|  |
| --- |
|  |
| **Evolving path finding and emergent group movement using genetic programming** |
| Computing Individual Project 2012/13 |
|  |
| **Jak Boulton** |
| **10026037**  **Supervisors:**  Dr. Ian Wilson  Dr. Paul Jarvis |

**Acknowledgements**

First and foremost I would like to extend my utmost gratitude to my main supervisor; Dr. Ian Wilson, for keeping me on track throughout the development of the project. He has also been a generally great lecturer for the past three years.

Secondly, I would like to thank my second supervisor Dr. Paul Jarvis for his helpful feedback.

The third person I would like to thank is Dr. Paul Angel for his excellent lecturing skills. He is the reason I chose to enrol at the University of Glamorgan. He is also owed my thanks for the permission to use his *OpenGL* framework within my project.

Finally I would like to thank my family and friends for their continued support over the years,

*Thank you.*

|  |
| --- |
|  |

Contents

[1 Introduction 5](#_Toc351463256)

[1.1 Objectives 6](#_Toc351463257)

[1.2 Requirements 7](#_Toc351463258)

[1.3 Project Plan 8](#_Toc351463259)

[2 Research and Analysis 9](#_Toc351463260)

[2.1 Path finding 10](#_Toc351463261)

[2.1.1 The A\* Algorithm 10](#_Toc351463262)

[2.1.2 Group Movement 13](#_Toc351463263)

[2.2 Artificial Intelligence: Learning and Evolution 18](#_Toc351463264)

[2.2.1 Artificial Neural Networks 18](#_Toc351463265)

[2.2.2 Back Propagation Algorithm 21](#_Toc351463266)

[2.2.3 Genetic Algorithms 22](#_Toc351463267)

[2.3 Rendering API 24](#_Toc351463268)

[2.3.1 OpenGL 24](#_Toc351463269)

[2.3.2 DirectX 24](#_Toc351463270)

[2.4 Additional Research 25](#_Toc351463271)

[2.4.1 Genetic Programming 25](#_Toc351463272)

[3 Design and Development 29](#_Toc351463273)

[3.1 Design Considerations 30](#_Toc351463274)

[3.2 Design Overview 34](#_Toc351463275)

[3.2.1 System Design 34](#_Toc351463276)

[3.2.2 AI Design 35](#_Toc351463277)

[3.3 Detailed Design 35](#_Toc351463278)

[3.3.1 OpenGL Framework 37](#_Toc351463279)

[3.3.2 System Design 38](#_Toc351463280)

[3.3.3 AI Design 39](#_Toc351463281)

[3.3.4 Map 44](#_Toc351463282)

[3.3.5 Resources & Utilities 44](#_Toc351463283)

[3.4 Screen Designs 45](#_Toc351463284)

[3.5 Justifications 46](#_Toc351463285)

[3.5.1 Graphics API 46](#_Toc351463286)

[4 Implementation 47](#_Toc351463287)

[4.1 Implementation Plan 48](#_Toc351463288)

[4.2 Final Implementation 48](#_Toc351463289)

[4.2.1 OpenGL Framework 48](#_Toc351463290)

[4.2.2 System 49](#_Toc351463291)

[4.2.3 Libgp 51](#_Toc351463292)

[4.2.4 A\* 55](#_Toc351463293)

[4.2.5 Evolving Path finding 56](#_Toc351463294)

[4.2.6 Evolving Group movement 60](#_Toc351463295)

[4.2.7 Resources & Utilities 62](#_Toc351463296)

[4.3 Run-Time Walkthrough 63](#_Toc351463297)

[5 Testing 69](#_Toc351463298)

[5.1 Test Plan 70](#_Toc351463299)

[5.2 Testing the System 70](#_Toc351463300)

[5.3 Testing the AI 73](#_Toc351463301)

[6 Evaluation 80](#_Toc351463302)

[6.1 Evaluation Plan 81](#_Toc351463303)

[6.2 Evaluation against Specification 81](#_Toc351463304)

[6.3 Improvements 86](#_Toc351463305)

[6.4 Future Additions 87](#_Toc351463306)

[6.5 Conclusion 89](#_Toc351463307)

[Bibliography 90](#_Toc351463308)

[Appendices 92](#_Toc351463309)

[Appendix I - LSEPI 92](#_Toc351463310)

[Appendix II – Objective Setting Performa 104](#_Toc351463311)

[Appendix III – Mark Weighting Form 105](#_Toc351463312)

[Appendix IV – Project Diary 106](#_Toc351463313)

# 1 Introduction

This report consists of the design and development of an application containing a three dimensional environment used to demonstrate the process of both the evolution of a path finding agent and the emergence of group movement using genetic programming to generate and evolve the agent’s behaviours. The path finding agent generates and evolves a program in order to navigate from a start to a destination point in a randomly generated map. The best program found is compared to an A\* (A Star) path to check for efficiency. The emergence of group movement is done through the evolution of a program to be executed by multiple agents who share a similar goal. In the project, the Painted Desert problem (Koza, 1992) is used to demonstrate how genetic programming can create emergent group movement and behaviour. The target audience for the project is teachers and lecturers who teach AI as a topic, it will allow them to demonstrate how AI is able to evolve within an environment which is easy to engage with for the students.

## 1.1 Objectives

1. Produce a document containing a detailed and extensive compilation of all the research needed to complete the project.
2. Produce a document containing a detailed design of the project.
3. Implement the project based on the design in the previous section.
4. Compile a test plan for each of the classes within the project and proceed to test the application.
5. Produce a document containing an evaluation of how successful the project was based on the overall aim and objectives.

*LSEPI Objectives*

1. To investigate the licencing issues of using software libraries within the development of the project.
2. To investigate any legal issues that may arise with the use of previously published algorithms that may be relevant to the project.

## 1.2 Requirements

The system must complete its goal by demonstrating AI agents which evolve. The group of requirements below have been used to prove the level of success of the project.

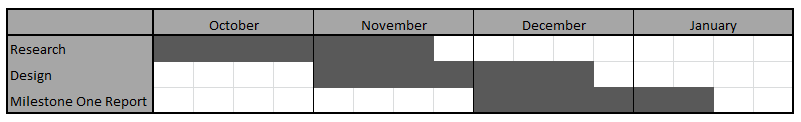
The completed system must be able to:

1. Display a basic three dimensional scene,
2. Display debug window,
3. Display a graphical representation of a path-finding AI using A\*,
4. Display a graphical representation of an AI generated using the method of genetic programming,
5. Allow the user to control when the genetic program creates a new population of programs,
6. Display the operations of the genetic programming method within the debug window,
7. Display the program generated using the genetic programming method used within the graphical representation,
8. Display a group of AIs generated using the genetic programming method showing emergent group movement,
9. Allow the user to switch between scenarios with ease,
10. Allow the user to alter the camera position and viewing direction,
11. Allow the user to restart the demonstration,
12. Allow the user to exit the application.

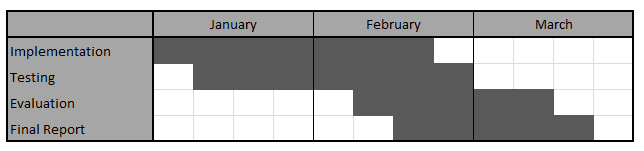
## 1.3 Project Plan

The project plan is where each of the tasks were initially planned to be started and completed in order to finish the research, design, implementation and report for the project before the deadline. Below is a simple time scale diagram detailing where each of the tasks were initially planned to be started and completed. The charts below show the proposed start and completion time of all the key sections of the project.

Milestone one plan:



Milestone two plan:



# 2 Research and Analysis

This section of the report contains all of the research areas which were required in order to begin the design and development stages of the project. The topics below have each been used to gain a larger degree of knowledge within each of the specific areas before the design and development of the project could begin. To begin the research we will look into path finding methods.

## 2.1 Path finding

The original title chosen for this project was “Group Movement and Path finding”. Although the title has progressed further to form a project which is based mainly around the evolution of AI agents through the use of genetic programming; path finding was a necessary section of research.

Path finding is the process of searching a map using an algorithm in order to find a path between two points (Matthews, 2002). The AI pathfinder is given the start point and destination point and computes a path between the two. This allows the characters to move across the 3D space in order to complete tasks. There are two approaches which have been used within path finding algorithms; directed and undirected. Undirected approaches to path finding will simply search the space with no planning or direction until a path is found. These path finding algorithms are not used to model believable behaviour since they normally spend most of their time moving down dead end routes. The main algorithms classed as taking an undirected approach are Breadth-first and Depth-first searches. The directed approaches to path finding do not blindly search the space. These algorithms have a method for measuring progress, a way of assessing the cost of getting from the starting point to each of the nodes that have been searched. The main strategies employed by algorithms using the directed approach are:

* *Uniform cost search* – the next node to be searched will always be the node which costs the least to get to from the starting node.
* *Heuristic search* – meaning that the algorithm will estimate how far each specific node is from the destination, more accurately; the cost of getting to the destination from the node.

### 2.1.1 The A\* Algorithm

The A\* Algorithm (pronounced A-Star) was the obvious first choice for a path finding algorithm and must be part of any path finding based research since it is the most widely used path finding algorithm for grid based search spaces (Ross Graham, 2003). The project uses a grid based approach to the map design meaning that A\* would be the algorithm of choice.

There are two main algorithms which employ the directed approach to path finding; Dijkstra and A\*. Dijkstra’s algorithm and A\* are very similar since they use a cost strategy in order to find an optimal path. The difference between these two algorithms is that A\* uses a heuristic search as well as the uniform cost search in order to search in the direction of the destination whereas Dijkstra’s algorithm simply searches from the starting point until it finds a path to the destination. The best way to describe the difference between the two algorithms is by comparing diagrams of the complete search spaces.

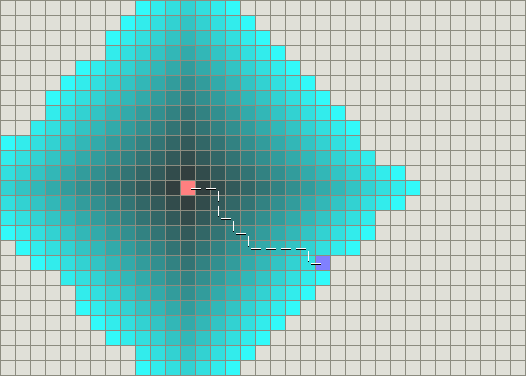


Figure 2.1.1 - 1 *Dijkstra’s algorithm performing basic path finding.*

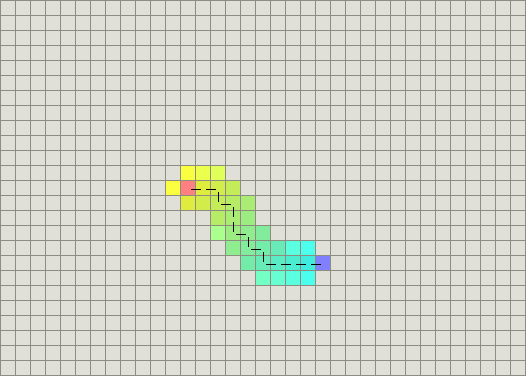


Figure 2.1.1 - 2 *A\* performing basic path finding.*

*Figures 2.1.1 - 1 and 2.1.1 – 2 are from* (Patel, 2012)*.*

The coloured area within the diagrams above show the area the algorithm had to search in order to find the path; the A\* algorithm found an optimal path without searching a great deal of the map showing that using heuristic search aids the path finding process greatly by minimising the amount of the map that needs to be searched before an optimal path can be found. Due to all of the reasons stated above; A\* was chosen to be used within the project. The next question is; how does the algorithm work?

The A\* algorithm requires that the map is pre-process or prepared before it can work. This means that the map must be broken down into points, locations or parts known as nodes. These nodes can take many forms when in use but the main two which are most widely used are waypoints or polygons (of a navigation mesh). The nodes are used to measure the search progress. These nodes hold their location on the map along with three main attributes; fitness, goal and heuristic (or f, g and h). The three attributes are defined as follows:

* **g** is the cost of getting from the start node to the current node.
* **h** is the heuristic, it is the estimated cost of getting from the current node to the destination node. This limits the number of nodes that need to be searched before a solution is found.
* **f** is the sum of the previous two attributes, the best estimate of the cost of the path through the current node. The lower the f value, the more efficient the path.

The pseudo-code for the algorithm is as follows:

1. *Let P = starting point.*
2. *Assign f, g and h values to P.*
3. *Add P to the Open list. At this point, P is the only node on the Open list.*
4. *Let B = the best node from the Open list (i.e. the node that has the lowest f value).*
   1. *If B is the goal node, then quit – a path has been found.*
   2. *If the Open list is empty, then quit – a path cannot be found.*
5. *Let C = a valid node connected to B.*
   1. *Assign f, g and h values to C.*
   2. *Check whether C is on the Open or Closed list.*
      1. *If so, check whether the new path is more efficient (i.e. has a lower f value).*
         1. *If so update the path.*
      2. *Else, add C to the Open list.*
   3. *Repeat step 5 for all valid children of B.*
6. *Repeat from step 4.* (Rabin, 2002)*.*

As the pseudo-code suggests; the algorithm maintains an open and a closed list. The open list is used to store the nodes which have been given f, g and h attribute values but have not been fully explored yet while the closed list contains the nodes which have been fully search and therefore do not need to be searched again. A node is classed as *fully explored* when all the nodes it has links to have been processed (the f, g and h attributes have been calculated and the parent node has been set).

### 2.1.2 Group Movement

Group Movement is how the AI move together as a collective unit and how they work together to reach their goals as a group. It would be easy to develop AI which work alone, but much more difficult to create AI which work intelligently as a group in order to eliminate their enemies. Group Movement is a massive area to cover considering the number of different elements that can be discussed including squad based movement, tactics and combat. AI Game Programming Wisdom (Rabin, 2002) contains several sections describing and explaining many areas of group movement which will be used within the project. This document contains more brief descriptions of the elements found within AI Game Programming Wisdom but all of the information found within the book will be put into account when designing and developing the AI group movement that will be found within the project.

The AI within the project need to show believable tactical behaviour, algorithms which generate tactics for the computer-controlled squads are needed to give this illusion. This is where strategic and tactical reasoning makes an appearance. There are many ways the AI (using tactics generation algorithms) can use the data available to them in order to generate ways to out-smart their enemies. The AI are able to use the data already at their disposal such as waypoint graphs (used primarily for path finding) in order to measure how good a specific position on the map is in terms of its strategic quality. On top of that; other data sources can be specifically designed and developed to give the AI the information they need in order to produce intelligent behaviour, below are a group of the possible additions:

* Visibility,
* Influence Maps,
* Grouping,
* Leadership Styles,
* Planned Maneuvers,
* Formations.

The first, visibility can be added to the AI decision making allowing them to see enemies and allies within a vision cone. Visibility gives the AI characters much more believability since they cannot attack or engage something that they can’t see (Liden, 2002). Influence Maps are used to give an AI character a risk value of entering an area of the map. They will check the map and dependant on their current state may engage or avoid areas controlled by enemy units in order to reach their goal. An example might be that the character has low health, meaning that its goal would be to find a medic, health station or health pick-up (based on the mechanics of the game). In this example the AI character would most likely want to avoid a combat situation at all costs in order to reach its goal.

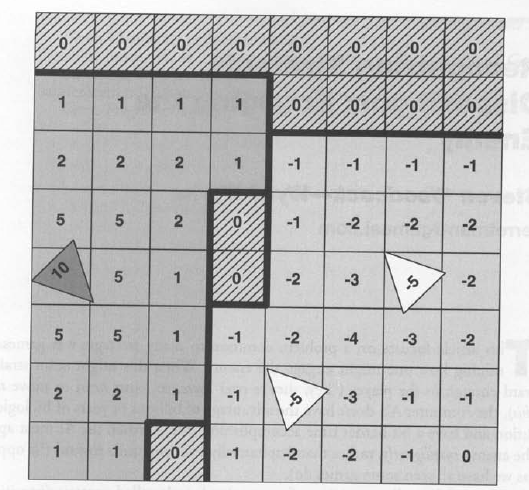


Figure 2.1.2 - 1 *Basic influence map.*

Figure 2.1.2 shows a basic influence map where the squares (or cells) containing a negative value are under enemy control and the squares containing a positive number are under control by allied units. The squares containing zero are neutral, not controlled by any of the units. A unit could avoid an area with negative influence in order to avoid enemies (Woodcock, 2002).

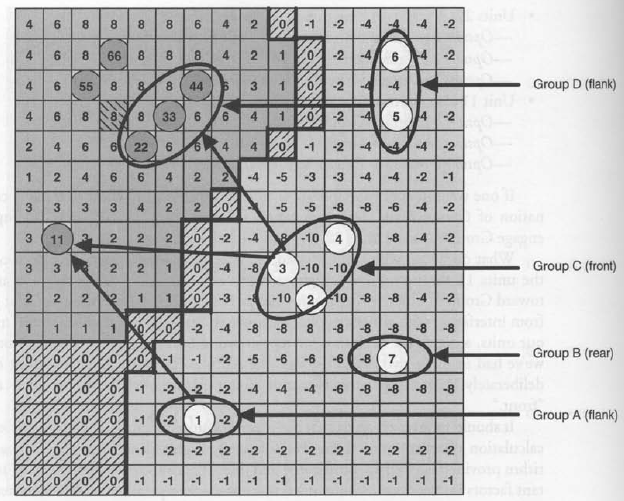


Figure 2.1.2 – 2 *A more complex influence map showing group tactics.*

Influence maps can also be used within grouping to generate tactics for multiple units as seen in figure 2.1.2 - 2 to gain a tactical advantage over the enemy. The units are grouped and all given separate tasks to complete. Flanking is also made a much easier task using influence maps since the AI characters could use the ‘edge’ of the enemies influence as a wall and follow it around before engaging from the target’s rear.

