GL commands to the Shader. Uniform types, however, remain the same across all invocations of a Shader for a given set of geometry and allow us to avoid carrying out matrix transformation of vertex data on the CPU. Output variables normal_out and fragpos_out are then defined, which are sent as inputs to the subsequent fragment Shader. In the main body of the Vertex Shader, the built-in variable gl_Position which determines the final position of a vertex is then assigned by multiplying the three uniform transformation matrices with the input position of the vertex from the input vertex array. As it is possible that the overall shape of the geometry may have been manipulated by multiplication with the model matrix, the vertex normal is multiplied by the inverse transpose model matrix to correct it in case of uneven scaling.

Implementing Diffuse Lighting

The Fragment Shader implements diffuse lighting, whereby the intensity of a fragment's colour is calculated by calculating the dot product between a vertex normal and the direction of a light source and then multiplying this result with the colour of the fragment:

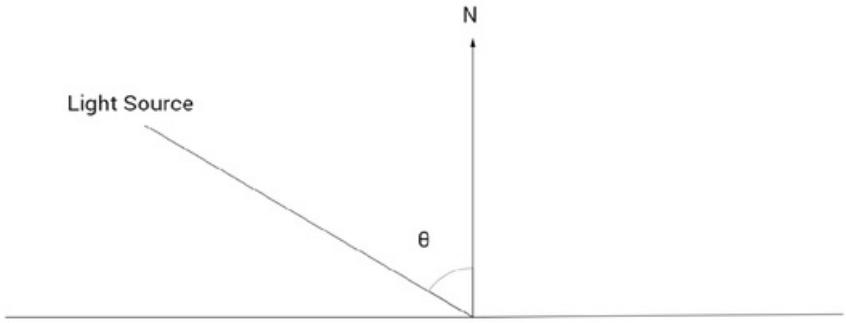


Figure 12 - Diffuse Lighting Model

The function that calculates an object's final colour in the Fragment Shader is outlined in ***Figure 13***.

```
vec3 calc_lighting(vec3 normal, Light light) {
        vec3 light_direction = normalize(light.position - fragpos_out);
        float diffuse = max(dot(normal, light_direction), 0.0);
        return light.colour * diffuse * uniform_colour.xyz * light.intensity;
}
```

Figure 13- Fragment Shader lighting calculation

Calculating the dot product requires 2 vectors to be known, the light direction and the vertex normal. Light direction is calculated by normalizing the difference between the light position vector and the fragment position vector. The light direction is used as an argument alongside the vertex normal in GLSL's dot function which calculates the dot product of input vectors. To avoid the diffuse component becoming negative, the max function is used such that the diffuse component will become 0.0 if the dot product becomes negative.

Rendering Text

Rendering text to the framebuffer is necessary in the development of the simulator, as one of the key project objectives is to have a functional User Interface. As with the geometry of the simulation (vehicles, walls, floor, etc), it is necessary to use OpenGL to achieve this. In addition to OpenGL, the FreeType library [27] is utilised to load TrueType font files from a file, into memory as a bitmap. This process is encapsulated within a new addition to renderer.cpp; the Text_Renderer class. The initialisation routine for this class uses the library to load the font file from a data directory within the project's file structure, and an OpenGL texture created for each glyph of the loaded font. It should be noted that the generated texture for each loaded character from the font-file only contains a red component, whereas typically, textures are created using RGB or RGBA arguments. Only the red internal format is used here as FreeType loads glyphs as 8-bit bitmaps, so each byte of the generated bitmap buffer can be mapped directly to a pixel in the generated texture [28].
A map is saved whereby the key is a GLchar byte value and the retrieved value is an instance of the Glyph structure outlined in ***Figure 14***.

```
struct Glyph {
        GLuint data;
        FT_Pos next_glyph_offset;
        vec2 glyph_size;
        vec2 bearing_offset;
};
```

Figure 14- Declaration of Glyph structure from renderer.h

The first attribute in this structure, data, is an OpenGL handle to a generated texture. After initialisation, this data handle can be used for binding the texture corresponding to the character to be drawn to a screen. The lifetime of this texture handle, and indeed the containing Glyph structure is expected to be for the duration of the simulator's execution. The next attribute is generated by FreeType and stored within the FT_Face type as the advance width, which specifies the pixel distance between the point of origin of one character and the point of origin of the subsequent character. Within the scope of this simulation, this attribute is used to correctly horizontally space characters being drawn to the screen. The final two attributes of the struct store the size of the glyph texture and its point of origin.

Like the other renderer classes, the rendering of text is encapsulated within a draw method, and is outlined in ***Figure 15***.

```
void Text_Renderer::draw(const std::string& msg, const vec2& position, bool centered, const vec4&
colour) {

	// ... Preamble

	for (GLchar c : msg) {
		Glyph g = glyph_map[c];

		float xPos = x + g.bearing_offset.x;
		float yPos = position.y - (g.glyph_size.y - g.bearing_offset.y);

		x += (g.next_glyph_offset >> 6);

		float vertices[6][4] = {
			{ xPos,                 yPos + g.glyph_size.y, 0.f, 0.f },
			{ xPos,                 yPos,                  0.f, 1.f },
			{ xPos + g.glyph_size.x, yPos,                 1.f, 1.f },
			{ xPos,                 yPos + g.glyph_size.y, 0.f, 0.f },
			{ xPos + g.glyph_size.x, yPos,                 1.f, 1.f },
			{ xPos + g.glyph_size.x, yPos + g.glyph_size.y, 1.f, 0.f }
		};

		glBindTexture(GL_TEXTURE_2D, g.data);

		glBindBuffer(GL_ARRAY_BUFFER, vbo);
		glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vertices), vertices);

		glDrawArrays(GL_TRIANGLES, 0, 6);
	}

	// ... Cleanup
}
```

Figure 15- Text Render functionality

This approach towards rendering text is adapted from [28]. The algorithm renders text by iterating through each character of the function's string argument and for each character, the quad displaying the glyph texture is positioned and resized and the handle to the glyph's texture data is bound, ready for rendering by the fragment Shader. One addition made in this implementation, not present in [28] is the ability to centre text by passing a boolean argument into the function.

Centring text is implemented by performing a first-pass iteration through each character of the input string to determine the total pixel width of the rendered text and then halving this value. The semi-width of the rendered text is then subtracted from the original x-position of the text had it been rendered left-aligned. This approach is shown in ***Figure 16***.

```
if (centered) {
	float x_ = x;
	for (GLchar c : msg)
		x_ += (glyph_map[c].next_glyph_offset >> 6);
	x -= ((x_ - x) * .5f);
}
```

Figure 16- Centring text to ease positioning of UI elements

As described in [28], it is necessary to bit-shift the offset value by 6 to convert the offset value to pixel width.

6.2.2 Braitenberg Vehicles

Vehicle Movement using Box2D

Box2D has two different object types which can be used to create physics simulations in 2D. Although the developed solution is rendered in 3D, only 2 axes need to be used to model vehicular movement; The XZ planes.

Fixtures define the properties of a Body on Box2D. For each vehicle, 5 fixtures are created to represent 4 wheels and the main vehicle body. Fixtures are assigned the following properties, as part of a FixtureDef object:

1. Density
2. Shape
3. Friction coefficient

Density in the context of Box2D determines how much mass there is to a fixture. By increasing the density, objects become slower and require more velocity to move. The four tyres of each vehicle in this simulation are assigned densities less than the density of the vehicle body. The shape of the fixture is used for collision detection. One of the challenges involved in integrating Bpx2D into the simulator was making sure that vertex data was to define cubes for the rendering component matched the vertices set in the physics portion of the codebase. Finally, the friction coefficient allows for velocity to be dampened depending on the surface a body is passing over. For the scope of this project, this coefficient is defined for all fixtures that compose a vehicle, although no surface coefficient is defined in the current implementation.

The most significant part of implementing Box2D such that it could be used to drive vehicular motion, was coming up with a way to combine the above elements in such a way that would result in convincing looking vehicle motion. The approach taken uses helper functionality adapted from [29] to compose a series of classes representing Tyres and the VehicleBody itself. The reason this functionality was used is that it provides a very convincing result while only requiring a modest amount of new code to be added into the existing codebase. In implementing vehicle physics in physics.hpp and physics.cpp, the implementation is kept as modular as possible, allowing for other vehicles beyond simple Braitenberg types to potentially be created in the future. We begin by examining the implementation of a Tyre class in ***Figure 17***. The constructor implemented in this project is based off the constructor presented in [29], however there are some differences that are added.

```
Tyre::Tyre(b2World* world, float max_forward_speed, float max_backward_speed, float max_drive_force,
float max_lateral_impulse) : max_forward_speed(max_forward_speed),
max_backward_speed(max_backward_speed), max_drive_force(max_drive_force),
max_lateral_impulse(max_lateral_impulse)
{
        b2BodyDef body_def;
        body_def.type = b2_dynamicBody;
        body = world->CreateBody(&body_def);

        b2PolygonShape polygonShape;
        polygonShape.SetAsBox(0.4, 0.4);
        b2Fixture* f = body->CreateFixture(&polygonShape, 1.f);

        b2Filter filter;
        filter.categoryBits = TYRE;
```

```
        filter.maskBits = ENV;
        filter.groupIndex = 0;

        f->SetFilterData(filter);

        body->SetUserData(this);
}
```

Figure 17- Constructor for Tyre

At the beginning of the constructor, a b2BodyDef object is instantiated and set as a dynamic body, before a reference to the BodyDef is passed as an argument into the helper function 'CreateBody'. The reason the BodyDef is assigned as dynamic is because the tyres in this simulation need to be able to react to external forces and can collide with other objects. Further, the dynamic body type allows the body to be positioned in the world manually, which will help the simulation class of the code assign this value.