Leadership styles are the next tactical concept to cover; this is how the individual units within the groups are split up and how they are structured. There are two main ways this can be done; centralised and de-centralised group organisation (Sterren, 2002). Figure 2.1.2 -3 shows how the two different types of group organisation compare.

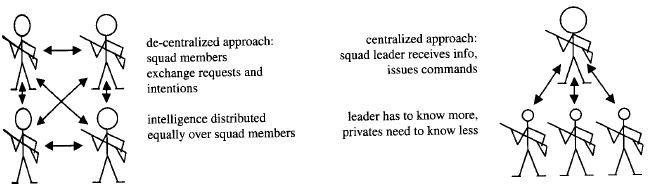


Figure 2.1.2 – 3 *De-centralised vs centralised.*

Planned maneuvers is about how the squads engage with the enemy or simply move from position to position while being either more strategic or generally more intelligent; no one wants AI characters that run across open terrain mindlessly (Sterren, 2002). How do the groups decide what to do in terms of a specific maneuver and how does the squad perform the maneuver once it has been chosen? Figure 2.1.2 – 4 should show how it is done in a basic form.

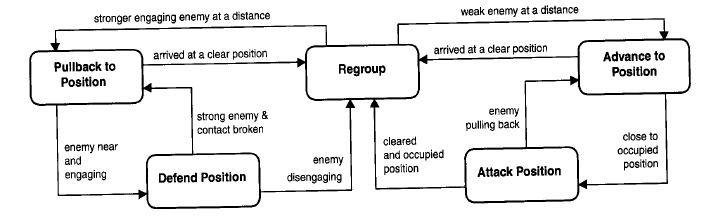


Figure 2.1.2 – 4 *A State machine defining the sequence of maneuvers.*

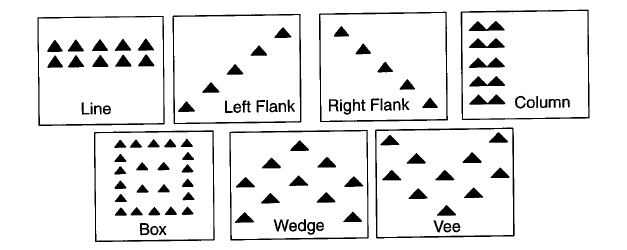


Figure 2.1.2 – 5 *Common formations.*

Formations are how the units within the squads are physically positioned based on the positions of others within the squad. These can be used to both gain a tactical advantage on the enemy and make squads look more organised and intelligent (Champandard, 2010).

## 2.2 Artificial Intelligence: Learning and Evolution

This section of the research was invaluable since the aim of the project was to develop AI which is able to evolve in order to solve problems or to reach a specific destination. Evolution and learning are both ways in which an AI is able to alter its decision making process over time based on previous experience and/or other exterior factors meaning research into both learning, through neural networks (using the back propagation algorithm) and evolution, through genetic algorithms and genetic programming are sections of research required for the project.

### 2.2.1 Artificial Neural Networks

An artificial neural network is an attempt to simulate human thought by modelling the natural elements of the brain into the world of computing (Rojas, 1996). Since neural networks are modelled on a natural network of neurons; it would seem apt to look into the biological neurons before moving onto how they are modelled artificially.

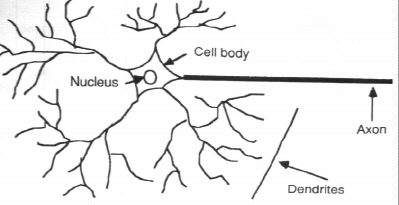


Figure 1.2.1 - 1 *Components of a Neuron.*

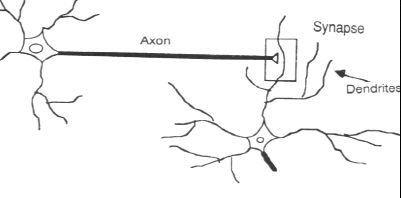


Figure 2.2.1 - 2 *The Synapse.*

These diagrams show what is believed to be the structure of the human brain. There is still however a great deal unknown about the operation of the brain and how we make decisions. There are theories however and they have been used to form the neural networks we have seen within computing today. The theories state that the human brain is uses electrical signals which are collected by neurons through many structures called dendrites. The neuron will send the signals collected across a long, thin stand called an axon. The axon is split into thousands of branches, at the end of each there is a structure called a synapse. The synapse is used to convert the signals in the axon to inhibit or excite the activity in the surrounding synapses and neurons. Over time the large number of components within the human brain is able to learn by altering the influence each neuron has on another using the process explained above.

The biological neuron system has been simplified for the use within the computing environment. The most basic element of the artificial neural network is the model of a neuron; these are often referred to as nodes.



Figure 2.2.1 - 3 *Artificial Neuron.*

An artificial neuron receives inputs from a number of connections (simulating the dendrites). Each of the connections has a specific weight that is assigned to it. The weights of each of the connections can be modified (simulating the synapse). All of the inputs are used within an activation function in order to set the values of the output connections (which are able to act as inputs for other nodes). In an Artificial Neural Network there are many of these nodes which work together to model the entire biological network.

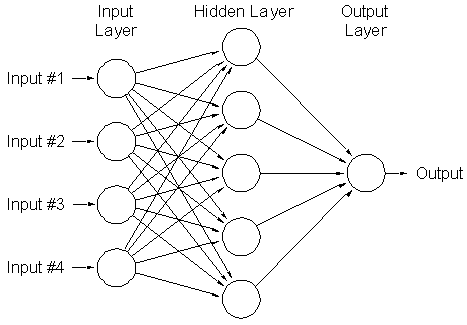


Figure 2.2.1 - 4 *Artificial Neural Network*

Most artificial neural networks use a multi-layered model when clustering the nodes; most systems use a model close to the model shown in figure 6. The main three layers are the input, hidden and output layers. The input layer is used to get input from the game world, this could be anything from what the AI agent is able to see, to some information the agents have on the player position. The designer of the AI within each individual application will decide what the inputs are in order to get the best results based on the specific system requirements. The hidden layer is used to simply apply some function to the inputs connected before passing the output to the output layer.

Artificial neural networks must be trained before they are able to solve problems or make decisions. The training process of an Artificial Neural Network begins by initialising the weights of all of the connections within the network with random weights. The network would then be introduced to a group of specific scenarios. The tests would most likely show the AI responding to situations in a way that would not appear intelligent since at this point the network would not be trained. Once the tests were complete; an algorithm would be used to alter the weights of the connections throughout the network based on the results. The whole training procedure would then consist of re-testing the network and altering the weights over and over until the network performed in a way that would meet the requirements of the application. The two methods described below are algorithms frequently used to train artificial neural networks.

### 2.2.2 Back Propagation Algorithm

The back propagation algorithm is a training algorithm used as a method of learning for neural networks (Rojas, 1996). The algorithm attempts to minimise the error the Neural Network is making by altering specific weights in order to train the network through multiple iterations. An error signal is calculated for each of the nodes and propagated back through the network updating the weights of the node connections based on a learning rate value. This function is mathematically complex and so the pseudo code is helpful in order to fully understand how the network is trained using the algorithm.

*Input: Problem Size, Input Patterns, Maximum Iterations, Learn Rate.*

*Output: Trained Network*

*Network <- Construct Network Layers*

*Network Weights <- Initialise Weights (uses Network and problem size)*

*Loop i to Maximum Iterations*

*Pattern(i) <- Select Input Pattern (uses Input Patterns)*

*Output(i) <- Forward Propagate (uses Pattern(i) and Network)*

*Backward Propagate Error (uses Pattern(i), Output(i) and Network)*

*Update Weights (uses Pattern(i), Output(i), Network and Learn Rate)*

*End Loop*

*Return trained network*

*Pseudo code found in* (Brownlee, 2011)*.*

### 2.2.3 Genetic Algorithms

Genetic algorithms are techniques written in an attempt to imitate the process of evolution within computing (Rabin, 2002). These algorithms perform selection from a genetic population pool, perform randomised (or mutated) crossover with the selected pairs in order to either produce entirely new population pools or to slowly evolve the current population pool.

There are several sections a genetic algorithm takes in order to ‘evolve’ the current population pool. The process begins by evaluating the fitness of the current population pool, this means testing how close the current population is to a potential solution. The next step is generating the new population; this requires selecting an amount of ‘parents’ from the current population in order to create offspring using the crossover process. The crossover process is used to create children from the parents that have been selected. How the crossover function works is entirely dependent on the data structure used to store the data for the population; for example, in a system where the population is stored as binary strings the method shown below is how crossover might work.

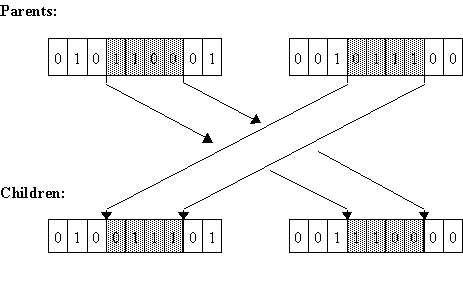


Figure 2.2.2 – 1 *Two point crossover of a binary string*

The method shown in figure 2.2.2 – 1 is called two point crossover; this is when two elements of a binary string are selected (most likely at random) and all of the elements within the two chosen are swapped between two parent strings in order to form two new child strings. Once crossover is complete; mutation takes place, this is used to stop the algorithm getting stuck in local optima by introducing new elements to the population. A new population is then created using either an entire set of children or a particular number of children replace a number of the current population. The whole process is then repeated until the goal state, or a specific set of conditions are met, the pseudo code below shows how the algorithm works.

1. *[Start] Generate random population of n chromosomes (suitable solutions for the problem)*
2. *[Fitness] Evaluate the fitness f(x) of each chromosome x in the population*
3. *[New Population] Create a new population by repeating the following steps until the new population is complete.*
   1. *[Selection] Select two parent chromosomes from a population according to their fitness (the better fitness, the bigger chance to get selected)*
   2. *[Crossover] With a crossover probability cross over the parents to form a new offspring (children). If no crossover was performed, offspring is an exact copy of parents.*
   3. *[Mutation] With a mutation probability mutate new offspring at each locus (position in chromosome).*
   4. *[Accepting] Place new offspring in a new population.*
4. *[Replace] Use new generated population for a further run of algorithm.*
5. *[Test] If the end condition is satisfied, stop, and return the best solution in current population.*
6. *[Loop] Go to step 2.* (Obitko, 1998)*.*

## 2.3 Rendering API

Games today generally use one of the main two graphics Application Programming Interfaces (APIs) in order to render the environments, characters and effects. These two APIs are called OpenGL and DirectX. The features and limitations of each are discussed below.

### 2.3.1 OpenGL

The main reference point for any *OpenGL* based report or discussion is the *OpenGL Superbible* book by Richard S. Wright *et al.* (Richard S. Wright, 2010)*.* It contains a guide to every element of *OpenGL* throughout its five editions giving the reader an overview of everything *OpenGL* has to offer. OpenGL offers a massive number of features including a separate language called GLSL (*OpenGL Shading Language*) for creation of vertex and fragment shaders (used to generate advanced modern graphics effects). *OpenGL* was written to be multi-platform meaning it can be used on Windows operating systems, Linux, MacOS and mobile operating systems. It also allows for extensions to be written on top of the basic functionality meaning an infinite amount of custom features can be added. The API is said to be clear to work with showing great ease of use due to its relaxed nature.

### 2.3.2 DirectX

Microsoft’s answer to a high performance modern graphics API is *DirectX*, or more specifically *Direct3D* (Walsh, 2008). DirectX is a collection of APIs used to manage all the elements of multimedia used today. *DirectX* keeps up with *OpenGL* in terms of features with the inclusion of shaders written in HLSL (*High Level Shading Language*) which was included first in *DirectX 8* with shader model 1.0. Through careful research it is easy to find that generally *DirectX* has been leading the charge in terms of new features, however *OpenGL* have since added all of the features making both of the APIs as good as each other in terms of the number of features. The main difference here is that *OpenGL* supports extensions whereas *DirectX* doesn’t. Both of the APIs are very fast at producing 3D graphics and have their good and bad points meaning they equal out overall.

## 2.4 Additional Research

This section of research was undertaken after the initial design and prototyping stages of the project, since through the process of completing these stages the topics below became relevant. The first of these topics was genetic programming, the initial design included a neural network but it became obvious that the collection of the specific data required in order to train the network would be difficult. Genetic Programming was the next direction for the project and so is discussed below.

### 2.4.1 Genetic Programming

Genetic Programming is very similar to the genetic algorithm explained above in that it is based around the idea of Darwinian natural selection (Koza, 1992). There is an initial population composed of individuals, the process of selection dictates which of the population are chosen for crossover and/or mutation and the new population is generated. The main difference with genetic programming compared to the genetic algorithm explained above is that the individuals which make up the population are programs bred to solve a particular problem.

The precise process of genetic programming starts with the generation of the initial population, the population is randomly generated (sometimes using several random number seeds) while ensuring that each individual of the population is unique (is not identical to another program). The population is (as stated above) comprised of individuals, which are programs, these programs are made up of a combination of elements from a terminal set and a function set. The terminal set could be seen as a group of variables available to the program, such as what the position is (x and y coordinates). The function set is the group of functions which the programs may execute. The terminal and function sets are very important in how successful the programs are since they are what the programs are made up of. The sets must contain functions and terminals which allow the program to be solved; otherwise the programs created will never solve the problem.

Once the initial population has been generated; each of the individuals are tested using a problem specific fitness function. A fitness function is entirely based on the problem at hand, for example, if the problem is to find a path through a group of obstacles; how many ‘moves’ the program takes before reaching the destination point would be a good way to measure the fitness of the program. Now that all of the individuals have fitness measures; the individuals are selected for crossover based on their fitness values, the better the programs are, and the more likely they are to be selected.

Crossover is the process of genetically breeding children from two parent individuals in order to create a new generation of individuals likely to be have an overall better fitness measure than the previous generation. The individuals in genetic programming are collections of terminals and functions from their respective sets, combined into a single program called a symbolic expression (or S-Expression). These symbolic expressions are normally viewed as LISP expressions or parse trees, as seen in figure 2.2.1 – 1.

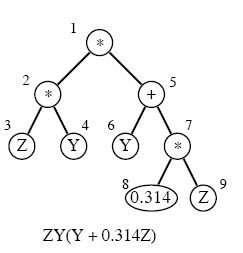


Figure 2.2.1 – 1 *Computer program symbolic expression*

When performing crossover, two parent programs are needed in order to create two child programs. The process of crossover randomly selects a sub-tree from each of the parents and swaps them to create the child programs which will be placed into the new generation.

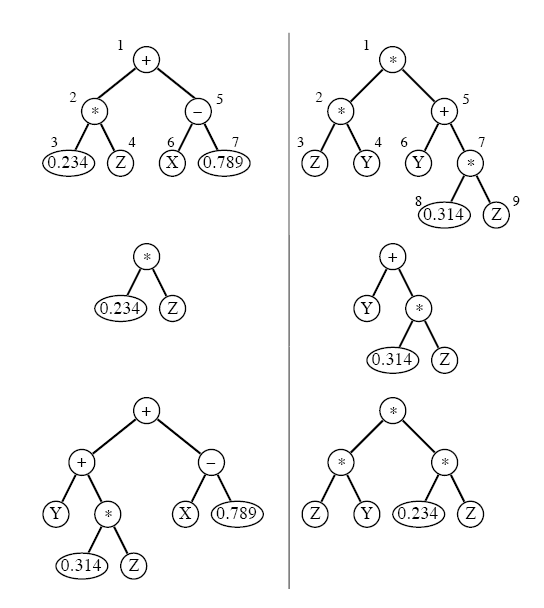


Figure 2.2.1 – 2 *The crossover process*

Figure 2.2.1 – 2 shows the process of crossover. The first part is the two programs selected for crossover, the second is the sub-trees selected for crossover and the final part is the final programs after the completion of the crossover procedure. Using the crossover method, and occasionally using mutation, the new generation is created. This process is completed several times until a set number of generations have been created or a specific end condition is met.

The highest level of the GP algorithm can be formalised as below:

*Let M = the population size;*

*Let G = the maximum number of generations to run;*

*Create M random programs for generation 1;*

*For gen = 1 to G do:*

*Evaluate the fitness for all the individuals;*

*If one solves the problem adequately, then*

*Terminate;*

*Construct the next generation of individuals;*

*Endfor;*

*Report the best individual found.* (Gritz, 1993)*.*

# 3 Design and Development

This section of the report contains the process of designing the implementation of the system. Contained within the sections below is the design for the entire system. Each section only details the system design at a high level using hierarchical design methods such as UML (Unified Mark-up Language) and Mind Map diagrams.

## 3.1 Design Considerations

The design of the project began by mind-mapping each of the elements of the project; these mind-maps would act as a collection of thoughts that could be drawn from when researching areas that may have been missed. They also worked well to make sure nothing was missed and that every area of the project had been given some thought. The mind-maps began at a high level detailing the elements of the project before being broken down further in every aspect in order to cover as much as possible.

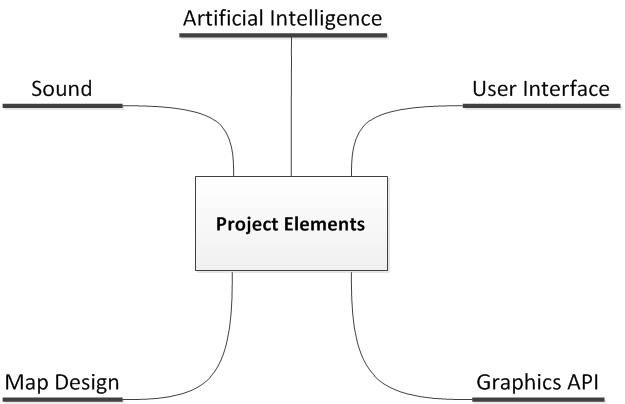


Figure 3.1 – 1 *The highest level ‘project elements’ mind-map.*

As you can see in figure 3.2 – 1; each of the elements of the system have been outlined, even the elements that are not present within the design at all, such as sound. The next group of mind-maps are breakdowns of each of the project areas found on figure 2.2 – 1.