Next, a b2PolygonShape object is instantiated and the utility function 'SetAsBox' defines the shape of this object with a width and height of 0.8 – the arguments to this function specify half-width and half-height, so the value of 0.4 is used in the code. A reference to the PolygonShape object is passed to the CreateFixture utility function, invoked from the member pointer variable *body. The second parameter to this function is a floating-point value specifying the density of the fixture being created, which in this case is kept as 1.

After this, a b2Filter object is created and the category and mask bits of this object are initialised using the 4-bit uint16 TYRE and ENV, shown in ***Figure 18***. Filters allow collisions between fixtures to be ignored, according to the category bits and mask bits set, as well as being able to identify the type of object involved in a collision within Box2D's BeginContact callback routine.

```
static uint16 TYRE    = 0x0001;
static uint16 VEHICLE = 0x0002;
static uint16 ENV     = 0x0004;
```

Figure 18- Bit values to be used for collision masking

The tyre is now initialised within a Box2D simulation. However, it requires some sort of external force to be applied to it to be able to move. The updateDrive function from [29] is implemented for the Tyre class as so:

```
void Tyre::update_drive(float desired_speed) {

       if      (desired_speed > max_forward_speed)  desired_speed = max_forward_speed;
       else if (desired_speed < max_backward_speed) desired_speed = max_backward_speed;

       b2Vec2 forward_normal = body->GetWorldVector(b2Vec2(0, 1));
       float current_speed = b2Dot(get_forward_velocity(), forward_normal);

       float force = 0;

       if      (desired_speed > current_speed) force =  max_drive_force;
       else if (desired_speed < current_speed) force = -max_drive_force;
       else                                    return;

       body->ApplyForce(force * forward_normal, body->GetWorldCenter(), true);
}
```

Figure 19- Moving a tyre forwards or backwards

The argument to this function is a floating-point value that determines the speed at which the tyre should be moving. The advantage of this approach is that each tyre within the simulation can move independently of the other tyres, closely resembling a Braitenberg Vehicle with its own independent motors.

At the beginning of this function, the desired_speed parameter is compared with the max_forward_speed and max_backward_speed member variables and assigned to either value if it is either greater or less than each of them, respectively. Next, the speed of the Tyre's body is determined by first calculating the direction of the tyre. This is achieved by using the GetWorldVector helper function which takes a vector representing local coordinates (in this case b2Vec2(0, 1), representing a forward direction locally) and transforms this vector into world-space coordinates. The result of this transformation is that body's current direction in world-space coordinates. The dot product of this direction and the current forward velocity is then calculated to determine the current speed of the body. If this calculated value for current speed is then less than or greater than the desired_speed parameter, the value of force is assigned to either positive or negative max_drive_force and used as the magnitude of the forward force applied in the ApplyForce method. If the current speed of the vehicle is not greater or less than the desired speed of the tyre, then no forward force is applied and the Tyre will naturally eventually slow down.

In its current state, the tyre would perpetually move in a direction without ever stopping (unless colliding with another dynamic body), so the updateFriction function from [29] is also adapted and implemented into physics.cpp as seen in

```
void Tyre::update_friction()
{
	b2Vec2 linear_impulse = body->GetMass() * -get_lateral_velocity();
	body->ApplyLinearImpulse(linear_impulse, body->GetWorldCenter(), true);

	float angular_impulse = .01f * body->GetInertia() * -body->GetAngularVelocity();
	body->ApplyAngularImpulse(angular_impulse, true);

	b2Vec2 forward_normal = get_forward_velocity();
	b2Vec2 drag_magnitude = -2 * forward_normal.Normalize() * forward_normal;
	body->ApplyForce(drag_magnitude, body->GetWorldCenter(), true);
}
```

Figure 20- Applying a force to resemble friction

The implementation outlined in [29] adopts a technique of cancelling lateral velocity to implement friction. First, the lateral velocity of the body is determined by computing the dot product between a localised (1, 0) vector transformed into world-space with the linear velocity of the Tyre's body. The negative value of this vector is then multiplied by the mass of the current vehicle body to determine the amount of force required to cancel out the lateral velocity of the vehicle. This impulse value is then used in the helper method: 'ApplyLinearImpulse', alongside a second argument indicating the position from which the impulse should be applied to (the centre position of the tyre in world-space coordinates). ApplyLinearImpulse is used instead of ApplyForce in this case as an impulse instantaneously affect velocity, whereas a force in Box2D applies it gradually over time. To slow the Tyre body down and prevent it from moving constantly in a direction, a drag force is next considered. This force is calculated by first determining the vehicle's forward velocity, and then multiplying this by a negative value representing the intensity of drag force and stored in drag_magnitude. This is then used in the 'ApplyForce' helper function to gradually reduce the forward velocity of the moving Tyre. ApplyForce is used in this case as the effect of drag needs to be gradual rather than instantaneous.

Another class representing a vehicle body is defined, using the implementation defined in [29] as a foundation.

```
void Vehicle::init(b2World* world, b2Vec2 position, float rotation, bool is_predator,
int index) {
	this->is_predator = is_predator;

	b2BodyDef body_def;
	body_def.type = b2_dynamicBody;
	body_def.position.Set(position.x, position.y);
```

```
        body = world->CreateBody(&body_def);
        body->SetAngularDamping(2);

        b2PolygonShape polygon_shape;
        polygon_shape.SetAsBox(8.f, 10.f);
        b2Fixture* fixture = body->CreateFixture(&polygon_shape, 0.1f);

        data = new VehicleData;
        data->instance_id = index;
        data->is_predator = is_predator;
        fixture->SetUserData((VehicleData*)data);

        b2Filter filter;
        filter.categoryBits = VEHICLE;
        filter.maskBits = VEHICLE;
        filter.groupIndex = 0;
        fixture->SetFilterData(filter);

        b2RevoluteJointDef joint_def;
        joint_def.bodyA = body;
        joint_def.enableLimit = true;
        joint_def.lowerAngle = 0;
        joint_def.upperAngle = 0;
        joint_def.localAnchorB.SetZero();

        // ... Tyre initialisation
}
```

Figure 21 - Vehicle Initialisation in Box2D

The Vehicle class includes a member bool variable to track its status as a predator or prey type vehicle, which is passed as one of the arguments to its initialisation routine shown in ***Figure 21***. This approach was taken due to its simplicity over an alternative implementation where a base vehicle interface defined and then sub-classed for predator and prey type vehicles. The reason the latter approach wasn't taken is that within the physics domain of the entire codebase of this project, all vehicles behave in the same manner and their collisions with the environment are identical. A b2BodyDef object is instantiated and as with the Tyre class, set as a dynamic body to allow for movement and collisions with other bodies and the position of the BodyDef is set using a b2Vec2 type object passed as a parameter. Being able to explicitly set the position of the vehicles is useful, as it allows for positions to be randomised at the beginning of each simulation (handled by simulation.cpp).

An instance of the VehicleData struct is created on the heap and its attributes initialised using the arguments passed into the function, instance_id and is_predator. This VehicleData instance is then cast to a void pointer type and used as an argument to the helper function 'SetUserData', allowing for these values to be retrieved within Box2D collision call-back routines as shown in ***Figure 22***, in which the VehicleData struct is retrieved in the event of a collision between two vehicles and used to insert a new entry into a set of VehicleData* pairs for processing by simulation.cpp.
In this way, an event-driven approach towards collision detection is adopted and physics.cpp and simulation.cpp require minimum communication between them.

```
void ContactListener::BeginContact(b2Contact* contact) {
        b2Filter fA = contact->GetFixtureA()->GetFilterData();
        b2Filter fB = contact->GetFixtureB()->GetFilterData();

        if (fA.categoryBits == VEHICLE && fB.categoryBits == VEHICLE) {
                VehicleData* dA = (VehicleData*)contact->GetFixtureA()->GetUserData();
                VehicleData* dB = (VehicleData*)contact->GetFixtureB()->GetUserData();

                vehicle_collision_events.insert(pair<VehicleData*, VehicleData*>(dA, dB));
        }
```

```
}
```

Figure 22- Box2D Collision Callback Function

Towards the end of ***Figure 21***, a b2RevoluteJointDef object is declared. This joint definition has anchor points set that define connections between bodies in a Box2D context. The joint_def object has a set of attributes which are initialised for each vehicle in the simulator, that determine the anchor points of the joint_def and how much the joint can move. The enableLimit bool attribute is set to true and both the lowerAngle and upperAngle attributes initialised to 0 – resulting in a static joint. The bodyA attribute tracks the first Box2D body to be connected to this joint, and in this case, is set as the main vehicle body, generated at the start of the initialisation routine. A similar attribute, bodyB, is defined for each instance of the Tyre class, and the member function of the Box2D World object, 'CreateJoint' is used to initialise that version of a joint_def object.

6.2.3 Maths Library

As the project relies heavily on interactions between 3D geometry, it was important to implement a set of classes and functions that can perform operations on vector and matrix types. The following types are defined:

Mathematical Construct	C++ Defined Type
2D Vector	vec2
3D Vector	vec3
4D Vector	vec4
4x4 Matrix	mat4

Figure 23 - C++ types defined within math.h

Primarily, the maths library is composed of overloaded operator definitions that facilitate being able to add, subtract, multiply or divide two custom types together. For example, for the vec3 type, representing a 3D vector the following definition is included:

```
friend vec3 operator + (const vec3& a, const vec3& b) {
     return { a.x + b.x, a.y + b.y, a.z + b.z };
}
```

Figure 24- Overloaded addition operator for vec3 type

This allows for a terser way of expressing vector arithmetic, and is defined for vec2, vec3, vec4 and mat4 types. For convenience and debugging purposes, the ostream operator >> is defined to easily print the value of any instantiated types to standard output.