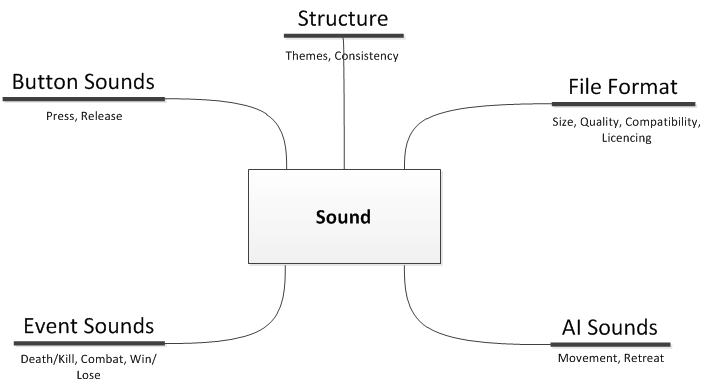


Figure 3.1 – 2 *Sound mind-map containing the areas of sound considered.*

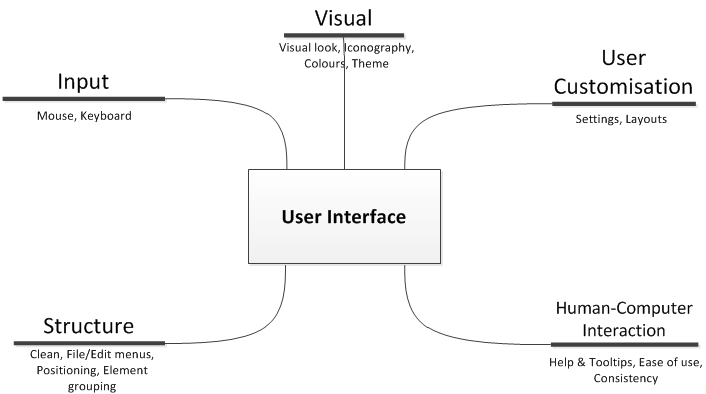


Figure 3.1 – 3 *User Interface mind-map consisting of the different areas of the User Interface considered.*

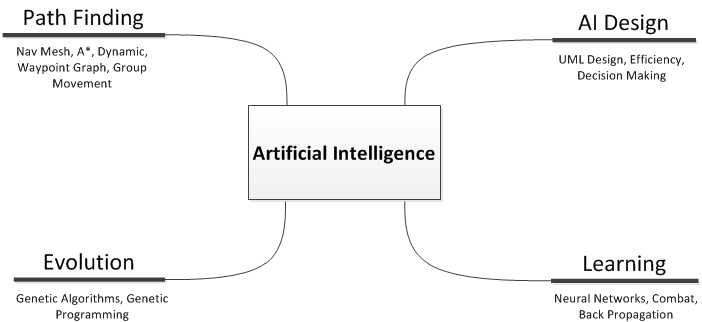


Figure 3.1 – 4 *Artificial Intelligence mind-map containing all the elements of AI considered.*

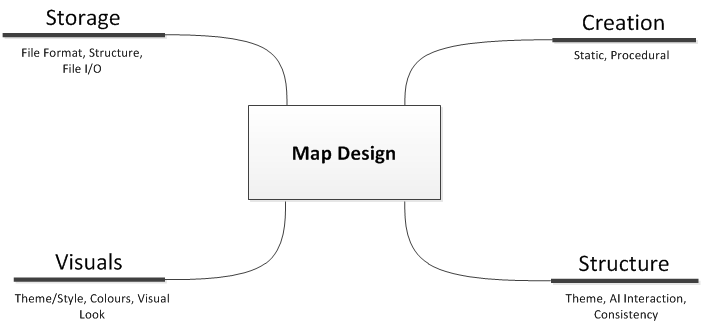


Figure 3.1 – 5 *Map Design mind-map consisting of all the areas of map design considered.*

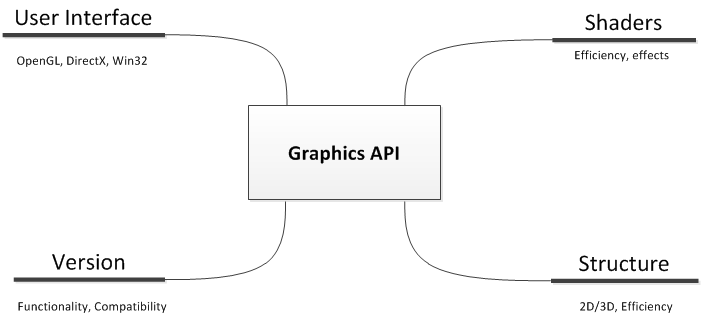


Figure 3.1 – 6 *Graphics API mind-map containing the considerations when choosing which API to use.*

The five figures above contain each element of the project broken down to show the areas which were considered while the system was being designed. Once all of the considerations were mapped out; each of specific elements was considered for inclusion into the system. All of the elements which were needed were added to the system design while the others were taken out; a specific example of this is sound.

## 3.2 Design Overview

The designs below show the class design in a number of different diagrams, each of which shows how it has been refined. A hierarchical approach has been taken meaning that the top level UML (unified mark-up language) diagram contains a basic overview of the system.

### 3.2.1 System Design

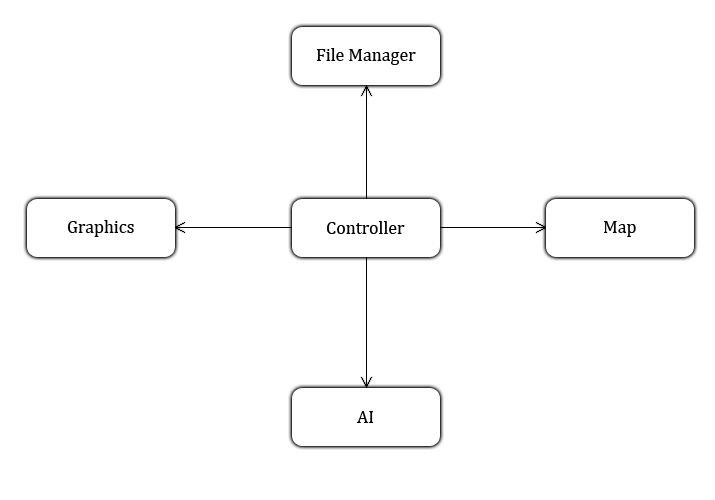


Figure 3.3 – 1 *High level system design.*

Figure 3.3 – 1 is a basic high level design for the system showing each of the elements in the most abstract form. The controller is the main section of the program which controls the applications main functionality. The graphics section will include the creation of the win32 window along with the initialisation of OpenGL. The map contains the map data, such as path finding node data and the visual layout of the map. The file manager contains all of the classes used for managing the resources for the application; this may include anything from textures to full meshes. The AI section will be fully explained within the next section since it is a larger and the most complex part of the system design.

### 3.2.2 AI Design

Designing the AI was one of the most difficult sections of the system since it required that a great deal of research into the areas of Genetic Programming and how the class would be used alongside the AI classes to allow the evolution of behaviours.

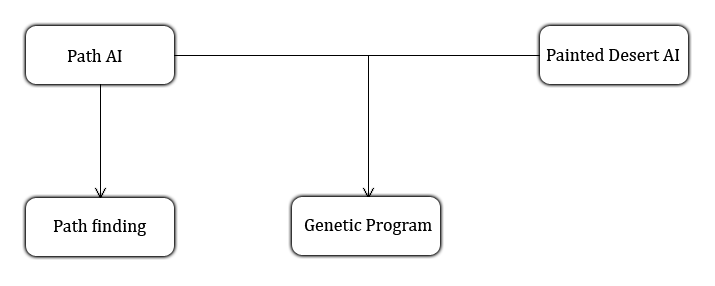


Figure 3.2.2 – 1 *High level AI design*

Figure 2.2.2 – 1 shows the interaction between the AI classes and the genetic programming library. Each of the AI classes has their own genetic program to control the process of generating and evolving programs. The Path AI also interacts with a path finding class, this is A\* path finding since the program generated by the genetic program is compared to A\* to show how successful it was.

## 3.3 Detailed Design

This section contains the design of the system at a much lower level. Explaining how the classes interact with each other to complete the aim of the project. Starting with the design of the system itself and how it works. The section after details the design of the AI and how they are integrated with the genetic programming library.

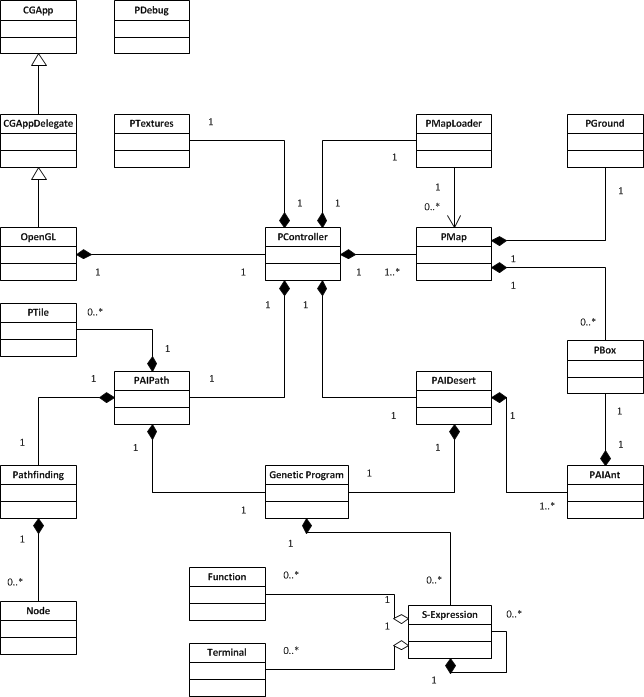


Figure 3.3 – 1 *Full System UML*

Figure 3.3 – 1 shows the entire system design in the form of a UML diagram. The following sections split this diagram into sections and explain what each part of the system does and how it works.

### 3.3.1 OpenGL Framework

The *OpenGL* framework was written by Dr. Paul Angel. He provided the code for use in a Real Time Rendering module but permission has also been granted to use the code within the project. It controls the basic setup of the window and initialisation of *OpenGL*. The code also provides a group of useful data types under the name of “*CoreStructures*”, these are a very useful group of data types that have been used when setting up vertices, texture coordinates and vertex buffer objects for rendering.

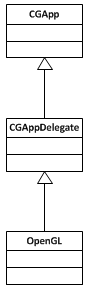


Figure 3.3.1 – 1 *OpenGL Framework Design*

Within the *OpenGL* framework, there is a great deal of inheritance used to allow for reusability and platform portability. There is a “*CGApp*” class which works as a wrapper class for the creation and running of the application window. A class named “*CGAppDelegate*” inherits from this in order to add functionality, however this is an abstract class meaning another class must inherit from it in order for an instance (and in this case, an application) to be created. The class which creates the instance of the window also manages the message loop and initialisation of *OpenGL* is the “*OpenGL*” class.

### 3.3.2 System Design

The system includes a group of classes which manage the general running of the system. The controller class provides the functional centre for the project while the map provides a means of storing and displaying the three dimensional environment. There are also the two AI classes which individually manage a genetic program in order to generate and evolve programs. The group of resource management classes are used to load and store maps and textures into the system. Finally, the debug class controls the creation and setup of the debug console as well as the redirection of messages to the window.

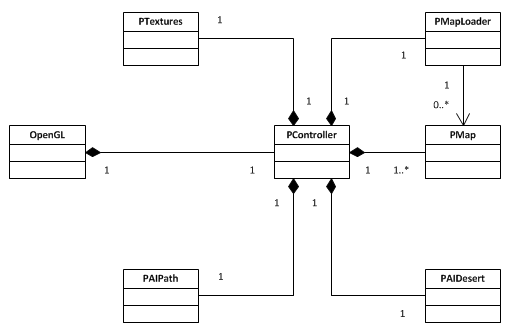


Figure 3.3.1 – 1 *Low level system design*

The *OpenGL* class (explained in the previous section) manages the messages sent to the window by the operating system (Windows 7 in this case) as well as managing a run loop sending update and render calls when required. The *OpenGL* class contains the only instance of the controller class for the project and hierarchically calls the update and render functions to keep the metaphorical ‘gears’ of the whole system turning.

The controller class contains all of the objects needed to allow the project to function. The map class is used to render the scene, the resource classes are used to load and store resources and the AI classes are used to control the AIs for their respective problems. Any calls to the update and render functions will subsequently call the update and render functions of all the required classes.

### 3.3.3 AI Design

There are two main AI classes; the AI Path class and the AI Desert class. The AI Path class is used to manage the creation of evolutionary path finding characteristics to find paths through a randomly generated map. The AI Desert class is used to manage the creation of evolutionary characteristics for a group of ants whose goal is to collect grains of sand (explained below).

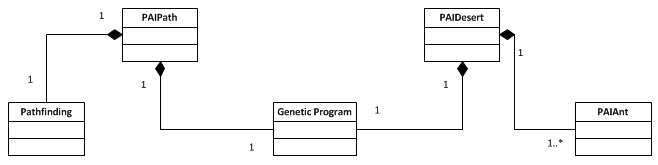


Figure 3.3.2 – 1 *AI Design UML*

The AI Path class first sets up the genetic program class by initialising each of the required aspects. The GP used within the project is a genetic programming library written by Larry Gritz in C++ (Gritz, 1993), he states (through the works of Koza) there are a group of essential elements required for any GP problem, and these are:

* Objective: A statement of the problem which we are attempting to solve.
* Terminal Set: The list of terminals which can be included in any symbolic expressions generated by the GP.
* Function Set: The list of functions which can be included in any symbolic expressions generated by the GP.
* Fitness cases: A problem dependant set of test cases which will be used by the fitness function to derive the raw fitness value for each individual.
* Raw fitness: The method by which the fitness function will calculate the raw fitness measure for each individual.
* Standardised fitness: An optional method of conversion between raw fitness and standardised fitness.
* Hits: An auxiliary measure of the number of fitness cases which achieve satisfactory fitness.
* Wrapper: An optional conversion between output values of an evolved symbolic expression and input values needed by the fitness function.
* Parameters: Any of the major or minor parameters which control the GP run.
* Success predicate: An optional condition which may terminate the GP run before the maximum number of iterations.

The AI Path class and the AI Desert class contain all of these elements based around their individual problems. Below is a listing of the elements for each of the AI classes.

AI Path elements:

* Objective: To produce a program that moves from the start point, to the destination point of the map.
* Terminal Set: X, Y.
* Function Set: Move North, Move South, Move East, Move West, IFLTZ, IFLTE.
* Fitness Cases: Difficult to produce test cases since the maps are randomly generated.
* Raw Fitness: The fitness is based on many factors. Firstly for each move the fitness will raise dependant on the distance to the destination, meaning the program will attempt to reach the goal as quickly as possible. The program will also be run multiple times, for each full run of the program; the fitness will increase based again on the distance to the destination. Finally, if the program doesn’t reach the destination; the fitness will be multiplied by a certain factor.
* Standardised fitness: Optional, not required in this case.
* Hits: The hit count depends on whether the program generated reaches the destination while its fitness is being tested.
* Wrapper: Optional, not required in this case.
* Parameters: For use of debugging, the verbose characteristic of the GP has been set to “DEBUG”. Each time the user presses the generation key (space), ten more generations are bred.
* Success predicate: Although optional, when running this project; we want to see results as quickly as possible and so the success predicate is when a program with a ‘hit’ has been found.

AI Desert elements:

* Objective: To produce a program which, when run in parallel for twenty ants, collects and relocates the grains of sand on the map to a single location.
* Terminal Set: X, Y, Colour, Carrying.
* Function Set: Move North, Move South, Move East, Move West, Move Random, Pick-Up, If-Drop, IFLTZ, IFLTE.
* Fitness Cases: Again, positions of sand are generated at random; test cases are difficult to produce.
* Raw Fitness: The fitness is based on where each grain of sand is located at the end of the running of the program. Since the program is run for each of the ants, the program is run for each of the starting positions of the ants before the fitness value is calculated.
* Standardised fitness: Optional, not required in this case.
* Hits: The hit count is dependent on whether each of the grains of sand is in the correct location (vertical band) once the program has been tested.
* Wrapper: Optional, not required in this case.
* Parameters: For use of debugging, the verbose characteristic of the GP has been set to “DEBUG”. Each time the user presses the generation key (space), ten more generations are bred.
* Success predicate: Optional, not required in this case.

The AI Desert class manages a GP used to solve the “Painted Desert” problem as explained by John R. Koza (Koza, 1992). The Painted Desert problem was proposed by Mitchel Resnick and Uri Wilenski of the MIT Media Laboratory. The version explained here is different to the problems proposed above.

The Painted Desert problem discussed here is composed of 20 ants and 20 grains of sand for each of the colours (black, dark grey and light grey). The 20 ants and 60 grains of sand are given random starting positions on a 20 by 20 grid as shown in figure 3.3.2 – 1.

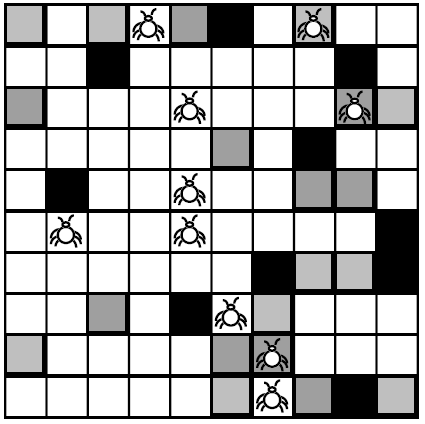


Figure 3.3.2 – 1 *Starting arrangements of the ants and grains of sand. Smaller than explained (10x10 instead of 20x20 for simplicity)*

As stated above, each of the ants executes a single program generated by the genetic program. The goal for the ants is to move all of the black grains of sand to the far left vertical band, the darker grey grains of sand to the next vertical band to the far left and finally all of the lighter grey grains of sand to the next vertical band as shown in figure 2.3.2 - 2.

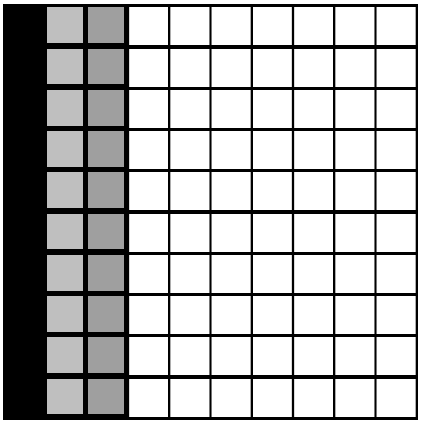


Figure 3.3.2 – 2 *Desired final arrangement of sand grains*

At first this sounds like a trivial problem, however the ants only have a local sense meaning they only hold data based on the immediate grid space where they are located. No more than one grain of sand can be at one position at one point in time.

### 3.3.4 Map

The map class is one of the more simple classes. It is simply designed to store all of the visual map data as well as the path finding node data. As the controller class updates and renders each of the elements within the project; it will call the update and render functions for the map displaying the map on the screen.

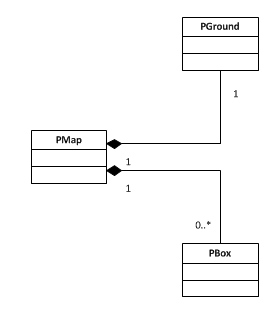


Figure 3.3.3 – 1 *Map UML Design*

The grid spaces on the map are either free or have a cube located at the position. Dependent on the map being used as well as the AI class selected within the project, a specific map is displayed.

### 3.3.5 Resources & Utilities

Resources and utilities, these are the resource loader classes and the debug class. The resource loaders are simple yet stop many issues with optimisation by holding the loaded data and only allowing anything to be loaded once.

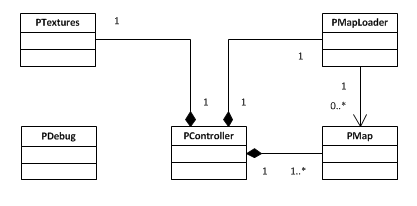


Figure 3.3.4 – 1 *Resource loader and Utility design*

The “*PTextures*” class simply allows the loading and storing of textures, each texture is accompanied by a string (the file path to the texture). This system is efficient since it allows any class to request a texture as many times as required and it simply returns the texture (as a *GLuint*) rather than loading the same texture multiple times.

## 3.4 Screen Designs

The screen design is an important section of the design since it is how the user will interact with the system. The application will display the scene at all times with little or no user interface since the project is based on demonstration and therefore will not require the use of UI elements.



Figure 3.5 – 1 *Influence map on grid based scene.*

The scene will be made up of a simple flat surface with different sized cube objects making up the layout of the map much like the scene in figure 3.5 – 1. This design was chosen since it is simple and easy to modify. This design also allows for the AI types to adequately display their outcomes.

## 3.5 Justifications

This section contains any justifications for choices made between several different algorithms or libraries.

### 3.5.1 Graphics API

The decisions made such as the use of the A\* algorithm for path finding or using genetic programming for program generation and evolution are obvious choices since both are the most popular algorithms/concepts used in their respective areas. A choice which needed to be made which was far less obvious was the choice between the leading graphics APIs, which should be used and why? The main contenders are *DirectX* and *OpenGL* (although *win32* must be used when creating applications under the Windows platform). The research undertaken into the features of both APIs show a balance since they both support all of the features necessary to render the demonstration with efficiency. The choice of which to use is almost entirely based on two simple factors; personal preference and previous experience, the graphics API which will be used within the project will therefore be *OpenGL*.

# 4 Implementation

This section of the report discusses the implementation of the system. Starting with how the implementation was planned, followed by a detailed description of each of the classes within the design explaining any changes made from the initial system design. Included in this section is also any libraries and algorithms used within the project implementation which were not specifically written for this purpose along with explanations of their use and functionality within the project.

## 4.1 Implementation Plan

The implementation was planned to be completed in a group of different sections. The first objective was to make sure the *OpenGL* framework was working correctly and the project was setup to allow for the addition of the project classes. Once the *OpenGL* framework was working correctly; all the core classes were fully implemented, these classes are the backbone of the project and were required before the AI classes and others could be added. The final step was to ensure the careful implementation of the AI classes.

## 4.2 Final Implementation

The project implementation is broken into sections where each is discussed and explained fully, including variables and methods, why they are needed and how they are used.

### 4.2.1 OpenGL Framework

The first section of the implementation to be discussed is the *OpenGL* framework. As stated in the design section, this was written by Dr. Paul Angel and so was mostly complete and fit for the purposes of the project.

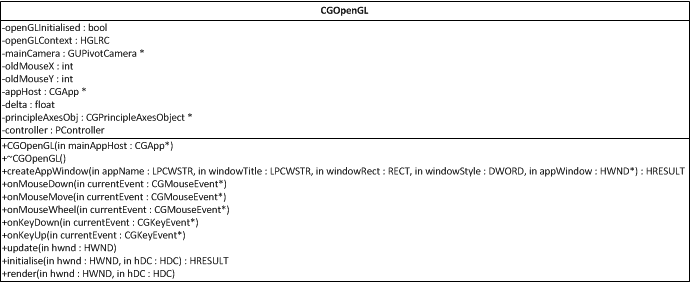


Figure 4.2.1 – 1 *CGOpenGL Class*

The “*CGOpenGL*” class shown in figure 4.2.1 – 1 is the main class that the project controller interacts with. There are many other classes within the framework including a singleton “*CGApp*” class, an abstract “*CGAppDelegate*” class and a large assortment of useful data storage classes to aid rendering under the name of “*CoreStructures*”. The basic running of the *CGOpenGL* class consists of setting up the application window, initialising *OpenGL* and managing the windows message loop. In essence, the entire project is run from this class, it contains an instance of the project controller class and calls update and render when needed.

There has however been a small group of changes to the class to allow the project to run correctly and have the required information. The main change is the addition of the controller object to run the project. Another important change is the addition of the “*appHost*” member variable. When the class is initialised, a pointer to the host application is sent in as a parameter to the constructor. This is simply taken and saved as a member variable. The reason for this change is that when the update function within the controller is called; a “*delta*” is needed in order to know the time between frames. This is used to calculate when objects should move or animate.

### 4.2.2 System

Once the *OpenGL* setup was modified to the requirements of the project; the system classes and main structure of the implementation were developed as the backbone of the project. This all began with the implementation of the controller class, named “*PController*”. All of the project classes were given the letter ‘P’ as a prefix to easily differentiate between project only classes and others.

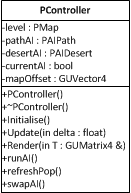


Figure 4.2.2 – 1 *PController Class*

The “*PController*” class is the main class for the project, without this class; the implementation wouldn’t work. The class contains the map, each of the AI types (Path and Desert) and the map offset (used to render the scene in the centre of 3D space). The update and render functions are used to hierarchically call the update and render functions of the map and different AI objects. *RunAI*, *RefeshPop* and *swapAI* are functions used for user input and are called when specific keys have been pressed. This class has undertaken a few alterations due to the time constraints of the project, mainly the removal of the “*PTextures*” and “*PMapLoader*” classes (found in the design in section 3.3.4). Descriptions of the original roles of these classes can be found in section 6.3.

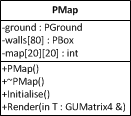


Figure 4.2.2 – 2 *PMap Class*

The “*PMap*” class shown in figure 4.2.2 – 2 has undertaken some significant changes from the initial design, again due to the time constraints of the project. The class now simply renders the ground, it was originally designed to manage the map objects such as the walls for the path finding problem and the grains of sand for the painted desert problem but the responsibility has since be moved over to the respective AI classes.

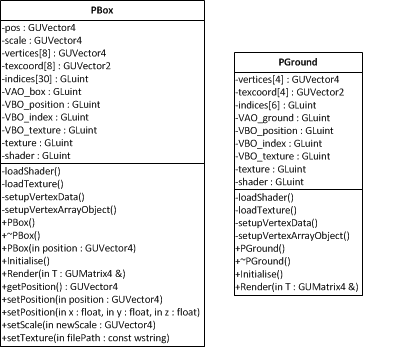


Figure 4.2.2 – 3 *PBox and PGround classes*

The “*PBox*” and “*PGround*” classes are basic objects used for rendering. The *PGround* class is essentially a simplified *PBox* which renders a two dimensional plane with a simple grid texture. Both classes contain a group of different features which aid the speed and efficiency of rendering; these include vertex buffer objects, vertex array objects and index arrays. These allow for the vertices, texture coordinates and other information to be moved onto the graphics card and rendered by the graphics processor which is optimised for graphics processing. Other notable functions are the loading of shaders and textures, these allow for easy texture and shader loading. This approach however is not the best since each instance of these classes create and load their own texture and shader, even if they use the same shader and/or texture. Given the time, additional classes would have been implemented to manage the loading of textures and shaders to prevent such problems existing.

### 4.2.3 Libgp

Libgp is the library used within the project to allow the generation and evolution of programs. It was written by Larry I. Gritz (Gritz, 1993). The library contains a small group of classes used to manage the generation and evolution processes of the programs. Since this is a large section of the project and part of the main aim, it was entirely re-written to gain a full understanding of how it works.

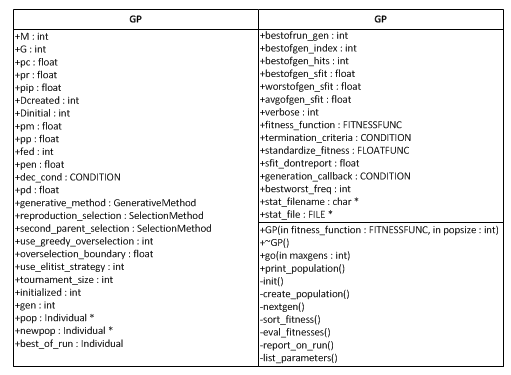


Figure 4.2.3 – 1 *GP Class (edited to save document space)*

Figure 4.2.3 – 1 shows the full GP class. This class is used alongside the AI classes and controls each step of the genetic programming process. The first function to note is the “*go*” function, once all of the initial variables have been setup, the function set and terminal set defined and the fitness function written and attached; this function is called to start the generation and evolution of the programs. The other functions are fairly self-explanatory, creation of the population, the crossover process and mutation within the next generation function, sorting and evaluation of the population (the process of genetic programming is explained in section 2.4.1).

The class attributes are much more difficult to understand based on their simplistic variable names and so below is a listing of all the variables along with descriptions of what they are used for:

* M: The population size.
* G: The number of generations to run.
* Pc: Probability of crossover.
* Pr: Probability of reproduction.
* Pip: Probability of internal crossover.
* Dcreated: Max depth of S-Expressions.
* Dinitial: Max depth of initial population.
* Pm: Probability of mutation.
* Pp: Probability of permutation.
* Fed: Frequency of encapsulation.
* Dec\_cond: Condition for decimation.
* Pd: Decimation percentage.

Each of the attributes above changes the characteristics of the evolutionary process in some way. These variables each have default values, many of which are unaltered within the project since they are either acceptable values or are not required. In the path finding problem; there is no mutation, permutation or encapsulation so each of those default to the value of “*0.0*” meaning they will never occur. Although this could create a problem with the programs generated in terms of the breeding out of functions or terminals which will move a program out of a local optima scenario; there are other means in place to stop this from happening ruling them out completely. As an example, the user is able to generate a completely new population whenever they deem it necessary; this could be at a point when the programs bred are trapped in a local optima scenario and would reintroduce terminals and functions which may have bred out of the population.

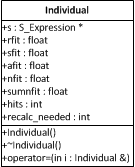


Figure 4.2.3 – 2 *Individual Class*

The class shown in figure 4.2.3 – 2 is the “*Individual*” class which makes up the population within the GP class described above. Each Individual contains an S-Expression which in turn can contain any number of S-Expressions (limited by the max depth setting). There are a small group of attributes which contain information for the raw fitness, standardised fitness, average fitness and normalised fitness. These attributes are all used by the GP to decide whether they should be used as a parent to perform crossover or simply to discard the individual.

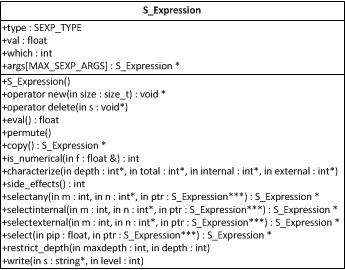


Figure 4.2.3 – 3 *S-Expression Class*

The “*S-Expression*” class shown in figure 4.2.3 – 3 is the representation of the actual programs generated by the genetic program. This class manages anything specific to the modification, creation and output of S-Expressions. There are a small group of attributes which contain information for the S-Expression, they are listed below:

* Type: Terminal or Function (from the respective set).
* Val: Value (if the S-Expression is a terminal).
* Which: An index to the function or terminal.
* Args: Arguments, a group of S-Expressions which could be terminals or functions which are passed to the S-Expression on execution (if the S-Expression is a function).

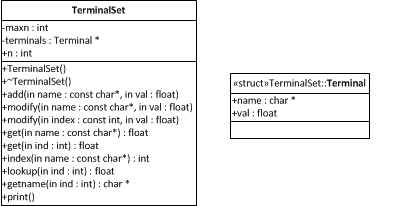


Figure 4.2.3 – 4 *TerminalSet Class*

The “*TerminalSet*” class shown in figure 4.2.3 – 4 contains the entire set of terminals that are used within the S-Expressions; the terminals themselves are stored simply using a name and a value which can be modified during the fitness testing of the programs. The terminal set class contains many functions allowing for the addition of new terminals, modification of existing terminals (through the means of the name or the index of the terminal), querying of existing terminal values and the printing of the entire terminal set.

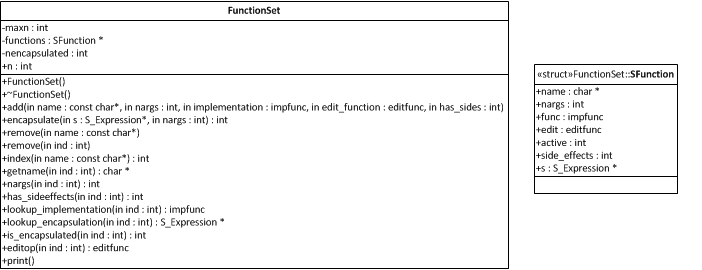


Figure 4.2.3 – 5 *FunctionSet Class*

The “FunctionSet” class is a little more complex since functions have to be stored with the ability to be executed when needed. The “SFunction” class is more complex than a simple character string to store the name and a float to store the current value. The SFunction class stores a group of extra attributes, each of which is listed and described below:

* Name: Stores the name of the function.
* Nargs: The number of arguments the function takes as parameters.
* Func: A pointer to the function itself.
* Edit: A pointer to the edit function (not used within the project).
* Active: Stores whether the function is currently available to be used within individuals in the population.
* Side\_effects: Does the function alter any of the terminal values?
* S: An S-Expression which contains this function.

### 4.2.4 A\*

A\* is the path finding algorithm used to compare to the genetic program generated by the GP (explained above). Since there is already a group of implementations of A\* path finding available, the algorithm used was written by YouTube user iGunSlingeRv2 within a four part video (iGunSlingeRv2, 2011).

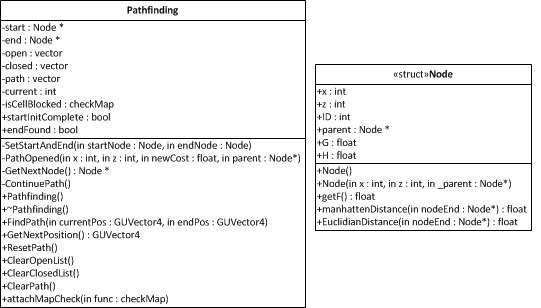


Figure 4.2.4 – 1 *Pathfinding Class*

The “*Pathfinding*” class shown in figure 4.2.4 – 1 is an implementation of the A\* path finding algorithm (explained in section 2.1.1). Most of the code is unchanged from the original author, however many of the variable and class names have been changed to increase the readability of the code. The “*EuclidianDistance*” function was added to the node structure to calculate the heuristic cost of each node more precisely, the “*GetNextPosition*” function was altered return the next position without erasing previous positions within the calculated path. The most interesting change however is the addition of the “*isCellBlocked*” function pointer; this is called to check the map for moveable positions and walls. During the setup of the path finding class, the new function “*attachMapCheck*” is called and passed a pointer to the map checking function.

### 4.2.5 Evolving Path finding

One of the key elements of the project is the evolution of the path finding programs generated by the GP. This section explains how the GP is setup to generate path finding programs including a description of the function and terminal sets, the fitness function and finally the termination criteria function.