Below, we outline the complete set of overloaded operators for each object type:

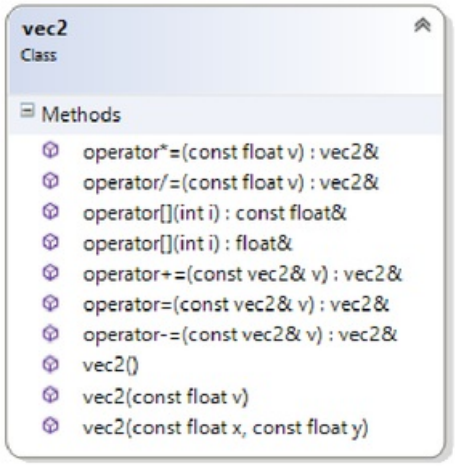


Figure 25- Class Diagram for vec2

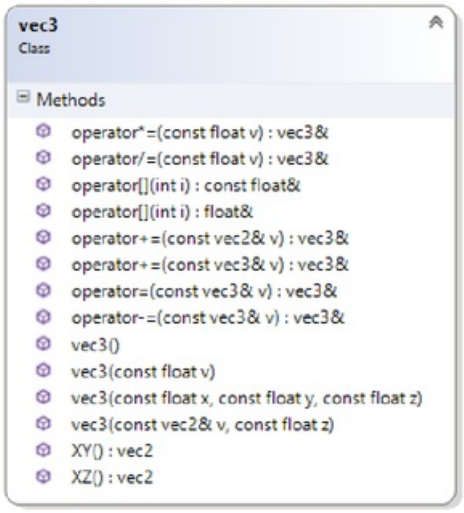


Figure 26- Class Diagram for vec3

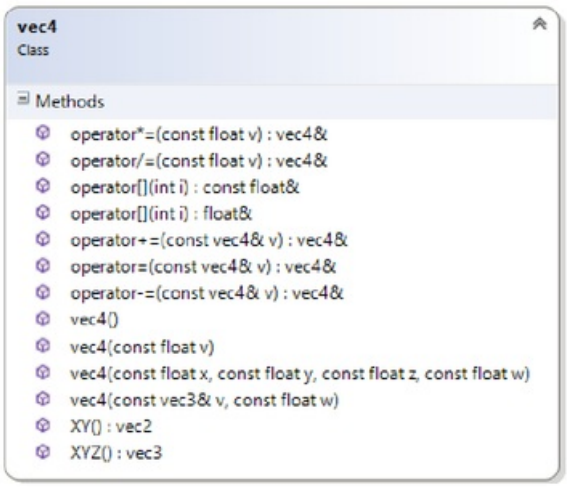


Figure 27- Class Diagram for vec4

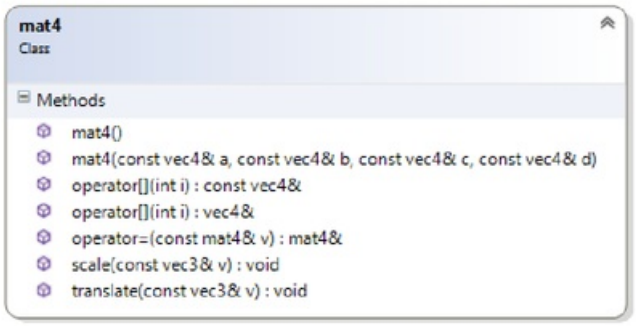


Figure 28- Class Diagram for mat4

Beyond simple arithmetical operations between objects of the same type, support for more complex operations was required.

Floating Point equality

Floating point values have limited precision, and depending on the compiler, may produce inconsistent results when compared against other floating point values [30]. This is problematic for areas in the codebase in which checking equality between floating point values cannot be avoided. To mitigate this issue, a function is included in the maths library that can check equality between two float values using an error factor.

```
bool almost_equal(float x, float y, float error_factor) {
	float diff = std::abs(x - y);
	return diff < error_factor;
}
```

Figure 29- Floating point equality function

This function takes two input floating point values and stores the absolute value of the difference between them (determined by subtracting the first from the second). The absolute value is necessary

here to avoid negative values – we cannot assume that all floating-point inputs to this function will be positive and that the first of the two values will always be larger as might be in the ideal case. The calculated absolute value is then compared against the provided error_factor argument, to see if it is less than the error factor. Essentially, the absolute value of the difference between the floating-point values must be less than the provided error_factor for equality to be true. The use of an accepted error_factor is a trade off in precision with the benefit of consistent float value equality checking.

Perspective Projection Matrix

A projection matrix is defined as so:

$$\begin{pmatrix} \frac{\cot\left(\frac{FOV}{2}\right)}{aspect} & 0 & 0 & 0 \\ 0 & \cot\left(\frac{FOV}{2}\right) & 0 & 0 \\ 0 & 0 & \frac{(near+far)}{(near-far)} & -1 \\ 0 & 0 & \frac{(near \cdot far)}{(near-far)} & 0 \end{pmatrix}$$

Figure 30- Composition of Projection Matrix *[31]*

The projection matrix is necessary in transforming vertex coordinates from a 3D coordinate system in world space, to a flat 2D plane, for display by the monitor. The coordinate system of the transformed vertices, Normalised Device Coordinates, maps the bottom left of the monitor to (0,0) and the upper right corner of the monitor to (1,1). Any newly mapped vertex that falls outside this range is clipped from view by OpenGL.

When constructing a Perspective Projection Matrix, there are 4 components to consider. The first is FOV (field of view), which determines the extent for how much of the scene is transformed into NDC. Secondly, the aspect ratio which is the ratio between the width and height of the rectangle forming the NDC space. Finally, the near and far clipping planes are also required.
These clipping planes determine how close and how far geometry must be for them to be excluded during the clipping processes. For example, a near-0 near clipping plane value would allow the camera to move close to geometry without any clipping taking place and a high value for the far plane would allow objects a long distance away from the camera to be drawn.

```
static mat4 perspective_matrix(float fov, float aspect, float near, float far) {
	float b = 1.f / tan(to_radians(fov) / 2.f);
	float a = b / aspect;

	return mat4{
		{ a, 0.f, 0.f, 0.f },
		{ 0.f, b , 0.f, 0.f },
		{ 0.f, 0.f, (far + near) / (near - far), -1.f },
		{ 0.f, 0.f, (far * near) / (near - far), 0.f }
	};
}
```

Figure 31- Function for calculating Perspective Projection matrix

View Camera Matrix

The View Matrix is defined to allow for movement around the scene. It is multiplied with the Perspective Projection matrix and an object's model matrix to form the final Model-View-Projection transformation matrix each frame. The View Matrix contains three vector components [32], that represent a camera's orientation in 3D space:

```
static mat4 view_matrix(const vec3& eye, const vec3& target, const vec3& up_rel) {
	vec3 forward = normalise(eye - target);
	vec3 right = normalise(cross_product(up_rel, forward));
	vec3 up = cross_product(forward, right);

	mat4 viewMatrix = {
		vec4(right.x, up.x, forward.x, 0),
		vec4(right.y, up.y, forward.y, 0),
		vec4(right.z, up.z, forward.z, 0),
		vec4(-dot_product(right, eye), -dot_product(up, eye),
			-dot_product(forward, eye),  1)
	};

	return viewMatrix;
}
```

Figure 32- Implementation of view matrix

- Forward Vector – The z-axis computed as a normalised vector between a position representing the camera's location and a position representing a target location.

- Right Vector – The x-axis computed as a normalised vector of the cross product between a vector representing desired up orientation and the previously calculated forward vector.

- Up Vector – The y-axis computed as the cross product between the forward vector and the right vector.

In this way, this function acts as a helper by which high-level camera attributes are converted to low-level vector representations of the camera's orientation, such that any set of input vectors can be provided. Wrapping the functionality to compute a view matrix in a static function means that the logic for computing the view matrix is decoupled too from the concept of a camera in the OpenGL scene, making it possible to create distinct types of camera by manipulating the three input parameters to the function.

Next, we review these parameters and explore how their values are changed within the Camera class. The three vectors required to generate a view matrix are as follows:

- Eye Vector – The position of the camera in world coordinates. Moving the camera around the simulation requires adding or subtracting offsets to each XYZ component of this vector.

- Target Vector – The position of the point the camera is looking at. Changing the direction of the camera is looking at requires adding or subtracting offsets to each XYZ component of this vector.

- Up Vector (up_rel) – Unit Vector representing the direction of the positive y-axis relative to the camera's current orientation. By default, this is set as (0, 1, 0) to indicate that moving upwards corresponds to an increase in y-axis values.

These vectors are then used in the Camera class' update routine to provide two different viewing modes for the simulation: a top-down aerial view of the simulation in which all vehicles can be viewed, and a follow-mode in which the camera will follow the currently targeted vehicle. The top-down mode provides a better general snapshot of vehicular interaction, while the follow-mode is best used to closely examine the individual behaviour of a vehicle.

The update routine is shown in ***Figure 33***.

```
void Camera::update(std::map<int, Transform>& transforms) {
	if (follow_vehicle) {
		vec2 direction = polar_to_cartesian(to_radians(transforms.begin()->second.rotation.y));
		direction *= target_distance;

		position_current = transforms.find(transforms.begin()->first)->second.position;
		position_current.y += target_distance;

		position_current.x -= direction.x;
		position_current.z -= direction.y;

		position_target = position_current + vec3{ direction.x, -target_distance, direction.y };
		orientation_up = vec3(0.f, 1.f, 0.f);
	}
	else {
		position_current = list_position_current[index_list_position_current];
		position_current.y = height;
		position_target = vec3{ 0.f, 0.f, 0.f };
		orientation_up = vec3(0.f, 1.f, 0.f);
	}

	matrix_view = shared::view_matrix(position_current, position_target, orientation_up);
}
```

Figure 33- Update method of camera class. Used to position and orient the scene's camera.