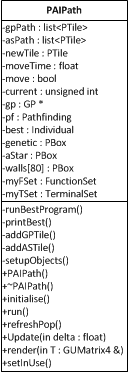


Figure 4.2.5 – 1 *PAIPath Class*

The “*PAIPath*” class shown in figure 4.2.5 – 1 was written for the project to manage the GP to generate and evolve programs that could find a path through a randomly generated map. There are a group of elements which must be defined and setup in order for the GP to successfully create programs, below is a listing of each (originally defined in section 3.3.2) with detailed descriptions of how they have been implemented:

Terminal Set

* X, Y.

These two simple float variables allow the GP to know where the AI agent is located on the map at any point during the running of the fitness function.

Function Set

* Movement functions (Move North, Move South, Move East, Move West),
* IFLTZ, IFLTE.

The movement functions simply check the current position of the agent before deciding whether the move is possible or not. If the move is not possible, the agent will not move and will return ‘-1’. If the move is possible; it will be taken and the function will return ‘1’. The IFLTZ and IFLTE functions are used a great deal within the area of genetic programming (Koza, 1992) since they allow the genetic program to make decisions based on the information returned from functions and/or the values of the terminals within the GP.

The IFLTZ or If-Less-Than-Zero function takes three arguments (or parameters), the first is evaluated, if it is less than zero the second argument is evaluated else the third argument is evaluated. In terms of code, it is as shown in figure 3.2.4 – 2.

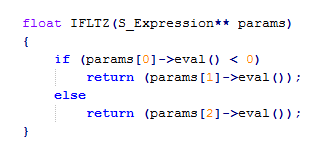


Figure 4.2.5 – 2 *IFLTZ Implementation*

The IFLTE or If-Less-Than-Or-Equal function takes four arguments, then first and second are evaluated, if the first argument is less than or equal to the second, the third argument is evaluated else the forth argument is evaluated. The code for the function is shown in figure 4.2.4 – 3.

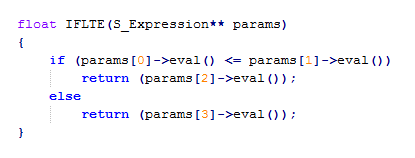


Figure 4.2.5 – 3 *IFLTE Implementation*

Raw Fitness

The fitness function needs to be able to differentiate the good solutions from the bad ones. The most obvious method of doing this is by checking that once the program has been evaluated; check if the X and Y terminals are the same as the destination location. This will check whether the program generated is a solution and therefore if it has scored a ‘hit’. The fitness function must also push the GP into creating solutions that are better; this means that a scoring system must be in place that scores the programs that reach the destination quickly better (lower in terms of fitness) than those that either reach the destination slowly, or do not reach the destination at all. The scoring system chosen for the project is that for each move made by the agent, the current X and Y terminal values are added to the fitness. On top of that, the programs that do not reach the destination get their fitness score doubled to ensure a high fitness value for the programs that do not solve the problem. When testing the fitness function, another problem arose, if programs did not move from the starting position; they would score a low fitness score. To solve this, after each run of the program (the program is evaluated fifty times) the current position is added to the fitness score. This alteration solves two problems, the issue it was designed to solve and it will score the programs that reach the destination in a single evaluation lower than those who reach the destination in multiple runs.

Hits

A ‘hit’ is scored if the program reaches the end destination while the fitness is being tested (explained above).

Success predicate

By default there is no success predicate (termination criteria) and so the evolution of the programs will run until the number of generations has been reached. We would like the GP to find a solution as soon as possible and so a simple termination criteria of a program with a single hit being found is enough to speed up the process of starting the animation of a working program. If a program has reached the destination within the fitness function, it is scored a hit, if a program is found within the GPs population with a single hit or more; the evolution stops and the best individual is taken from the GP to be displayed.

### 4.2.6 Evolving Group movement

Another important element of the project is the evolution of a program that when evaluated by twenty ‘ants’ with random starting positions will display emergent group movement. This section explains how the GP is setup to generate programs to collect and move grains of sand to a specified location, including a description of the function and terminal sets and the fitness function.

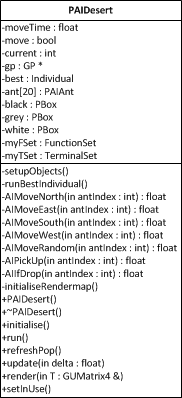


Figure 4.2.6 – 1 *PAIDesert Class*

The “*PAIDesert*” class shown in figure 4.2.6 – 1 was written for the project to manage the GP to generate and evolve programs that could collect randomly generated grains of sand and place them at the left most part of the map (as explained in section 2.3.2). There are a group of elements which must be defined and setup in order for the GP to successfully create programs, below is a listing of each with detailed descriptions of how they have been implemented:

Terminal Set

* X, Y,
* COLOUR,
* CARRYING.

The X and Y values are float variables used to store the position of the current ‘ant’ being tested by the GP. The COLOUR terminal is the colour of any grain of sand on the ant’s immediate grid location, ‘-1’ if there is not a grain of sand, ‘0’ for black, ‘1’ for the darker grey and ‘2’ for the lighter grey. CARRYING is the colour of the grain of sand the ant is currently carrying, again ‘-1’ if the ant is not carrying any sand, ‘0’ for black, ‘1’ for the darker grey and ‘2’ for the lighter grey.

Function Set

* Movement functions (Move North, Move South, Move East, Move West, Move Random),
* If-Drop, Pick-Up,
* IFLTZ, IFLTE.

The movement functions simply check the current position of the agent before deciding whether the move is possible or not. The movement functions all return the colour of the sand on the tile they move to (-1, 0, 1 or 2). The difference with these movement functions to the movement functions defined for the path finding function set is that if the ant is next to the edge of the map; they are able to wrap around and move to the opposite side of the map. The Move-Random function simply chooses one of the other movement functions at random and evaluates them. The IFLTZ and IFLTE are as explained in the previous section.

Raw Fitness

The fitness function needs to be able to differentiate the good solutions from the bad ones. The main method used to make this possible was to score each of the programs based on where the grains of sand are after the program has been run (three hundred) times in this case. There was a problem here however; there are twenty ants, and running a single program would mean the problem would become almost impossible to solve. A solution was proposed by John R. Koza (Koza, 1992), this was to allow previous programs to affect the sand grain positions and therefore fitness scores of the programs. This appeared to solve the problem, however, within the project an output was required and a single program was required to solve the problem. The solution used within the project was to run the program generated by the GP for each of the starting positions of the ants, allowing a single program to be scored based on positions of the grains of sand after the program had been run.

The precise fitness is measured by checking the entire map space for black, light grey and dark grey grains of sand and adding to the fitness based on how far each grain is from the vertical band they should be positioned within based on the problem definition.

Hits

A ‘hit’ is scored if all of the grains of sand are located within the correct vertical band defined within the problem definition.

Success predicate

The termination criteria function is similar to the path finding problem termination criteria function explained in section 3.2.5. The function is in place to speed up the generation process to allow a graphical representation to be shown to user in as little time as possible.

### 4.2.7 Resources & Utilities

The final classes which were left until the end of the implementation stage of the project were the resource management and utility classes. In previous sections it was explained that the resource classes have been left out due to time constraints, however, a debug window was included in the final implementation.



Figure 4.2.7 – 1 *PDebug Class*

The “*PDebug*” class shown in figure 4.2.7 – 1 was one of the final classes to be implemented. It was a great deal simpler than first expected and allows for a simple debug window to be displayed along with the redirection of the standard output to the window instead of another output file.

## 4.3 Run-Time Walkthrough

This section was added in order to give the reader an idea of how the completed program works in terms of flow. The program can be split into three sections; setup, initialisation and the run loop. The three sections are described and explained in full below.

Setup

The program begins with the setup of the main application window and the debug window. The standard output is rerouted to the debug window and the “*CGOpenGL*” class is responsible for the creation of the main application window. Once the main setup of both windows is complete; the message loop begins and the windows are displayed. The setup stage of the program is relatively short since the windows are quickly setup leading into the initialisation stage.

Initialisation

The initialisation stage of the program begins with the initialisation of *OpenGL*, which is completed by the *CGOpenGL* class. The controller, map, both AI instances and the map objects are all initialised ready for update and rendering. Within the classes used for rendering such as the ground and box classes; the vertex array and vertex buffer objects are setup and initialised. This stage is again, very short as there is not a great deal to explain or discuss. Once this stage is complete, the main run loop of the program begins.

Run Loop

The program has fully setup the windows, *OpenGL*, the render objects and AI. The main application window now displays the initial scene, the path finding AI scene. During the initialisation section; the map was randomly generated and the GP was setup leaving everything in place for display. The program at this point will simply wait for the user to start the evolution of the program using the space key.



Figure 4.3 – 1 *Initial view of path finding scene*

Figure 4.3 – 1 shows how the display and debug windows look once the setup and initialise stages have been completed. At this point, the user is able to move the camera around the scene, start the evolution of the GP, change displayed scene and force a re-generation of a random population.

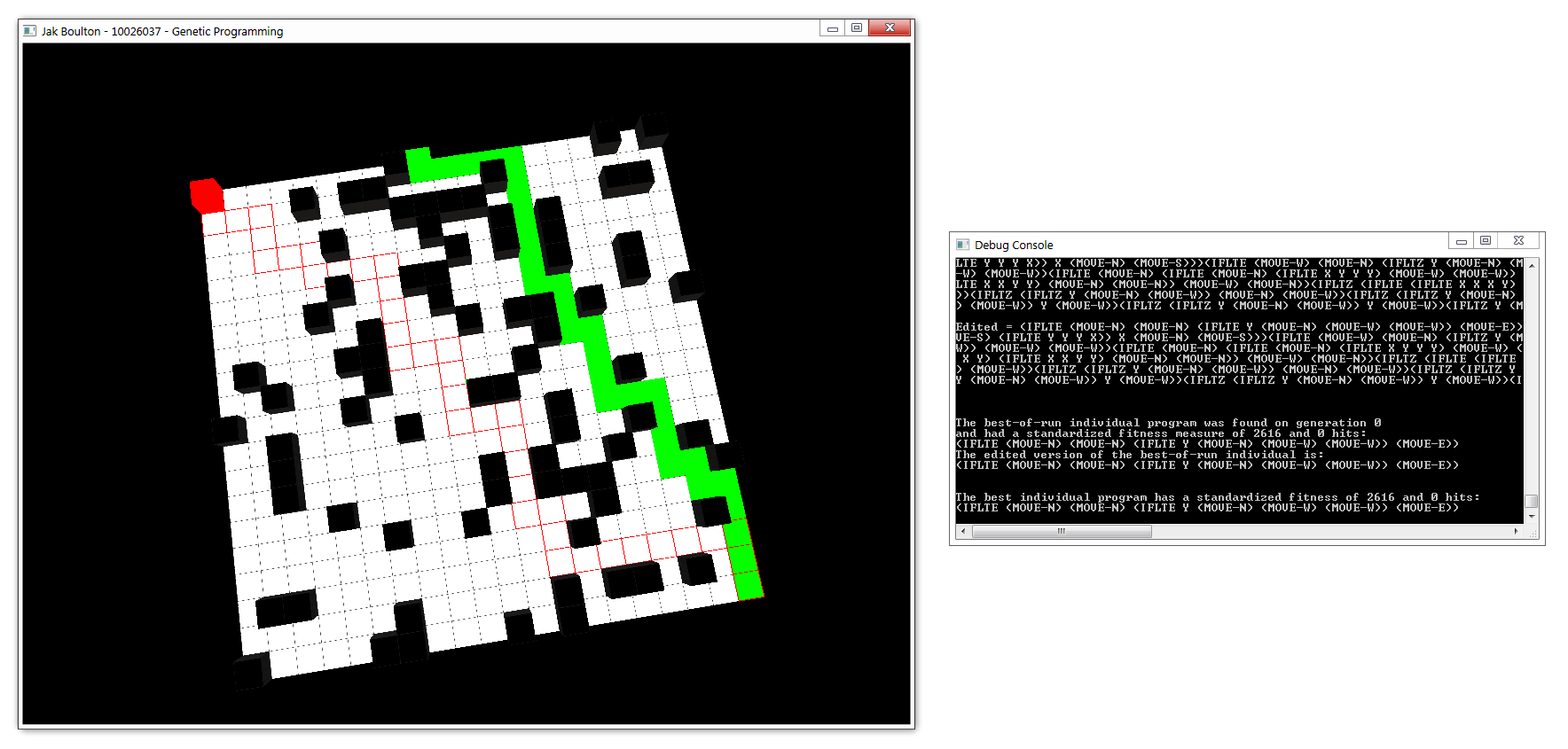


Figure 4.3 – 2 *View of scene when the GP controlled path finding has failed to reach the destination point*

Figure 4.3 – 2 shows how the main application display and debug windows look once the user has started the evolution and the GP has failed to generate a program which solves the problem. The red cube and red highlighted tiles show the A\* path to the destination while the green tiles and green cube show the path generated by the GP. The debug window shows all of the output produced by the GP consisting of the initial population, calculated fitness values for each and the S-Expression which is the current best of the run. The best individual is then run again and the path that it takes is recorded in order to be displayed in steps.



Figure 4.3 – 3 *View of scene when the GP controlled path finding has evolved to complete the goal of reaching the destination*

Figure 4.3 – 3 shows the further evolution of the programs leading to the success of a program. The GP generated program also managed to reach the destination point as fast as the A\* algorithm while having a largely different path.

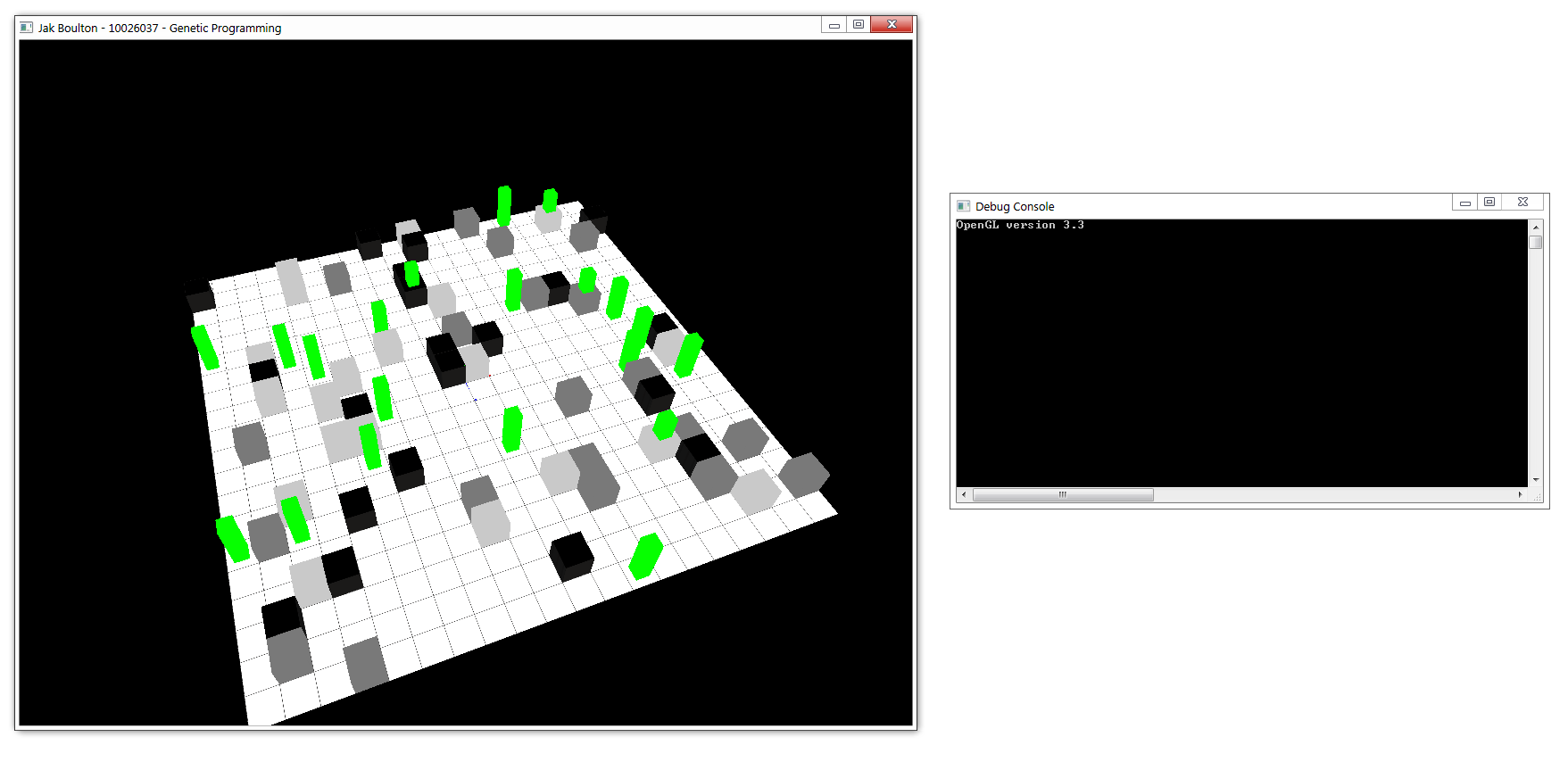


Figure 4.3 – 4 *Initial view of the Painted Desert scene*

Figure 4.3 – 4 shows the initial view of the Painted Desert scene. Green rectangles represent ants while the other cubes represent the three different grains of sand. The user is able to switch back and forth between the two scenes with the use of the left or right arrow keys. Once the user presses the space key, the evolution for the scene currently selected (and on display) begins. The Painted Desert AI controls the GP to create an initial population, fitness test each of the population and evolve the population using crossover. The best individual is then taken for the animation. The difference between the path AI and the Painted Desert AI is how the animated sequences are stored, for the path AI; a vector of positions is stored and the agent moves through the positions until the destination is reached. The Painted Desert AI however is stored as a string of characters for each of the AI ants and through each update iteration the next character is checked and the corresponding function is called.

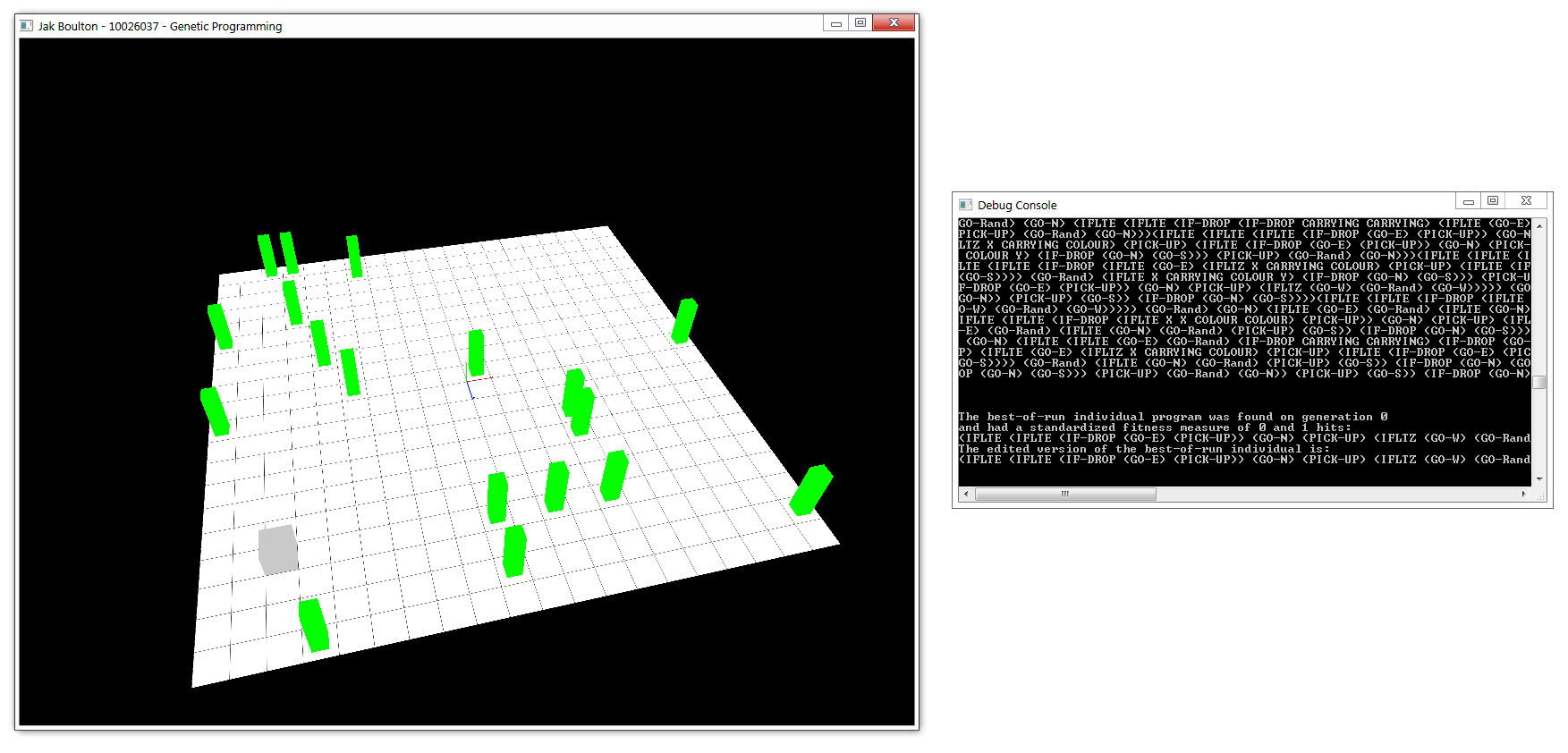


Figure 4.3 – 4 *View of the scene when the GP generated programs that did not solve the problem*

Figure 4.3 – 4 shows a screenshot taken mid-development of the Painted Desert problem. There was a problem with the fitness function causing the AI ants to simply collect each of the grains of sand which were not in the correct vertical band. This problem was caused because the fitness function calculated the fitness at the end of the evaluation of the program for each of the ants starting positions. The fitness function would check each of the map points and score the fitness based on how far the grain of sand was from where it should be, the problem was that after the evaluation of the program; any grains of sand were not dropped back onto the map.

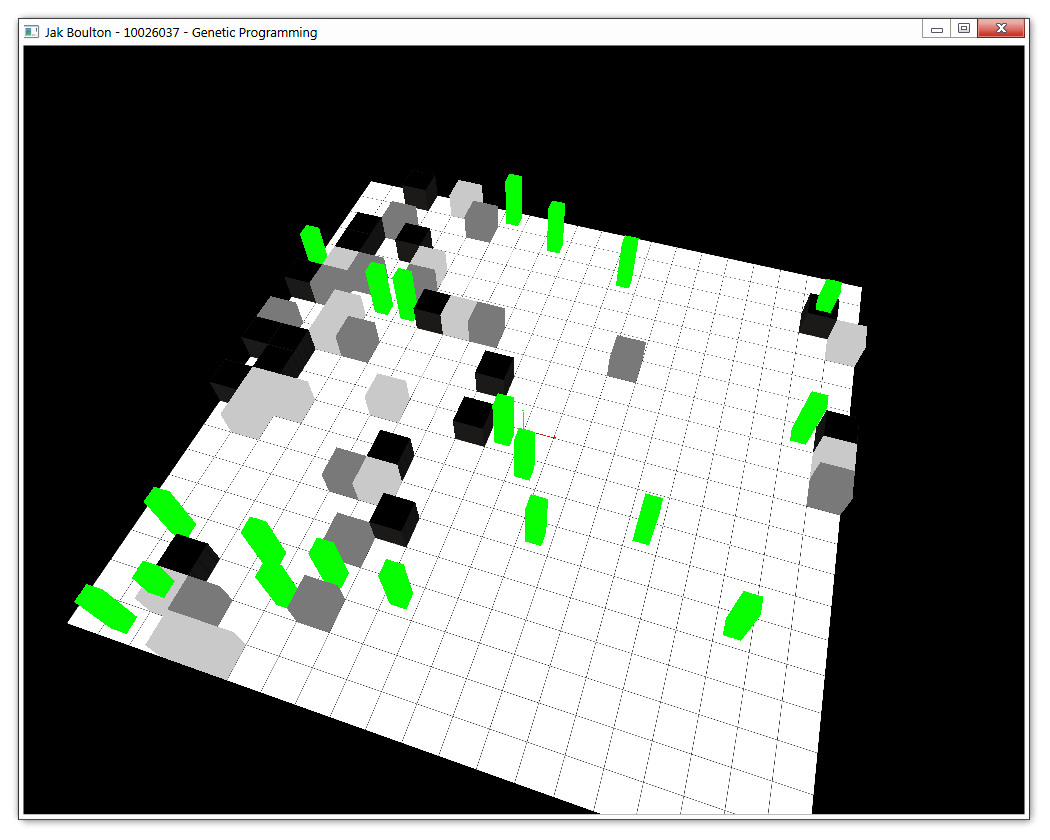
Figure 4.3 – 5 *View of the scene when the GP generated a program nearing the solution*

Figure 4.3 – 5 shows the Painted Desert scene after the fitness function and other problems were improved. Finding a perfect solution would take a group of adjustments which were decided to be left out due to the time constraints.

# 5 Testing

Once the implementation was complete, the next step was to test the program for bugs and other inefficiencies. This section details the testing undertaken to ensure the stability of the program.

## 5.1 Test Plan

Extensive testing was conducted throughout the implementation of the project ensuring that each feature was fully implemented and tested before implementing subsequent features. The main plan was to test each of the classes to ensure expected results; this includes testing of each of the more complex functions within the classes. Once the entire class was tested, the characteristics of the class as a whole were examined to ensure expected results. The testing was broken into the two main project areas; the system and the AI.

## 5.2 Testing the System

The system is comprised of a group of classes all of which can be tested simply. The classes which make up the system are PController, PMap, PGround and PBox. The controller manages the whole system and so nothing would be setup, initialised or displayed if the controller was implemented incorrectly. The map class is responsible for rendering the ground and so if the ground is rendering as expected both the map and ground classes are working correctly. The box (or cube) class is responsible for the visual appearance of the walls, grains of sand and agents. If all of the objects are rendering as expected with the correct scale and in the correct positions; the class is working as expected without error.

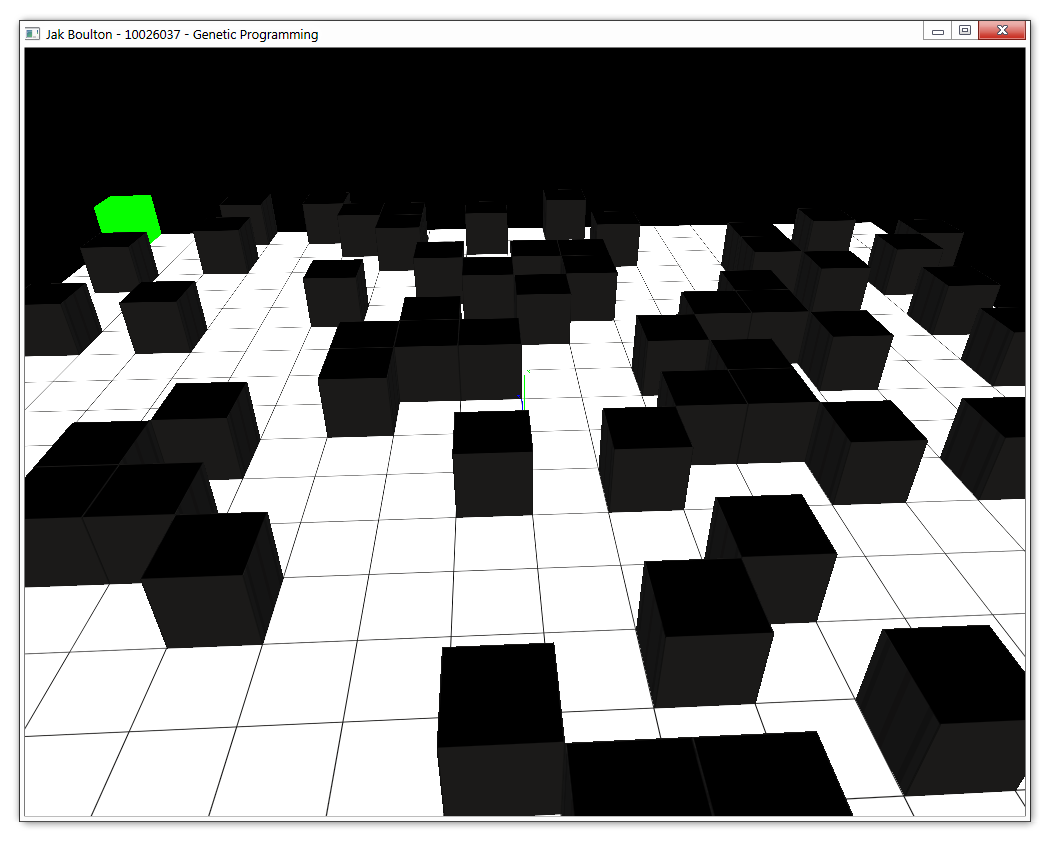


Figure 5.2 – 1 *Correct rendering of scene objects*

Figure 5.2 – 1 shows the path finding scene rendering correctly. The boxes have been rendered correctly; the ground has also been rendered correctly. The simple fact that there is a visual output in the main application window shows that not only is the controller class working but the OpenGL framework is also working correctly and OpenGL has been correctly initialised.

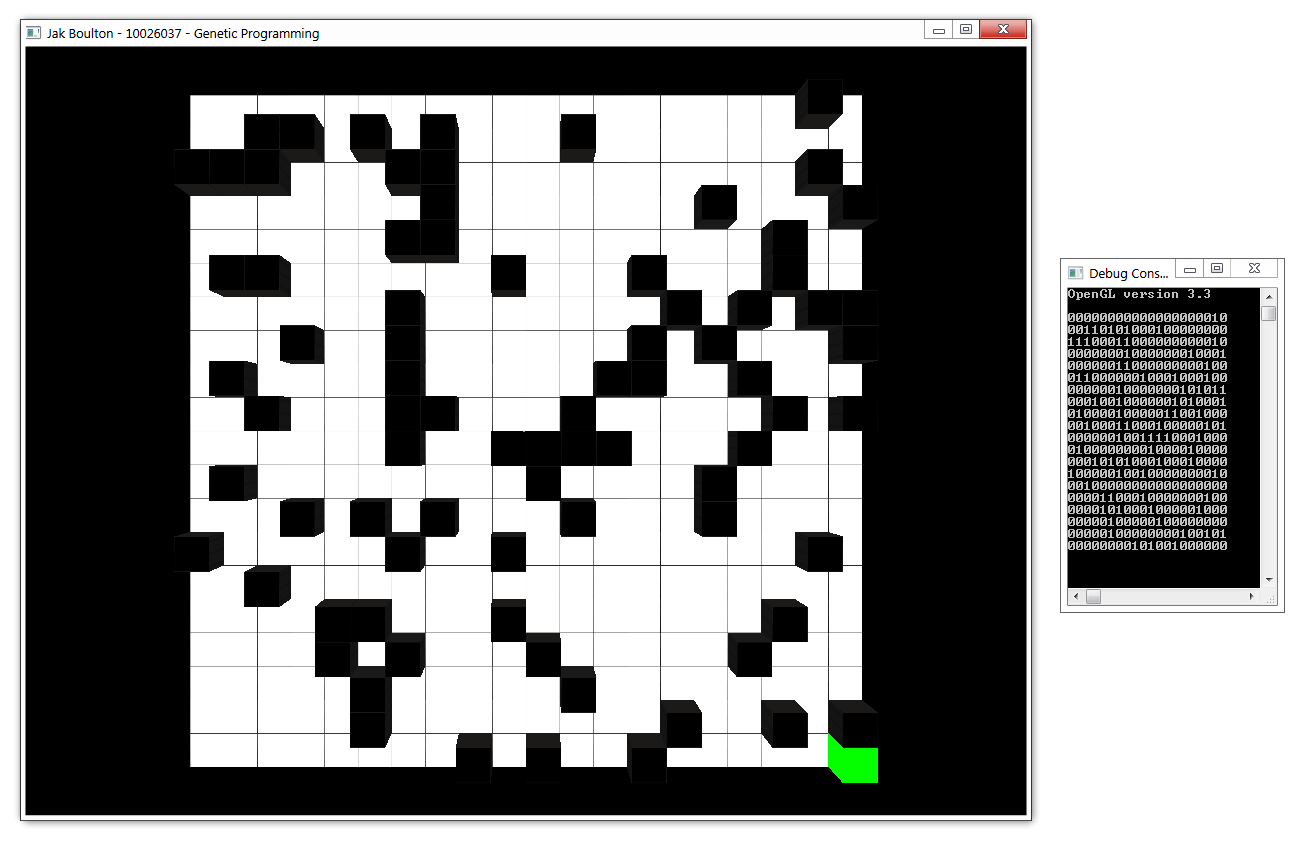


Figure 5.2 – 2 *Correct rendering positions of scene objects*

Figure 5.2 – 2 shows that the box objects within the scene have been rendered in the correct locations. The debug window shows a simple output of where each of the walls is placed on the 20x20 grid (1 for a wall, 0 for a blank grid space). All of the boxes have been rendered in the correct locations and with the correct texture colours meaning that textures and shaders have been loaded correctly.

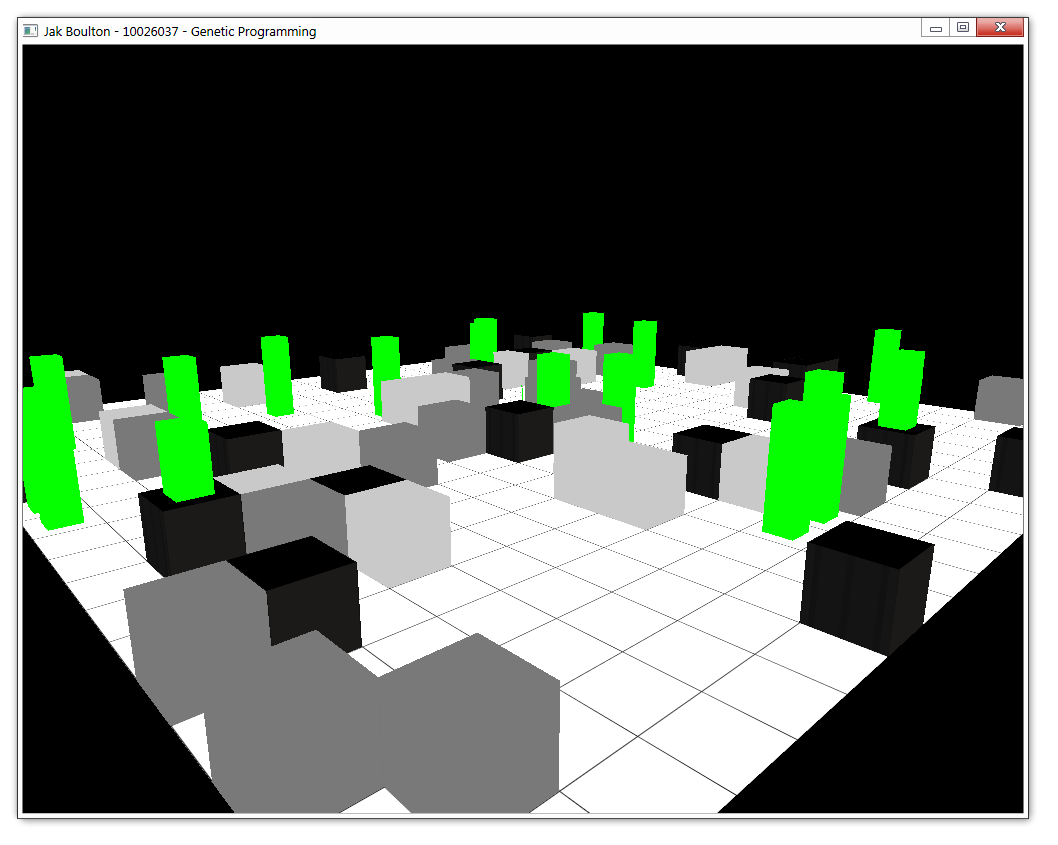


Figure 5.2 – 3 *Correctly rendering Painted Desert scene*

Figure 5.2 - 3 shows that the rendering of all the system objects is also working as expected for the Painted Desert problem.