A simple conditional statement is used to determine whether the camera is following a vehicle or providing a top-down display of the simulation.

When following a vehicle, a vector offset from the target vehicle is used to determine an offset vector (direction). This offset is subtracted from the target vehicle's position to create a new position_current vector, used as the eye vector of the view_matrix. The target vector is determined by taking the newly calculated position_current vector, and adding the inverse of the computed direction offset to it, essentially projecting a ray forwards in the direction of the position vector. Finally, the up vector is reset to its default value of (0, 1, 0).

The scenario for viewing the simulation from an aerial top-down perspective is simpler to calculate. The current position of the camera is determined by sampling a pre-defined position from a list of potential camera positions, representing fixed-width positions surrounding the boundary of the simulator. The index used to sample this position from the stored array is changed by either pressing the left or right arrow keys. The height of this sampled position is then tweaked using the member variable 'height' of the Camera class. This allows the altitude of the aerial view of the simulator to be tweaked using the up and down arrow keys. The target of the camera view is set to the centre of the simulator area, at origin position (0, 0, 0), making the camera always look in this direction. As before the up vector is reset to its default value.

The camera class ultimately aims to abstract the complexity of tweaking view_matrix attributes directly, and with this current implementation could be extended to support different viewing modes beyond vehicle following and an overview display.

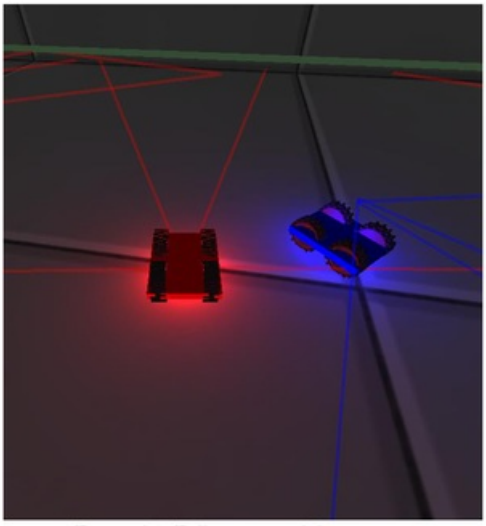


Figure 34- Follow view of the camera

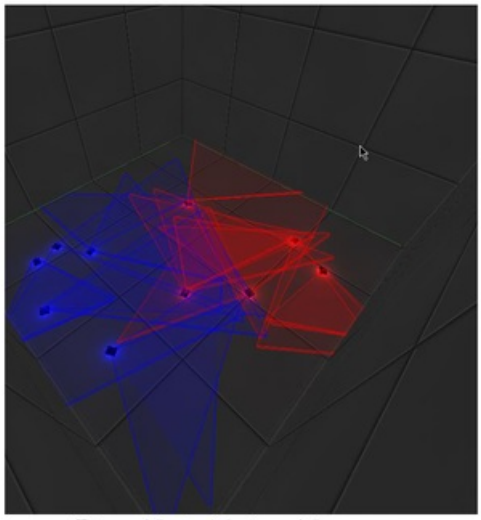


Figure 35- Aerial view of the camera

7. Testing: Verification and Validation

7.1 Performance Testing

Performance testing the developed simulation was important in making sure the program ran at a steady frame rate and didn't consume too many system resources at runtime. A higher frame rate ensures the user does not perceive any stuttering caused by too few frames being rendered per second.

This testing was conducted by generating new vehicles up to the maximum permitted by the simulator, and by monitoring system memory, CPU usage and GPU usage while leaving the simulation running for one minute. It is unlikely that the user will want the maximum permitted count of vehicles active at any point running the program, but even-so being able to guarantee a steady frame rate for this upper threshold implies the simulator will run successfully for fewer vehicles. To be certain, the number of vehicles is incremented by 10 and a new test issued for each launch, up to the vehicle maximum.

To carry out the performance testing on the simulator, the Performance Profiler and constituent tools from Microsoft Visual Studio 2015 were used. Within this feature, it is possible to select either Memory profiling, CPU profiling or GPU profiling.

GPU Performance Runs

Each performance result lists lowest frame rate, highest frame time and GPU usage:

Vehicle Quantity	Maximum GPU Usage (%)
10	21.6
20	34.2
30	45.3
40	57.8
50	74.0

Figure 36- GPU Performance across different executions of the simulator with varying vehicle quantity

CPU Performance Runs

Each performance result lists CPU usage:

Vehicle Quantity	Average CPU Usage (%)
10	17.325
20	17.475
30	18.575
40	19.55
50	19.4

Figure 37- CPU Performance across different executions of the simulator with varying vehicle quantity

Memory Performance Runs

When evaluating the memory usage of this simulation, we are particularly interested in heap consumption by objects representing a vehicle. We begin by identifying the per-vehicle attributes from the simulation class:

- Transform[] – Array of Transform structs
- Tyre *[] – Pointer to Array of Tyre structs
- VehicleData – Struct of extra vehicle attributes

As before, we examine the effect of increasing the vehicle count by an increment of 10 on the resultant memory usage.

Number Vehicles	Object Type	Count	Size (Bytes)
10	Vehicles.exe!Transform[]	10	1,440
	Vehicles.exe!Tyre	40	800
	Vehicles.exe!Tyre *[]	10	160
	Vehicles.exe!VehicleData	10	80
20	Vehicles.exe!Transform[]	20	2,880
	Vehicles.exe!Tyre	80	1600
	Vehicles.exe!Tyre *[]	20	320
	Vehicles.exe!VehicleData	20	160
30	Vehicles.exe!Transform[]	30	4,320
	Vehicles.exe!Tyre	120	2,400
	Vehicles.exe!Tyre *[]	30	480
	Vehicles.exe!VehicleData	30	240
40	Vehicles.exe!Transform[]	40	5,760
	Vehicles.exe!Tyre	160	3,200
	Vehicles.exe!Tyre *[]	40	640
	Vehicles.exe!VehicleData	40	320
50	Vehicles.exe!Transform[]	50	7,200
	Vehicles.exe!Tyre	200	4,000
	Vehicles.exe!Tyre *[]	50	800
	Vehicles.exe!VehicleData	50	400

Figure 38- Memory Performance across different executions of the simulator with varying vehicle quantity

The user may increase or decrease the number of active vehicles used for a simulation, it is also important to check that data allocated on the heap is successfully freed when a vehicle is removed to ensure no memory leak. To carry out this test, the simulation was started with 20 active vehicles and then vehicles are removed. A snapshot of the simulator's heap space is taken prior to removal of vehicles and after removal of vehicles. We compare the heap space from the different snapshots below:

Object Type	Count Diff.	Size Diff. (Bytes)	Count	Size
Vehicles.exe!Transform[]	-10	-1,440	10	1440
Vehicles.exe!Tyre	-40	-800	40	800
Vehicles.exe!Tyre *[]	-10	-160	10	160
Vehicles.exe!VehicleData	-10	-80	10	80

Figure 39- Comparison of heap sizes when vehicles removed during execution

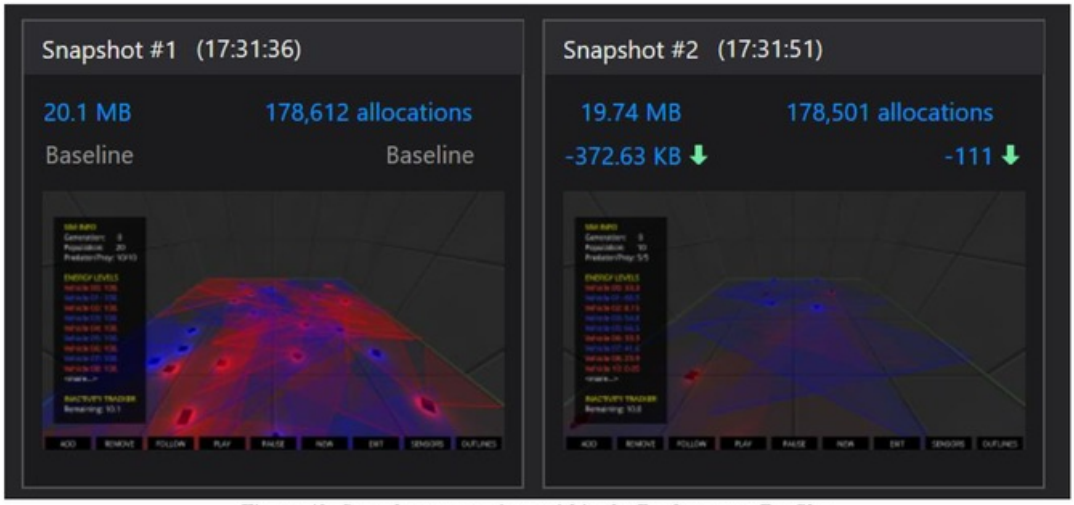


Figure 40- Snapshot comparison within the Performance Profiler

7.2 Acceptance Testing

As part of the acceptance testing stage, we verify whether the developed solution meets the requirements specified in 5.3 Acceptance Criteria. These tests aim to match the behaviour and functionality of the simulation against the criteria specified.