The screenshots above all show that the rendering code for each of the objects, along with the shaders is working correctly. OpenGL has a tendency to either render correctly if the code is written without error or render nothing at all if there are errors. The correct rendering of the all objects shows that the code is working as expected.

## 5.3 Testing the AI

This is an essential part of the project testing since it is an essential part of the project aim. The testing of the AI was the largest of the testing sections since the correct setting of the GP variables and settings also had to be tested to ensure the correct running of the AI classes. There are two AI classes within the project; the path AI and the Painted Desert AI.

Path AI

The testing of the path AI consists of testing the GPs setup including the adding of the terminal and function sets and the correct implementation and expected outcome of the fitness and termination functions. The next test was to check if the animated path was the same as the best program generated by the GP.

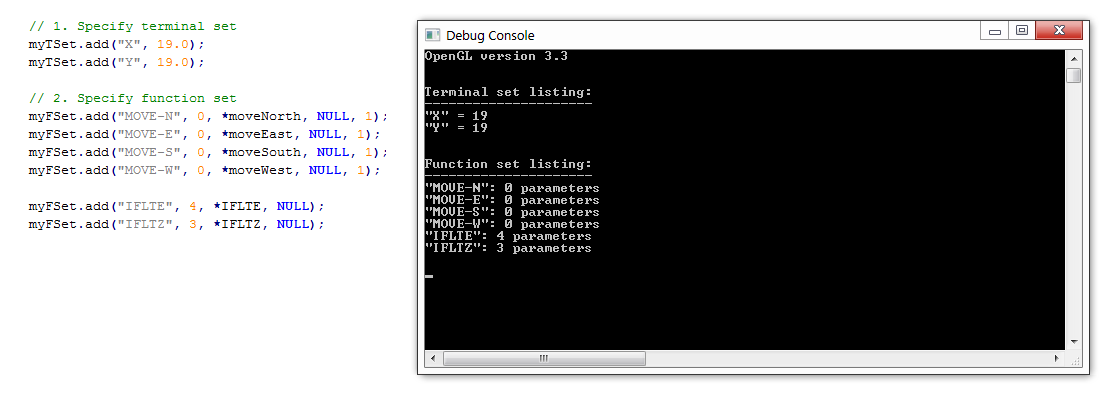


Figure 5.3 – 1 *Adding of the terminal and function sets*

Figure 5.3 – 1 shows that the terminal and function sets have been successfully added to the GP ready to generate programs.



Figure 5.3 – 2 *A general output of a GP run*

Figure 5.3 – 2 shows a general GP run in terms of the console output. This shows that the termination criteria function is working correctly since the GP has stopped on a generation with a single hit. The figure also shows that the GP is correctly generating programs using the specified terminal and function sets. The next and arguably the most difficult part to test was the fitness function, it is the most important part of the evolutionary process and therefore had to be working as expected. This was the function that took a great deal of testing during the implementation process of the project, as mentioned in section 4.2.5. There were a multitude of problems during the implementation but the function is now generating the expected results causing the GP to generate better and better programs as they are evolved.

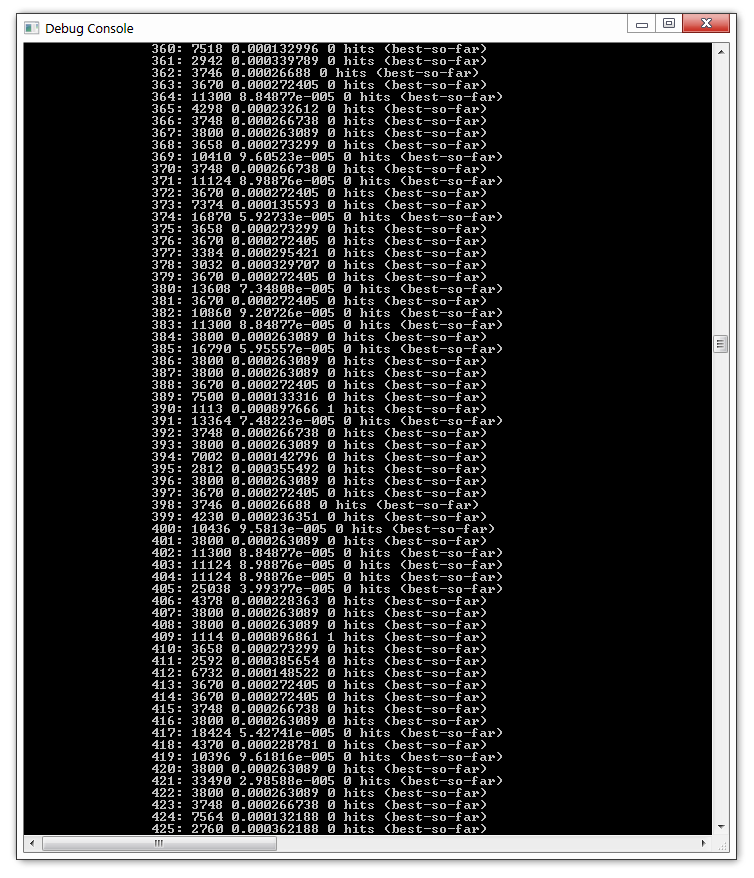


Figure 5.3 – 3 *Section of program fitness test results*

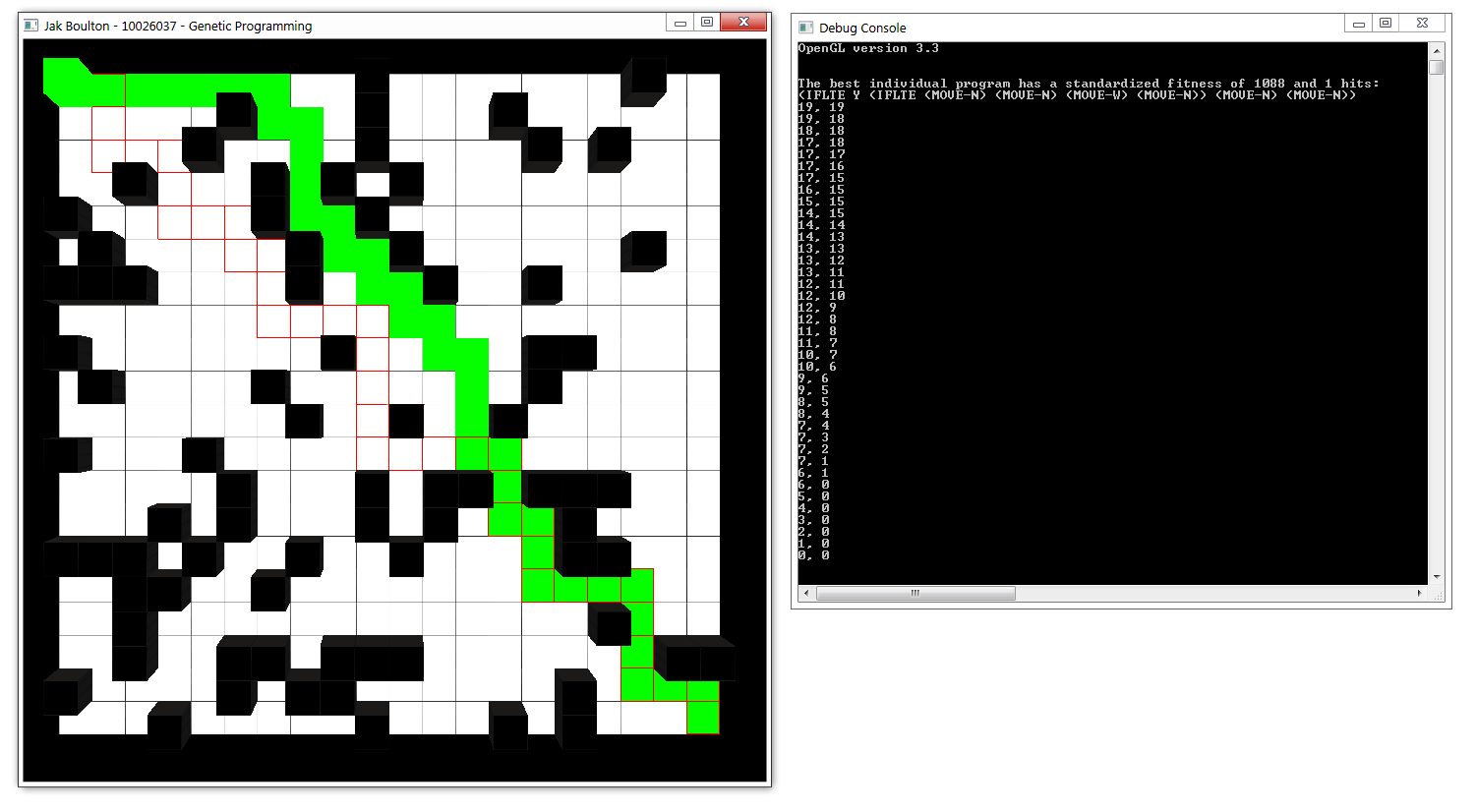


Figure 5.3 – 4 *Scene with path output*

Figure 5.3 – 4 shows that the program generated has been translated into a path correctly and that the collected path has been displayed correctly. This also shows that the tile class has been implemented correctly since it has been rendered along both the A\* and GP paths correctly.

Painted Desert AI

The Painted Desert problem is solved in a similar way to the path AI problem; using the GP to generate programs, evolve them and display the output to the main application window. Since the problems are solved in the same manner, they can be tested in the same way to make sure that the GP is generating programs using the terminal and function sets. The fitness and termination functions and correct translation of the programs from data to animation also need to be tested.

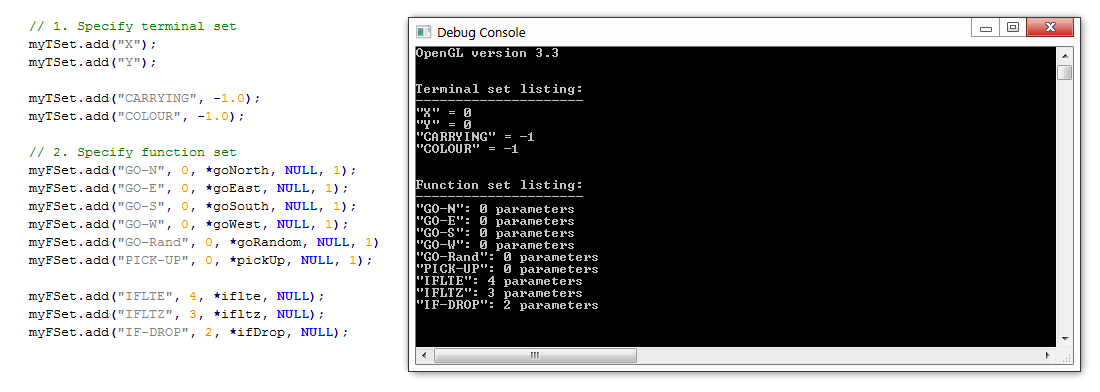


Figure 5.3 – 5 *Adding of the terminal and function sets*

Figure 5.3 – 5 shows that the terminal and function sets have been successfully added to the GP ready to generate programs.



Figure 5.3 – 6 *A general output of a GP run*

Figure 5.3 – 6 shows a general GP run in terms of the console output. This shows that the termination criteria function is working correctly since the GP has stopped on a generation with a single hit. The figure also shows that the GP is correctly generating programs using the specified terminal and function sets. The next part to test was the fitness function; it is the most important part of the evolutionary process and therefore had to be working as expected for the programs to evolve correctly. This was the function that took a great deal of testing during the implementation process of the project, as mentioned in section 3.2.5. There were a multitude of problems during the implementation and the current implementation of the function still contains a few errors which are causing the evolution process to act unexpectedly at times.

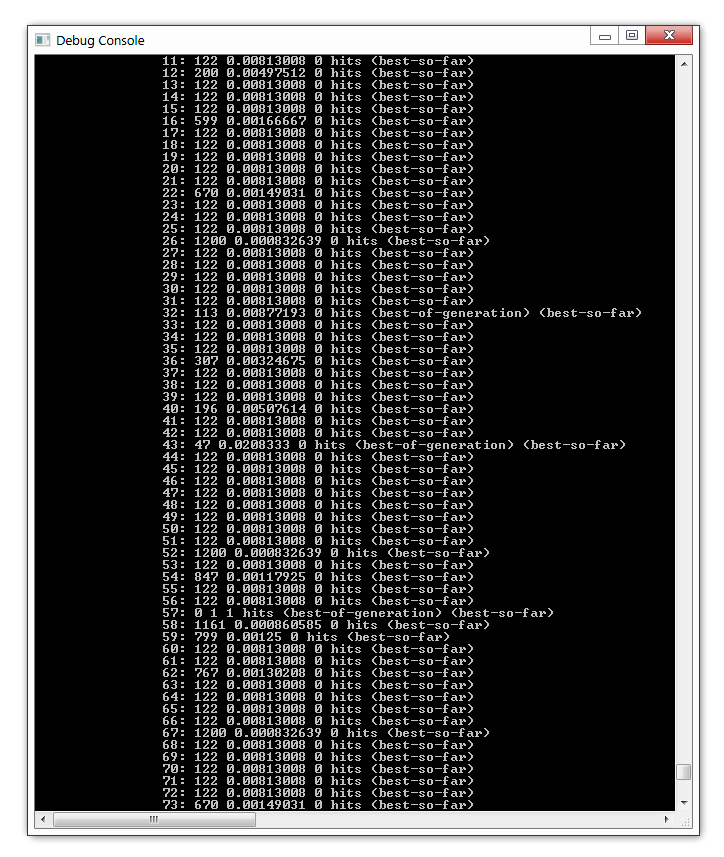


Figure 5.3 – 7 *Section of program fitness test results*



Figure 5.3 – 8 *Program output*

Figure 5.3 – 8 shows the output of the program function calls. Each of the function calls in the GP generated programs is stored within a vector of characters and when necessary, the functions are called to move each of the ants and allow them to pick-up and drop grains of sand. These function calls are all working correctly, through several tests each of the functions has been executed.

# 6 Evaluation

This was the final section of the report, used to discuss the success of the project as a whole. Simply put, does the project meet the requirements set at the start of the project development cycle?

## 6.1 Evaluation Plan

The best way to ensure the correct evaluation of the project was to check back over each of the objects and functional requirements. If the project successfully completes each of the objectives and presents each of the functions set at the start of the project development cycle, it was successful. In the sections below the functional requirements, objectives as well as possible improvements and future additions have been discussed.

## 6.2 Evaluation against Specification

The first part of the evaluation was to check against each of the objectives and functional requirements listed on pages 5 and 6. They have been included again here and evaluated to ensure a full and comprehensive evaluation of the project.

Evaluating against the functional requirements:

1. ***Display a basic three dimensional scene,***

The screenshots throughout this document prove that a three dimensional scene is displayed to the user showing all of the information required to view the evolution of both path finding and emergent group movement.

1. ***Display debug window,***

There are again, many screenshots of the debug window display working correctly throughout this document. The debug window is used to display information regarding the initial programs generated by the GP and the steps the GP takes to score, evolve and select the best individual program. When starting a general GP run, the user is shown a great deal of information about the current GP run, this allows the user to view the fitness values that the programs generated by the GP are and how they are improving after each evolution of the population.

1. ***Display a graphical representation of a path-finding AI using A\*,***

The A\* path finding can be viewed in several of the screenshots mainly within the later stages of the implementation section and throughout the testing sections. It is viewed as a red cube on the path finding problem scene and the path through the randomly generated map is graphically represented by both the animation of the cube and highlighted red tiles.

1. ***Display a graphical representation of an AI generated using the method of genetic programming,***

Within the finished prototype, there are two scenes; the path finding scene and the Painted Desert scene. The two scenes show graphical representations of agents controlled solely by programs generated using the method of genetic programming. The paths found by the green cube on the path finding scene are not simply paths, but executed functions. Each step taken by the program is stored for the graphical output.

The Painted Desert scene contains a group of 20 ‘ants’ graphically represented as rectangles, tall and green. These rectangles are controlled by programs generated through the method of genetic programming. The programs are stored as character strings (as explained in section 5.3).

1. ***Allow the user to control when the genetic program creates a new population of programs,***

The genetic program is controlled by user input, if the user wants the program to evolve additional populations in order to evolve a better solution; they would simply use the space key. If the user wanted to restart the GP so that an entirely new set of individuals were created to make up a new population; they would use the ‘N’ key.

1. ***Display the operations of the genetic programming method within the debug window,***

The operations of the genetic program are all output to the debug window, as shown within several screenshots within sections 4.3, 5.2 and 5.3. When the GP creates a population of individuals; each and every individual in the form of a string representation of an S-Expression is displayed to the debug window. The GP process, once the initial population has been created, then tests each of the population with the fitness function, all of the test results are output to the debug window. Finally, the GP selects the best-of-run individual and it is displayed to the debug window.