Acceptance Criteria	Result
The 3D scene should suffice for the user to understand how the vehicles are behaving and should provide a simple graphical representation of the artificial brain simulation.	Launching the simulation results in successful loading and generation of a 3D rendered scene with predator and prey vehicles. When the simulation is running, the vehicles move around fluidly, and the visualisation of the sensors helps understand different vehicle behaviour.
Vehicles powered by Braitenberg brains should be clearly identifiable and it should be possible to distinguish these from other vehicles.	The Braitenberg Vehicles used in the simulation are colourised per their designated Predator or Prey class. Lighting is used to further emphasise this colourisation.
The Braitenberg vehicles should act per their Predator-Prey class designation.	Predator Vehicles move towards Prey vehicles when they are scanned by the vehicle's sensors. Prey vehicles move away from Predator vehicles they detect as being nearby.
The GUI must be clear and easy to use.	The GUI is composed of a series of buttons and an information window to view the status of the simulation. When the mouse cursor moves over a button, it is highlighted to indicate that it can be pressed.
Each button should be intuitive and correctly convey its function.	The buttons are large and a colour that stands out against the 3D rendered view is chosen. The font used for each

	button is consistent with the design choices made for the information status tab.
A standalone C++ library with utility functions for physics calculations.	External 2D physics library Box2D is used for physics calculations.
It should be fully unit tested, and these tests packaged with the library as a reference for the user.	The math module of the developed solution is fully unit-tested, and the development environment configured to automatically execute these unit tests after successful compilation.

Figure 41- Acceptance Testing results. Each of acceptance criteria are verified.

7.3 Unit Testing

A set of unit tests were created for the developed mathematics module. Such tests guarantee that new changes to the project's codebase do not break existing functionality, and that the developed routines within this module act per the expectations of the developer.

The reason unit testing is important for the math module portion of the codebase is because the classes and functions contained within are heavily used elsewhere in the codebase. In this way, the developed math module forms the backbone of the product.

To facilitate the executions of the unit tests, a utility function was created responsible for the output formatting of the test's status. This utility function takes the test index, test description and test result as its input and returns a string with the formatted test result as its output, making use of the C++ standard libraries: stringstream and iomanip in doing so [33] [34].

```
string test_result(int num, const string& check, bool test) {
	std::stringstream s;
	s << setw(6) << setfill(' ') << to_string(num) + " - ";
	s << left << setw(28) << setfill(' ') << check;
	s << setw(6) << setfill(' ') << (test ? "PASS" : "FAIL");
	s << endl;
	return s.str();
}
```

Figure 42- Utility function used within unit testing module

An output log for a single execution of the unit tests is shown below. In this instance, all tests have passed.

```
2D Vector type checks:

  1 - initialise                 PASS
  2 - instance addition          PASS
  3 - instance subtraction       PASS
  4 - instance scalar (multiply) PASS
  5 - instance scalar (divide)   PASS
  6 - array data access          PASS
  7 - proxy array access         PASS
  8 - equality (true)            PASS
  9 - equality (false)           PASS
 10 - addition                   PASS
 11 - subtraction                PASS
```

```
 12 - scalar (multiply)           PASS
 13 - scalar (divide)             PASS

3D Vector type checks:

  1 - initialise                  PASS
  2 - instance addition           PASS
  3 - instance subtraction        PASS
  4 - instance scalar (multiply)  PASS
  5 - instance scalar (divide)    PASS
  6 - array data access           PASS
  7 - proxy array access          PASS
  8 - equality (true)             PASS
  9 - equality (false)            PASS
 10 - addition                    PASS
 11 - subtraction                 PASS
 12 - scalar (multiply)           PASS
 13 - scalar (divide)             PASS


4D Vector type checks:

  1 - initialise                  PASS
  2 - instance addition           PASS
  3 - instance subtraction        PASS
  4 - instance scalar (multiply)  PASS
  5 - instance scalar (divide)    PASS
  6 - array data access           PASS
  7 - proxy array access          PASS
  8 - equality (true)             PASS
  9 - equality (false)            PASS
 10 - addition                    PASS
 11 - subtraction                 PASS
 12 - scalar (multiply)           PASS
 13 - scalar (divide)             PASS

Math operation checks:

  1 - vec3 cross product          PASS
  2 - vec2 dot product            PASS
  3 - vec3 dot product            PASS
  4 - vec4 dot product            PASS
  5 - vec2 magnitude              PASS
  6 - vec3 magnitude              PASS
  7 - vec4 magnitude              PASS
  8 - vec2 normalise              PASS
  9 - vec3 normalise              PASS
 10 - vec4 normalise              PASS
 11 - vec2 determinant            PASS
 12 - float comparison            PASS
 13 - vec2 distance               PASS

Type sizes:

  1 - sizeof vec2                 PASS
  2 - sizeof vec3                 PASS
  3 - sizeof vec4                 PASS
  4 - sizeof mat4                 PASS
```

Figure 43- Output for unit tests. All this instance, all tests have passed

8. Discussion: Contribution and Reflection

The performance testing carried out was useful in providing an accurate picture of how the simulator was consuming resources during run-time. The GPU testing showed the most increase in usage of resources as more vehicles were added, by 10-15% for every 10 vehicles added. This would have to be resolved as more features get added to the developed simulation, as other constructs within the created world would also vie for GPU usage. What the GPU usage results show, is that there is a significant need to change the chosen architecture by which OpenGL commands are executed. As we recall, a set of renderer classes were created to abstract the complexity and differences between mandated OpenGL commands when drawing different primitives. Opting for this architecture style improved development time by making erroneously used OpenGL commands less probable, but seems to have increased GPU workload beyond an acceptable threshold. For example, in ***Figure 44*** taken from the Cube Renderer's draw function, we see that there are five OpenGL commands that issue uniform data to the currently bound Shader. The cube renderer's draw function is invoked for every vehicle active in the simulation, and these commands resend the same data to the GPU. To better improve GPU usage, one possibility would be to externalise the process by which uniform data is sent to the GPU, and having uniform the values shared between all renderer draw methods.

```
shader.set_uniform("model", utils::gen_model_matrix(size, position, rotation));
shader.set_uniform("view", camera.matrix_view);
shader.set_uniform("projection", camera.matrix_projection_persp);
shader.set_uniform("uniform_colour", colour);
shader.set_uniform("light_position", vec3{ 0.f, 30.f, 0.f });
```

Figure 44- A portion of the code-base issuing repeated OpenGL commands.

CPU performance analysis shows a much smaller increase as the number of vehicles is increased. Such an increase likely incorporates the CPU overhead of issuing the OpenGL commands in the first place, though this could likely be avoided by reusing Shaders to avoiding wasting CPU ticks on unnecessary rebinding. GPU usage seems to show a similar linear increase as the number of vehicles is incremented however the magnitude of each increase is much larger – nearing a 15-20% increase for every 10 vehicles. These results are somewhat unexpected as very simple data types are used to represent the geometry in the scene, and effort was made to decouple as much OpenGL code from object structure definitions as possible, to allow for instantiation of data types as normal within C++. One of the potential reasons for this significant increase in GPU usage could be the repeated call to glBufferSubData in some draw functions within renderer.cpp. We can see this is being used in ***Figure 45*** where the repeated call updates the new set of transformed vertex data are used to define the transformed sensor within the triangle renderer's draw command. This could be resolved by instead transforming the triangle vertices a Shader, to avoid reissuing the vertex data, though performance improvements could also be achieved by replacing the call to glBufferSubData with glBufferData, as the entire VBO is updated, rather than a sub-region of the Buffer Object [35].

```
void Triangle_Renderer::draw_3D_coloured(const Camera& camera, const vec3& a, const
vec3& b, const vec3& c, const vec4& colour)
{
	// ... Bind Context

	glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vec3) * 3, &values);

	// ... Render and Cleanup
}
```

Figure 45 Repeated vertex data upload each draw call

The unit tests proved to be successful in testing whether existing functionality within the maths portion of the codebase remained as expected when new features were added to the program. However, this is by no means an exhaustive set of tests to guarantee this across the entire scope of the project. Ideally, another source file containing a new set of unit tests to check whether the order of execution for drawn elements might prove to be effective in debugging the OpenGL portion of the codebase and for the physics portion of the codebase. The difficulty in writing effective Unit Tests for OpenGL applications is the difficulty in receiving Shader outputs once static data is sent to the GPU. This might be accomplished using Shader Storage Buffer Objects, which can read from Shader stages – allowing the CPU to retrieve modified GPU data. However, implementing this new type of Buffer into the codebase might require a notable change to the way in which initialisation for the renderer classes is carried out and for now is best left as a 'should-have' feature for future improvements.

The acceptance tests were carried out to verify whether the acceptance criteria specified when defining the solution approach were met. For each, we analyse how the simulator met the specified criteria and consider ways in which these criteria might then be expanded for future iterations of development.

The 3D scene should suffice for the user to understand how the vehicles are behaving and should provide a simple graphical representation of the artificial brain simulation.

The implementation of the simulator successfully provides a way in which the simulation can be viewed, and the OpenGL API was used to leverage this. One of the limitations of the current approach however is that there are only two different camera modes to provide a perspective of the current simulation. Further, a GUI was implemented to try to convey information about the simulator to the user, however the amount of interaction possible with the interface and the amount of information displayed is currently limited and could be expanded to allow for more information and further ways of interacting and configuring the simulation.

Vehicles powered by Braitenberg brains should be clearly identifiable and it should be possible to distinguish these from other vehicles.

The renderer classes implemented during the development of this simulator were expanded enough to allow for cuboids of different dimensions to be rendered using distinct colours. This was successful in being able to identify vehicles according to their predator-prey designation, however the scope of this acceptance criteria has not been fully met yet. The only type of vehicles implemented in the current version of the simulator are Braitenberg vehicles, powered by reaction to their environment, whereas the acceptance criteria implies more than one type vehicle being available to simulate. Future developments of the simulator would therefore include a greater range of vehicle types being made available to the user. Considering the scope to which just the implementation of the Braitenberg Vehicles could be fully explored (reaction to other types of energy source, such as heat and smell), the implementation of other vehicle types than the Braitenberg type may not be such a high priority.

The Braitenberg vehicles should act per their Predator-Prey class designation.

The predator-prey vehicles in the simulation successfully moved and reacted with other vehicles relative to their designated class. The implementation approach that accomplished this disparity utilises a point-triangle intersection function using the position of another vehicle and the current vehicle's defined sensor boundaries. One of the successes of this approach was preventing predator vehicles from sensing other predator vehicles, and prey vehicles from detecting other prey vehicles. However, this approach is rather simplistic and doesn't accurately model the diverse types of ocular receptor an actual organism might have, nor does it accurately model the strength of a visual signal (another vehicle). This limitation would thus need to be promptly overcome in a future iteration of development, to provide a more realistic simulation of how the Braitenberg vehicles might behave if modelled using actual hardware.

The GUI must be clear and easy to use.

Efforts were made to ensure that the GUI component of the simulator was clear and easy-to-use. One of the key successes of the implementation of this component was making the size of the text labels and buttons scale according to the resolution of the window running the simulator. During the implementation of this component, a design decision was made to include the energy level of each vehicle prompted by a recognised need to provide the user with a metric by which the predator-prey vehicles could be evaluated. In addition to this information, it was also decided that the inactivity timer tracking the absence of motion by any vehicles in a simulation could be displayed as part of the GUI. Although this information was successfully included into the interface, there is currently little the user would want or need to use this information for. Thus, we identify a limitation of the GUI in in the relevance of the information it displays to the user. Future improvements in this aspect of the simulation would involve finding alternative ways to visualise real-time metadata regarding the active simulation.

Each button should be intuitive and correctly convey its function.

The implementation of the buttons within the simulation is contained within a separate GUI class to avoid the main simulation class containing code that is unrelated to the functionality and control of active vehicle instances. The decision to separate this functionality out improved the ability to maintain the core simulation class during the development of the project. A visual cue was successfully added to each button to indicate whether a user's cursor was above each button, as a prompt to indicate the button components could be clicked. This functionality was achieved by using a point-quadrant intersection function which is invoked for each button to check whether the position of the mouse cursor intersects the bounds of each button. One of the limitations of this approach however is that repeatedly invoking this function for new clickable GUI components that potentially get added in the future, may becoming expensive in terms of utilising the CPU during the main update loop.

A standalone C++ library with utility functions for physics calculations.

The standalone library developed for this objective emphasises utility functions more for vector and matrix operations than functionality for physics calculations. The latter instead was provided by the Box2D library. The implemented library was divided into a header file containing class definitions for vector and matrix types, as well as declarations of standalone functions utilising these types, while the source file contained the definitions for these functions. Re-using this functionality in other projects related to the simulator however is cumbersome, due to the library having to be recompiled as part of that project's executable.

It should be fully unit tested, and these tests packaged with the library as a reference for the user.

The developed mathematics library was unit tested to guarantee that functions produced the expected outputs given different inputs. This was successful in providing a guarantee that the functionality of this library was valid, and that new functionality didn't affect the behaviour of existing functions within the library. The output of the unit tests conducted for the testing gave a succinct overview of all the functions that had passed, which proved to be successful throughout the development of the project in validating whether the functionality of the simulator was correct. However, the unit testing only covered the maths library and didn't include any tests that confirmed whether other aspects of the simulator. This limitation didn't necessarily impact the development of the project in its first iteration as a functioning simulator, but may need to be overcome as the project continues to develop and as its complexity begins to increase.

9. Social, Legal, Health & Safety and Ethical Issues

When developing the Project Initiation Document, a risk assessment was carried out to identify any potential health and safety issues. In this section, we examine these issues is some more detail, and then explore social, legal and ethical issues that may stem from the development of this project.

The first hazard is the use of display screen equipment. The prolonged exposure to the blue-light emitted by a monitor could disrupt sleeping patterns, resulting in long-term fatigue [36]. The course of action identified to mitigate this risk is to make use of monitor-brightness settings throughout the day to reduce exposure to the light.

The use of portable tools and equipment was identified as a hazard. Specifically, the risk involved in using a Laptop to develop the project given that device's portability. While it is necessary to conduct work on the laptop due to it having a GPU, there are risks involved with this – especially when transporting the laptop onto the University of Reading campus. One of the risks involved is potential theft of the Laptop and subsequent loss of information and project progress as a result. This can be mitigated largely by making extensive use of cloud-storage services to back up the project data, so that backups can be downloaded in the case of a theft. This also mitigates against the risk of laptop failure preventing progress in the development of the project.

Another hazard associated with the development of this project is lone working / working out of hours. Specifically, working for extended periods of time on the project may be beneficial to its development in the short-term, however the fatigue and stress caused by doing this could cause greater problems in the long-term. To mitigate this, the active working hours allocated to development of the project is limited to eight hours each day. Ideally, these eight hours are split across the morning and afternoon, to avoid working late at night and potentially disrupting the sleep cycle.

One of the potential social issues surrounding the development of this project is the potential for users who speak and read different languages other than English to not understand the user interface elements displayed in English. One of the potential ways this issue could be addressed is by emphasising simplicity of visual prompts such as labels used in the interface to make it easier for a non-English speaker to understand the meaning and functionality of the GUI elements. This concern also impacts the extensibility of the simulator, whereby effort may be made in designing code such that strings of text are externalised to their own file and loaded into the GUI upon initialisation. This approach would allow for translated versions of this file for different languages to be created in the future as the need may arise to target an international audience of users.

The Simulator provides a framework by which new robot types can be tested. Considering the ethical implications of developing such a piece of software raises some concerns over how it may be misused. Specifically, caution about the simulator being adapted to develop nefarious robots should be exercised whereby a user could design and adapt robot types that become adept at inflicting actual harm on other people. Although limited in its current limitation to simple Braitenberg vehicles, the framework and current codebase could be used and adapted by another as a starting point for development of their own version of the simulator that is more capable of developing malicious robots.

The use of the GitHub service to store the repository of version controlled source code comprising the implementation of the project raises some potential legal concerns over the project being copied by others and claiming it as their own original property. This is mitigated by marking the GitHub repository as private, ensuring only the Developer has access during the development phase and that the repository contents remain unlisted from search engines.

10. Conclusion and Future Improvements

The objectives established at the beginning of this project were as follows:

The primary objective for the product is to serve as a learning tool, by which users can gain an intuition of how complex behaviours can emerge from distinct types of Braitenberg vehicle.

A secondary objective is for the simulation to function as a basic prototyping tool, for those wishing to create Braitenberg vehicles using actual hardware.

The development of the project was overall successful in producing a Braitenberg Predator Prey simulator that met the objectives specified. We find that that it is possible to create a range of vehicular behaviours just by tweaking a vehicle's sensor-actuator connections. This could be used in the future to model new behaviours by creating new configurations of vehicle neuron. One of the most exciting areas of research to apply this simulator to in the future is modelling the behaviours of simple organisms in nature, by completely implementing their neuronal structure. This would extend the scope of the simulator from its focus on using Braitenberg Vehicles to convey this point.

The observation of the predator-prey dynamic was made possible by explicitly creating multiple classes of vehicle and hardcoding behaviour such that predator vehicles would seek prey, and prey would flee. This approach is relatively simple, and it does ensure the primary objective of the simulator being an effective learning tool is met. However, there are many ways this approach should be improved upon. The first way it could be improved is by making no distinction between vehicle types in the implementation itself. Specifically, the vehicles should be allocated as a predator or prey type vehicle purely based on its neuronal configuration. This approach would be much more data-driven and would give the user greater opportunity to interact with the simulation. The second way the implementation of predator-prey vehicles could be improved is to generalise the concept of a vehicle sensor. In the Discussion, it was mentioned that support could be added for distinct types of energy – which could further be developed as an additional implementation by allowing each vehicle to have more than two sensors. Naturally, the implementation of connections between sensor and actuator would also need to be generalised such that any number of 'neurons' can be added to a vehicle.

In developing the rendering component for the simulation, OpenGL proved to be an appropriate choice of API for drawing to a window's framebuffer as it offered versatility in the way the architecture of the solution could be defined, and provided a large amount of control over the look and feel of the developed project. One of the key learning points in using OpenGL in a C++ environment is that the Object-Oriented Programming Paradigm is not necessarily the most effective to use, as constructor definitions invoked implicitly upon object declaration should assume that an OpenGL context has been setup, and that the necessary GL handles are actively bound for the set drawing functionality. We therefore find that a more suitable way of encapsulating OpenGL functionality is to create structures with explicit initialisation and destruction routines that are invoked when needed within the program. However, this might be best for a project with less developers as it defies a ley C++ principle, RAII (Resource Acquisition Is Initialisation).

The approach taken towards vehicle locomotion using Box2D was more effective at meeting the acceptance criteria than the prototyped solution using simple vector offsets. The decision to use an existing physics library rather than implementing the functionality from scratch was invaluable to the development and progression of the project. Naturally, some development time is taken up by learning to use the library and reading through its documentation, as well as integrating it within the project. Ultimately, this shows that making use existing functionality from external libraries should always be considered a feasible option when developing any software, although caution should be

exercised for large, complex and relatively new libraries. The advantages and disadvantages of both should be discussed by relevant participating stakeholders before informing this decision.