1. ***Display the program generated using the genetic programming method used within the graphical representation,***

The project displays two graphical scenes, both of which have a GP running in the background generating programs to solve the problems. The graphical sequences the user is able to see within both of the scenes are graphical representations of the programs generated by the GP. There are screenshots and descriptions of how the programs are stored and animated in sections 4.2.5, 4.2.6, 5.2 and 5.3.

1. ***Display a group of AIs generated using the genetic programming method showing emergent group movement,***

The Painted Desert scene was specifically developed to meet this requirement and although there are still a small group of changes that would improve the functionality of the scene; emergent group movement is demonstrated. The idea of emergent group movement is for a group of AI agents each of which have their own set of goals to work together to solve, or at least partially solve the problem at hand (Koza, 1992). Section 4.2.6 explains how the Painted Desert scene was developed.

1. ***Allow the user to switch between scenarios with ease,***

The program opens showing the path finding scene, allowing the user to simply use the space or ‘N’ keys to control the GP for that problem. The user is also able to use the arrow keys (left and right) to switch between both of the implemented scenes.

1. ***Allow the user to alter the camera position and viewing direction,***

The three dimensional scenes displayed within the application can be controlled in terms of camera position and viewing angle by using the mouse buttons and move movement. The user would simply have to hold the left mouse button and drag the cursor to alter both the viewing angle and camera position with one simple movement.

1. ***Allow the user to restart the demonstration,***

The AI classes control the resetting of the scene demonstrations, however using the space or ‘N’ key; the user is able to restart the demonstration. These keys also control the continuation of the program evolution and the generation of a new population.

1. ***Allow the user to exit the application.***

The user is able to exit the application at any point by using the “Esc” key or by pressing the close button in the top right hand corner of either the main application window or the debug window.

Evaluating against the objectives:

1. ***Produce a document containing a detailed and extensive compilation of all the research needed to complete the project.***

This main body of this document begins with a large section dedicated to the research undertaken for the project. In order to design and implement several areas of the project mentioned within the aim, areas of research were required. The project has evolved and changed a great deal over the course of the six months of research, design and development and a great deal of research around the several topics are evident within the research section (section 2).

1. ***Produce a document containing a detailed design of the project.***

The design section (section 3) begins with the high level design of the project focusing on how the overall design will work in terms of the general flow of the program. The next part of the design delves more deeply into the specifics of each of the classes’ responsibilities and interactions. The design section includes extensive descriptions of each of the classes with diagrams for clarity.

1. ***Implement the project based on the design in the previous section.***

The implementation section (section 4) contains an extensive and comprehensive description of the implementation process. It is chronologically ordered and provides descriptions for each of the classes and how they were implemented. Each of the classes in this section is also presented in a diagram format to ensure the understanding of the functionality and role of the class. The final implementation has also been developed and is shown throughout the document in the form of screenshots and diagrams.

1. ***Compile a test plan for each of the classes within the project and proceed to test the application.***

The testing section (section 5) consists of a test plan explaining how each of the classes was tested. Although much of the testing is not present within this document (each of the methods was tested during the implementation stage) the testing presented is to ensure the expected behaviour of the essential classes.

1. ***Produce a document containing an evaluation of how successful the project was based on the overall aim and objectives.***

This section (section 6) is the evaluation section and contains the critical evaluation of each of the functional requirements, objectives and the overall aim of the project.

Evaluating against the main project aim:

***To demonstrate evolving path finding and emergent group movement using genetic programming.***

Within this section, each of the functional requirements and objectives have been evaluated ensure that they have been met to a reasonable standard. The program is currently fully functional with the ability to display two graphical scenes. The path finding scene is used demonstrate evolving path finding using the method of genetic programming. The Painted Desert scene is used to demonstrate the emergence of group movement when a group of agents are given similar goals and are driven by a program generated using the method of genetic programming. Each of the scenes is functional and demonstrates their section of the overall project aim with a varying level of success. The path finding problem has been solved, the GP generates a program which is able to navigate a randomly generated map and can also be evolved further in order to successfully improve the path taken through the map. The Painted Desert, or emergent group movement scene has a varying level of success dependent on several factors and given more time; could be improved further to solve the problem fully.

## 6.3 Improvements

During the development of the program, there were varying levels of success while implementing specific features into the project. This section provides descriptions of how the sections of the implementation could have been improved to gain a better result overall. Below is a listing of each of the major improvements with descriptions of each:

1. ***PTextures***

As explained in section 3.3.4 it was originally designed to implement a texture management class under the name of “*PTextures*”. The main design of this class was to work as a resource manager for textures, a texture would be queried in the form of a file name, if the texture was not already loaded; it would be loaded and returned as a *GLuint*. If however the texture was already loaded, the class would simply return the *GLuint*. This would improve the project as a whole since in the current system a new texture is loaded for each of the instances of several classes (even if the texture is the same) which is a waste of memory resources.

1. ***Model duplication***

Another possible improvement would be to remove any case of model duplication which would improve the use of memory resource further. In the current system, when rendering the path finding map; there are a group of PBox instances representing the each of the walls on the randomly generated map. Resources could be saved if a single wall was used, but rendered at multiple positions.

1. ***PMapLoader***

The *PMapLoader* class was also planned within the design section to be implemented as part of the project; however it was not included due to time constraints. This class would work similarly to the texture resource manager described above but would instead load maps.

1. ***Map class design***

The design of the map class originally suggested it would manage the walls and grains of sand including the updating of their positions and rendering. When implementing each of the classes however, the AI classes were programmed to manage the map layouts. This does not follow the original design and would improve the encapsulation and object orientation of the classes if rectified.

1. ***PAIDesert***

As mentioned within section 6.2 (and others) there are currently a small group of errors in the fitness function causing the GP to generate unexpected results at times while generating expected results at other times. A clear improvement to the project would be to fix the errors or more precisely optimse the function to ensure the fitness scores are correct.

1. ***Shader management***

The final improvement to suggest would be a resource management class for the loading of shaders. In the current implementation, one shader is used to render all of the objects, but it is being loaded in multiple times (once per object). The problem with this is again the wasted use of memory resources.

## 6.4 Future Additions

Now that the project has been completed it is important to discuss any possible future additions which could have been added given extra time. The list below is a group of possible additions which would improve the usefulness of the project:

1. ***Larger Scene Selection***

The completed project consists of two scenes demonstrating the two parts of the aim. A possible future addition would be to add a one or more scenes to show a wider variety of problems what could be solved using the method of genetic programming. Possible additions could be as explained by Larry I. Gritz (Gritz, 1993). The cart centering problem, this is where a cart is given a random starting position and velocity on a frictionless one dimensional track. The cart has a constant force emitted by a rocket on either the right or the left. The goal is to develop a strategy or formula which tells us where the force is being applied and how to counter it. There are many more possible problems already proposed by John R. Koza, or a new problem could be proposed.

1. ***Graphical User Interface***

At this current state within the project, the user interaction in terms of the user interface is lacking. The user is able to observe a scene using mouse interaction and press specific keys on the keyboard to control when a demonstration is played out. A future addition could be the addition of a graphical user interface giving the user more control over several aspects of the demonstration as well as providing more information to user about key aspects of the scene. Rather than using the debug window as the main output for the GP, a separate context could be used within the main application window with a listing of the programs which have been generated, allowing the user to alter the programs as they are being tested or generated.

1. ***More User Interaction***

Increased user interaction in demonstration based applications helps the user to engage with the content. As stated above, the user could interact with the user interface to make alterations to the programs. Other options could be altering the maps while the demonstration is happening to see how the GP generated program would react. There is a large list of user interactions that could be added including the addition of new objects to the scenes, more control over the GP and user made changes to the function and terminal sets for each of the problems.

1. ***Custom Maps***

The completed project consists of two scenes which feature randomly generated maps, although this gives almost limitless possibilities for the map layouts, it limits the user interaction by only displaying one map each time the program is run. A welcomed addition would be the ability for users to create custom maps which could be loaded in and used with the demonstrations. If part of the improvement section “*PMapLoader*” was included in the final implementation; this would be possible.

1. ***Scalable Maps***

The maps included in the project are based on a twenty by twenty grid. This is fairly large for these types of problems since Koza and other authors of GP based problems tend to have grid sizes of ten by ten. A possible addition however, would be to allow the user to scale the map to a size of their choice. This would really challenge the GP to generate solutions to endless variations of problems and map sizes as well as increasing the level of user interaction.

## 6.5 Conclusion

To begin the project development cycle; research into all of the areas required to gain the knowledge needed to design the system was undertaken. The next step was to build a design for the entire system, from the ground up. The design began at a high level detailing the simple connections between classes before the detailed design section explained what was planned for the implementation. The implementation section had begun and each of the classes was written one after another to bring the project together. Extensive testing of the entire system was undertaken to ensure it was all working as expected. The process of the research, design and development of the system was complete. This document was compiled, bringing the whole process to one place.

Evolutionary path finding and emergent group movement are both demonstrated within the final implementation of the project. Improvements could be made to the overall system as explained in section 6.3, including better management of resource loading to improve the performance of the system. The Painted Desert problem demonstrating emergent group movement has not been as successful as hoped but the evolution of path finding is working as expected and is able to evolve paths which reach the destination as fast as A\*. Many future additions as explained in the previous section (6.4) would improve the usability of the program and allow users to extend the program indefinitely. The final conclusion; the functional requirements, objectives and overall aim were all completed with success in all areas.

# **Bibliography**

Brownlee, J., 2011. *Clever Algorithms: Nature-Inspired Programming Recipes.* 1st ed. Austin: Jason Brownlee.

Champandard, A. J., 2010. *Pathfinding in Static and Dynamic Environments and the Future of Multi-core HPA\*,* s.l.: AI Game Dev.

Gritz, L. I., 1993. *A C++ Implementation of Genetic Programming,* Washington: s.n.

iGunSlingeRv2, 2011. *A\* (A-Star) Pathfinding C++ Tutorial 1 - part 1/4.* [Online]   
Available at: https://www.youtube.com/watch?v=NJOf\_MYGrYs  
[Accessed 16 March 2013].

Koza, J. R., 1992. Evolution of Emergent Cooperative Behavior using Genetic Programming. In: R. Paton, ed. *Computing with Biological Metaphors.* California: Chapman & Hall., pp. 1-23.

Koza, J. R., 1992. *Genetic Programming.* 1st ed. Massachusetts: The MIT Press.

Koza, J. R., 1994. *Genetic Programming.* 2nd ed. s.l.:The MIT Press.

Koza, J. R., 1999. *Genetic Programming.* 3rd ed. s.l.:Morgan Kaufmann Publishers.

Liden, L., 2002. Strategic and Tactical Reasoning with Waypoints. In: S. Rabin, ed. *AI Game Programming Wisdom.* Hingham: Charles River Media, pp. 211-220.

Matthews, J., 2002. Basic A\* Pathfinding Made Simple. In: S. Rabin, ed. *AI Game Programming Wisdom.* Hingham: Charles River Media, pp. 105-113.

Obitko, M., 1998. *Genetic Algorithm.* [Online]   
Available at: http://www.obitko.com/tutorials/genetic-algorithms/ga-basic-description.php  
[Accessed 17 January 2013].

Patel, A., 2012. *Introduction to A\*.* [Online]   
Available at: http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html#dijkstras-algorithm-and-best-first-search  
[Accessed 17 January 2013].

Poli, R., 2008. *A Field Guide to Genetic Programming.* 1st ed. Essex: lulu.com.

Rabin, S., 2002. *AI Game Programming Wisdom.* 1st ed. Hingham: Charles River Media.

Richard S. Wright, N. H. G. S. a. B. L., 2010. *OpenGL Superbible.* 5th ed. s.l.:Addison-Wesley Professional.

Rojas, R., 1996. *Neural Networks: A Systematic Introduction,* Berlin: Springer.

Rojas, R., 1996. The Back Propagation Algorithm. In: *Neural Networks.* Berlin: Springer-Vertag, pp. 151-184.

Ross Graham, H. M. a. S. S., 2003. Pathfinding in Computer Games. *ITB Journal,* Issue 8, pp. 10 - 13.

Sterren, W. v. d., 2002. Squad Tactics: Planned Maneuvers. In: S. Rabin, ed. *AI Game Programming Wisdom.* Hingham: Charles River Media, pp. 247-259.

Sterren, W. v. d., 2002. Squad Tactics: Team AI and Emergent Maneuvers. In: S. Rabin, ed. *AI Game Programming Wisdom.* Hingham: Charles River Media, pp. 233-246.

Walker, M., 2001. *Introduction to Genetic Programming,* s.l.: s.n.

Walsh, P., 2008. *Advanced 3D Games Programming with DirectX 10.* 1st ed. Texas: Wordware Publishing.

Woodcock, S., 2002. Recognizing Strategiv Dispositions: Engaging the Enemy. In: S. Rabin, ed. *AI Game Programming Wisdom.* Hingham: Charles River Media, pp. 221-232.

Section 6 – Bibliography/Evidence of Research

[1] Baris Simsek in *Libraries*, 2004. Accessed on December 28th 2012.

[2] General Public License, (<http://www.gnu.org/licenses/gpl.html>), June 29th 2007. Accessed on December 28th 2012.

[3] Free Software Directory, (<http://directory.fsf.org/wiki/License:APSLv1.x>), January 4th 2001. Accessed on December 28th 2012.

[4] OpenGL, (<http://www.opengl.org/documentation/current_version/>), August 6th 2012. Accessed on December 28th 2012.

[5] OpenGL Licensing, (<http://www.sgi.com/products/software/opengl/license.html>), 2009. Accessed on December 28th 2012.

[6] BSD License Definition, (<http://www.linfo.org/bsdlicense.html>) , April 22nd 2005. Accessed on December 28th 2012.

[7] Mozilla Public License, (<http://www.mozilla.org/MPL/>), February 1st 2012. Accessed on December 28th 2012.

[8] SGI Free Software License B, (<http://oss.sgi.com/projects/FreeB/>), September 18th 2008. Accessed on December 28th 2012.

[9] GNU Operating System, (<http://www.gnu.org/copyleft/>), June 10th 2012. Accessed on December 28th 2012.

[10] DirectX Graphics and Gaming (Windows), (<http://msdn.microsoft.com/en-US/library/windows/apps/ee663274.aspx>) , November 11th 2012. Accessed on December 28th 2012.

[11] Microsoft Software License Terms, Microsoft DirectX Software Development Kit (SDK), August 14th 2009. Accessed on December 28th 2012.

[12] Android Developers, (<http://developer.android.com/about/index.html>), N/A. Accessed on December 28th 2012.

[13] Tech Crunch, (<http://techcrunch.com/2012/07/18/adfonic-android-tops-ios-as-most-popular-platform-on-global-ad-network-iphone-ipad-still-top-devices/>), July 18th 2012. Accessed on December 28th 2012.

[14] Game Dev, (<http://www.gamedev.net/page/resources/_/business/linux-as-a-gaming-platform-r1135>) July 25th 2000. Accessed on December 28th 2012.

[15] Richard Raysman in *Intellectual Property Licensing: Forms and Analysis,* 2006. Accessed on January 3rd 2013.

[16] WIPO IP Services, (<http://www.wipo.int/patentscope/en/patents_faq.html#patent>), February 27th 2009. Accessed on January 3rd 2013.

[17] LexisNexis, (<http://w3.nexis.com/sources/scripts/info.pl?278252>), October 1st 2004. Accessed on January 3rd 2013.

[18] Section 39: Right to employees’ inventions, (http://www.ipo.gov.uk/practice-sec-039.pdf), April 2008. Accessed on January 3rd 2013.

[19] Patents for software?, (http://documents.epo.org/projects/babylon/eponet.nsf/0/a0be115260b5ff71c125746d004c51a5/$FILE/patents\_for\_software\_en.pdf) , 2012. Accessed on January 3rd 2013.

[20] CiteSeer, (<http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.32.8093>), 2000. Accessed on January 3rd 2013.

[21] Search Engine Land, (<http://searchengineland.com/will-software-patents-including-on-search-algorithms-go-away-14421>), July 22nd 2008. Accessed on January 3rd 2013.

[22] The Business of Software, (<http://discuss.joelonsoftware.com/default.asp?biz.5.764733.15>), July 3rd 2009. Accessed on January 3rd 2013.

[23] Paul Graham, (<http://www.paulgraham.com/softwarepatents.html>), March 2006. Accessed on January 3rd 2013.

[24] DP Review, (<http://forums.dpreview.com/forums/post/3049656>), July 18th 2002. Accessed on January 3rd 2013.

[25] Machine Learning, (http://yaroslavvb.blogspot.co.uk/2011/02/how-to-patent-algorithm-in-us.html) , February 21st 2011. Accessed on Januray 3rd 2013.

[26] IEEE Golbal History Network, (http://www.ieeeghn.org/wiki/index.php/History\_of\_Lossless\_Data\_Compression\_Algorithms) , 2012. Accessed on Januray 3rd 2013.

[27] The Register, (<http://www.theregister.co.uk/1999/09/01/unisys_demands_5k_licence_fee/>), September 1st 1999.

[28] Patentability Pending, (<http://patentability-pending.com/news/us-news-limitation-patentability-algorithms>), December 24th 2009. Accessed on January 3rd 2013.

[29] Path Finding – A\* Algorithm, (<http://www.edenwaith.com/products/pige/tutorials/a-star.php>), December 26th 2002. Accessed January 3rd 2013.

[30] Fiona Nielsen in *Neural Networks – Algorithms and applications* (December 12th 2001). Accessed on January 3rd 2013.

[31] The Backpropagation Algorithm, (http://www.speech.sri.com/people/anand/771/html/node37.html) , October 16th 1999. Accessed on January 3rd 2013.

[32] IPO (Intellectual Property Office), (<http://www.ipo.gov.uk/types/patent/p-other/p-infringe.htm>), October 5th 2011. Accessed on January 3rd 2013.