In the Discussion, it was recognised that the repeated invocations of the point-quad intersection function to test whether a user had clicked a button may restrict the ability to extend the functionality of the GUI without impacting performance. This could be resolved in future iterations by recursively subdividing the window area in sub-regions, forming a quad-tree. The point-quad intersection functions could then be calculated by performing the point-quad intersection for each level of the tree, until reaching a leaf node of the tree representing the boundary formed by a button component. This approach would require some amount of processing to form the tree in the first place, but as more buttons get added to the simulator to extend its functionality, the trade-off in generating the tree and preventing repeated unnecessary quadrant collision checks for every active button may yield greater CPU performance.

The current limitations of the GUI of the simulator was addressed in the Discussion. This prompts a use-case in the future for extending this component of the simulator by adding more components to it that increase the possible interactions that the user can have with it. One set of features that could be added and made available through the user-interface is the ability to save and load simulations to a file on the user's hard-drive, allowing them to reload the saved simulation files on a subsequent launch of the simulator back into memory. This might be successfully accomplished by enumerating all class instances of simulator-specific entities, and defining an output function that makes use of the C++ standard ostream class to write their attributes to a file.

Another component of the developed solution that could be improved in the future is the set of unit tests developed to validate the maths library. It was recognised that the unit tests in their current state are suitable for validating this single component of the full solution, but do not tests other parts of the codebase. It would therefore be useful to extend the unit testing conducted on the solution so that other parts of the codebase are tested. For example, the physics module developed to implement vehicular movement could have a set of unit tests written for it that check whether a vehicle in each start position will reach an end position given a configured set of parameters about the forces operating in that world.

The limitation of the developed maths library not being suitable for use in other projects due to its strong integration within the current simulator project as a set of source code that is compiled as part of another executable was earlier recognised. This limitation should ideally be overcome by instead compiling the mathematics functionality developed as a standalone .dll or .lib file, that would get linked into the compilation project of future projects, but not recompiled. Alternatively, this functionality could be refactored into a header-only library, such that only the header file would need to be distributed when using its functionality in other projects. The header-only approach would be particularly useful when using the library for different Operating Systems, whereby external linkage would typically require distinct types of library file to be provided.

11. References

[1] V. Braitenberg, Vehicles - Experiments in Synthetic Psychology, The MIT Press, 1986.

[2] V. Braitenberg, "Vehicles - Experiments in Synthetic Psychology," The MIT Press, 1984, pp. 3-5.

[3] V. Braitenberg, "Vehicles - Experiments in Synthetic Psychology," The MIT Press, 1984, p. 6.

[4] V. Braitenberg, "Vehicles - Experiments in Synthetic Psychology," The MIT Press, 1984, pp. 15-16.

[5] W. Harwin, "Braitenberg vehicle and 'Preditor/Prey' simulator," [Online]. Available: http://www.personal.reading.ac.uk/~shshawin/LN/braitenberg.html. [Accessed 2 April 2017].

[6] Khronos Group, "Fixed Function Pipeline," 10 April 2015. [Online]. Available: https://www.khronos.org/opengl/wiki/Fixed_Function_Pipeline. [Accessed 11 April 2017].

[7] Khronos Group, "GLUT - The OpenGL Utility Toolkit," [Online]. Available: https://www.opengl.org/resources/libraries/glut/. [Accessed 11 April 2017].

[8] R. Mitchell, "Begin Robotics Simulations," University of Reading, [Online]. Available: https://www.reading.ac.uk/UnivRead/vr/OpenOnlineCourses/Files/Simulation2/BeginRobotics Simulationindex2.pdf. [Accessed 11 April 2017].

[9] FutureLearn, "Begin Robotics - Online Course," FutureLearn, [Online]. Available: https://www.futurelearn.com/courses/begin-robotics. [Accessed 11 April 2017].

[10] R. Mitchell, "BRAITENBERG ROBOT," [Online]. Available: http://www.reading.ac.uk/UnivRead/vr/OpenOnlineCourses/Files/Simulation2/robotbrait.html. [Accessed 11 April 2017].

[11] CareerRide, "MVC - Benefits of MVC pattern," CareerRide, [Online]. Available: http://www.careerride.com/MVC-benefits.aspx. [Accessed 2017 April 11].

[12] F. D and K. L, "Evolution of Adaptive Behaviour in Robots by Means of Darwinian Selection.," *PLoS Biol,* vol. 8, no. 1, p. e1000292, 2010.

[13] The Khronos Group, "The OpenGL Graphics System: A Specification," 24 October 2016. [Online]. Available: https://khronos.org/registry/OpenGL/specs/gl/glspec45.core.pdf. [Accessed 02 April 2017].

[14] The Khronos Group, "OpenGL Platform & OS Implementations," [Online]. Available: https://www.opengl.org/documentation/implementations/.

[15] Microsoft, "Getting Started with Direct3D," [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/hh769064(v=vs.85).aspx.

[16] L. Gomila, "Simple and Fast Multimedia Library," [Online]. Available: https://www.sfml-dev.org/. [Accessed 12 April 2017].

[17] "WebGL and OpenGL Differences," Khronos Group, [Online]. Available: https://www.khronos.org/webgl/wiki/WebGL_and_OpenGL_Differences. [Accessed 13 April 2017].

[18] S. Willems, "GPU hardware info database launchpad," [Online]. Available: http://opengles.gpuinfo.org/. [Accessed 13 April 2017].

[19] Google, "Android Studio Features | Android Studio," Google, [Online]. Available: https://developer.android.com/studio/index.html. [Accessed 13 April 2017].

[20] Apple Inc., Apple Inc., [Online]. Available: https://developer.apple.com/xcode/ide/. [Accessed 2017 April 13].

[21] Microsoft, "Visual Studio | Developer Tools and Services | Microsoft IDE," Microsoft, [Online]. Available: https://www.visualstudio.com/. [Accessed 13 April 2017].

[22] iforce2d, "Anatomy of a collision - Box2D tutorials - iforce2d," iforce2d, 7 May 2014. [Online]. Available: http://www.iforce2d.net/b2dtut/collision-anatomy. [Accessed 13 April 2017].

[23] E. W. Weisstein, "Barycentric Coordinates," Wolfram Research, Inc., 28 March 2003. [Online]. Available: http://mathworld.wolfram.com/BarycentricCoordinates.html. [Accessed 21 April 2017].

[24] C. Jules, "Accurate point in triangle test," Totologic - Blogspot, 25 January 2014. [Online]. Available: http://totologic.blogspot.co.uk/2014/01/accurate-point-in-triangle-test.html. [Accessed 21 April 2017].

[25] O. (.-t. Trutner, "geometry - Efficiently find points inside a circle sector," 4 April 2013. [Online]. Available: http://stackoverflow.com/questions/13652518/efficiently-find-points-inside-a-circle-sector. [Accessed 21 April 2017].

[26] iforce2d, "Sensors - Box2D tutorials - iforce2d," iforce2d, 14 July 2013. [Online]. Available: http://www.iforce2d.net/b2dtut/sensors. [Accessed 21 April 2017].

[27] The FreeType Project, "The FreeType Project," [Online]. Available: https://www.freetype.org/. [Accessed 19 April 2017].

[28] J. De Vries, "Text Rendering," [Online]. Available: https://learnopengl.com/#!In-Practice/Text-Rendering. [Accessed 20 April 2017].

[29] iforce2D, "Box2D C++ tutorials - Top-down car physics," iforce2D, [Online]. Available: http://www.iforce2d.net/b2dtut/top-down-car. [Accessed 15 April 2017].

[30] D. Goldberg, "What Every Computer Scientist Should Know About Floating Point Arithmetic," *ACM Computing Surveys,* vol. 23, no. 1, pp. 5-41, 1991.

[31] "The Perspective and Orthographic Projection Matrix," Scratchapixel, [Online]. Available: https://www.scratchapixel.com/lessons/3d-basic-rendering/perspective-and-orthographic-projection-matrix/building-basic-perspective-projection-matrix. [Accessed 6 April 2017].

[32] NVIDIA, "Background Information," NVIDIA, [Online]. Available: https://docs.nvidia.com/gameworks/content/technologies/desktop/nv3dva_background.htm. [Accessed 29 April 2017].

[33] "<iomanip>," cplusplus.com, [Online]. Available: http://www.cplusplus.com/reference/iomanip/. [Accessed 5 April 2017].

[34] "stringstream," cplusplus.com, [Online]. Available: http://www.cplusplus.com/reference/sstream/stringstream/.

[35] Khronos Group, "Vertex Specification Best Practices," Khronos Group, 22 February 2016. [Online]. Available: https://www.khronos.org/opengl/wiki/Vertex_Specification_Best_Practices. [Accessed 15 April 2017].

[36] NHS, "Do iPads and electric lights disturb sleep?," NHS choices, 23 May 2013. [Online]. Available: http://www.nhs.uk/news/2013/05May/Pages/Do-iPads-and-electric-lights-disturb-sleep.aspx. [Accessed 1 May 2017].

12. Appendices

12.1 Appendix 1: Project Initiation Document (PID)

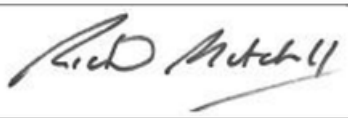
Individual Project (CS3IP16)

Department of Computer Science

University of Reading

Project Initiation Document

PID Sign-Off

Student No.	**21010731**
Student Name	**Oliver Reynolds**
Email	**O.Reynolds@student.reading.ac.uk**
Degree programme (BSc CS/BSc IT)	**BSc CS w/ Year in Industry**
Supervisor Name	**Prof Richard Mitchell**
Supervisor Signature	
Date	**29/09/16**

SECTION 1 – General Information

Project Identification

1.1	**Project ID** (as in handbook)
1.2	**Project Title**
	Predator-Prey robots
1.3	**Briefly describe the main purpose of the project in no more than 25 words**
	To develop a simulator of Braitenberg Vehicles that demonstrate a predator-prey behaviour. This behaviour must be achieved by developing a simple neural network that influences the signal sent to a vehicle's actuator from its sensors.

Student Identification

1.4	**Student Name(s), Course, Email address(s)** e.g. Anne Other, BSc CS, a.other@student.reading.ac.uk
	Oliver Reynolds, BSc CS w/ Year in Industry, O.Reynolds@student.reading.ac.uk

Supervisor Identification

1.5	**Primary Supervisor Name, Email address** e.g. Prof Anne Other, a.other@reading.ac.uk
	Prof Richard Mitchell, R.J.Mitchell@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

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.**

Valentino Braitenberg developed a series of robotic vehicles that exhibit behaviour dependent on the connections between their sensors and the actuators driving their wheels. It was shown that a wide variety of different vehicle behaviours could be demonstrated by changing these connections, such that some wheeled vehicles displayed human emotions such as curiosity. Motion is determined by the construction of connections between its sensors and its motors. New behaviours can emerge by modifying the construction of these circuits – and more complex configurations resemble a neural network [1].

Researchers have been strongly interested in using nature-inspired algorithms to solve new classes of problem. One such example of a pattern exhibited by living species in nature is the predator-prey coevolutionary dynamic, whereby a population of predator and prey organisms co-adapt to meet the pressures of not being able to seek prey, or to defend from predators [2].

OpenGL is a low-level graphics API that allows developers of languages such as C++ to render graphics to a framebuffer. The API itself is a specification, and the implementation is dependent on the vendor of a system's graphics device driver. Since its 3.2 release, the specification has seen some significant changes to its underlying render pipeline, that allows developers to write their own Shaders and gain access to greater control over OpenGL data transfers to a graphics device using specialised buffers [3].

These three areas of research will be pivotal to the development of the project and will be utilised to inform design and implementation decisions.

References:

[1] Necsi.edu. (2017). Evolution. [online] Available at: http://necsi.edu/projects/evolution/co-evolution/pred-prey/co-evolution_predator.html [Accessed 1 May 2017].

[2] Braitenberg, V. (1986). Vehicles. Cambridge, Mass.: MIT Press.

[3] Opengl.org. (2016). OpenGL SDK. [online] Available at: https://www.opengl.org/sdk/docs/ [Accessed 28 Sep. 2016].

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.

- **AI:** Implement artificial brains using Braitenberg vehicles and Neural Networks. The vehicles may have some aspect of memory, so they can recall where obstacles were. Allow the artificial brains to be swapped out and configured at run-time.
  - Allow weights between sensory input and simulated comparators to be configured by the user, so that it is possible to observe the relation between a vehicle's weights and its behaviour.
  - Allow connections between sensory inputs and comparators to be changed by the user, so that is it possible to observe the relation between a vehicle's neural network and its behaviour.
- **Physics:** Create a physically plausible simulation, in which the vehicles will operate. A framework will be created with functions implementing different types of forces. A Newtonian approach will be adopted in these forces affecting the acceleration and angular acceleration of the vehicles.
  - Position will be determined using Euler Integration using Acceleration and Velocity to improve the accuracy of the simulation.
  - Rotation will be determined using Euler Integration using Angular Acceleration and Torque to improve the accuracy of the simulation.
  - Friction against wheel modelled using relation between wheel's surface area applied to ground, velocity, and friction constant of the surface.
- **GUI:** Create a functional interface for the user to interact with the simulation.
  - The simulation will feature simple prompts to direct the user to specific features.
  - One portion of the GUI will be dedicated to displaying textual information about the current simulation state, to inform the user.
- **OpenGL Scene:** Represent the simulation using graphics implemented using the OpenGL API.
  - The sensors will be represented using sectors of a circle, rendered using a Vertex-Geometry-Fragment pipeline of GLSL Shader programs. Alpha blending will be utilised to overlay render of sensor field of view over scene geometry to emphasise sensor-stimulus events.
  - The vehicle will be represented using a simple quad, rendered using a Vertex-Geometry-Fragment pipeline of GLSL Shader programs.
  - Obstacles in the scene will be represented using arrays of quads set in different positions. A future extension of this would be user-defined n-gons that still work using the sensor-collision functionality.
- **Maths:** Vectors and Matrix types will be the backbone of the graphical components of the simulation. A small library providing implementations of various operations using these types will speed development time.
  - Define 2D, 3D and 4D vector types to aid the implementation of high-end simulation logic.
  - Define 4D matrix type to aid the implementation of high-end simulation logic.
  - Derive and implement Orthographic 4D matrix routine, with which to provide the simulation a birds-eye view, scaled 1:1 with the viewport resolution of OpenGL context window – this will be used for the construction of an object's final Model-View-Perspective matrix for GL rendering.
  - By extension, implementation of Perspective 4D matrix routine, with which to provide a more user-friendly 3D view of the simulation.

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.
	This project aims to overcome the complexity of testing different robotic behaviours when analogue circuits are the driving force for a robot's motors. The simulation will provide a basic 3D environment in which wheeled robots will move around automatically. The movement of these robots will be directly affected by their environment. Each robot will have a set of sensors, which the user will be able to modify and control. At run-time, the user will be able to pause and resume an initialised simulation. In a paused state, the user will be able to configure the robot's wiring connecting sensors to motors. This configuration mode will have a distinct display separate from the simulation. The robot's neural network will be modified by connecting lines between nodes. This will be flexible enough to accommodate complex networks and should allow for basic behaviours to be developed by the simulated robot. The simulation must be cross platform and not over-rely on external libraries. The project will use OpenGL 3.2+ It is not feasible to create a perfectly accurate simulation of reality, accounting for all forces that might affect vehicular movement. The project will aim to achieve most of these. Further, the first iteration of this simulation will be constrained to wheeled 'Braitenberg' style robots, however, it is acknowledged that the simulation can be extended in the future to account for other styles of robotic locomotion.
2.4	**Describe the social, legal and ethical issues that apply to your project. Does your project require ethical approval?**
	The project does not require ethical approval. The code for the project will be backed up using a public GitHub repository. A software license may need to added to control the way in which people who view the code can actually use it. However, this software license cannot contradict the University's Coursework Submission Policy.
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
	No purchases necessary.
2.6	**State whether you need access to specific resources within the department or the University e.g. special devices and workshop**
	No specific resources necessary.

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	**Background Research**		
1.1	Ongoing research to supplement implementation of new features.	3+	Updated logbook to reflect research.
2	**Analysis and design**		
2.1	Construction of Project Initiation Document and breakdown of key objectives.	1+	The PID will adapt at the project continues, though the key objectives will remain mostly static.
3	**Develop prototype**		
3.1	Maths implementation.	4	A comprehensive and thoroughly unit tested mathematics library, to augment the project with necessary routines.
3.2	Physics implementation.	4	The movement of the robot in the simulation resembles that of actual robots in reality.
3.3	GUI implementation.	2	The interface is clear and intuitive – all icons make sense, even for a new user.
3.4	OpenGL scene implementation.	4	An interactive 2D display of the simulation, featuring graphical entities.
3.5	AI implementation	5	User can adjust robot's sensors and neural network at run-time.
4	**Testing, evaluation/validation**		
4.1	Unit Testing	1	Unit tests will execute as a separate binary within the project folder.
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**	**26.5+**	

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: 26/09/2016												
	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													
Ongoing research to supplement implementation of new features.													
2 Analysis/Design													
Construction of Project Initiation Document and breakdown of key objectives.													
3 Develop prototype.													
AI implementation													
Physics implementation.													
GUI implementation.													
OpenGL scene implementation.													
Maths implementation.													
4 Testing, evaluation/validation													
5 Assessments													

AREA HEALTH AND SAFETY RISK ASSESSMENT FORM (RA1)

Assessment Reference No.		**Area or activity assessed:**	Completion of final year Project: Robotics Simulation
Assessment date	**28/09/16**		
Persons who may be affected by the activity (i.e. are at risk)	**Oliver Reynolds**		

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	X	**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	X
5.	Display screen equipment	X	**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. For more complex activities or projects you are advised to use Form RA2.*

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	Display Screen Equipment	Reduction of prolonged exposure to monitor's blue-light mitigated by auto-adjustment of monitor brightness settings according to time of day.		X		No further action necessary.
11	Use of portable tools / equipment	The loss or theft of the laptop computer used to develop the project is reduced by avoiding its transportation onto University Campus when necessary. All data is backed up regularly to an external device.			X	No further action necessary.
29	Lone working / work out of hours	Extensive stress caused by lone working and working out of hours is controlled by working only within a designated set of hours, only on a weekday.		X		No further action necessary.
Name of Assessor(s)		Oliver Reynolds	**SIGNED**			
Review date		28/09/16				

jp010731_Final_Year_Report

GRADEMARK REPORT

FINAL GRADE

/100

GENERAL COMMENTS

Instructor

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23

PAGE 24

PAGE 25

PAGE 26

PAGE 27

PAGE 28

PAGE 29

PAGE 30

PAGE 31

PAGE 32

PAGE 33

PAGE 34

PAGE 35

PAGE 36

PAGE 37

PAGE 38

PAGE 39

PAGE 40

PAGE 41

PAGE 42

PAGE 43

PAGE 44

PAGE 45

PAGE 46

PAGE 47

PAGE 48

PAGE 49

PAGE 50

PAGE 51

PAGE 52

PAGE 53

PAGE 54

PAGE 55

PAGE 56

PAGE 57