Title page

DECLARATION

I, Callum Terris  confirm that this work submitted for assessment is my own and is  
  expressed in my own words. Any uses made within it of the works of  
  other authors in any form (e.g., ideas, equations, figures, text,  
  tables, programs) are properly acknowledged at any point of their  
  use. A list of the references employed is included.  
 Signed:..................................................................

Date: ...................................................................... ''

Contents

[1 Introduction 5](#_Toc361745825)

[2 Literature Review 6](#_Toc361745826)

[2.1 Game Engines 6](#_Toc361745827)

[2.2 Current Game Standards 9](#_Toc361745828)

[2.3 Evolutionary Games 11](#_Toc361745829)

[2.4 Neural Networks 14](#_Toc361745830)

[2.5 Interfacing In-between Games 17](#_Toc361745831)

[2.6 Literature Review Conclusion 18](#_Toc361745832)

[3 Methodology 19](#_Toc361745833)

[3.1 Prototype Method 19](#_Toc361745834)

[3.2 Tutorial 19](#_Toc361745835)

[4 Results 21](#_Toc361745836)

[5 Discussion 22](#_Toc361745837)

[6 Conclussion 23](#_Toc361745838)

[6.1 Future work 23](#_Toc361745839)

List of Tables

[Figure 1 One weapons evolution at various generations. The above image shows how one weapon evolves from generation to generation. Image was taken from (Hastings et al., 2009) paper. 12](#_Toc362111190)

[Figure 2 Basic Multi-layer ANN layout. The topmost layer is the output neurons. The middle layer is the hidden layer and the bottom layer is the inputs. Image taken from AI Techniques for Game Programming (Buckland, 2002) 15](#_Toc362111191)

[Figure 3: An example of how two parents combine to make a child. Image taken from (Stanley and Miikkulainen, 2002a) 16](#_Toc362111192)

[Figure 4 Shown below 24](#_Toc362111193)

# Introduction

# Literature Review

## Game Engines

This project will require a game engine. A game engine is a tool that allows for developers to create games on. Think of it like a framework that contains all the tools that a game developer would generally need.

With a wide number of game engines available for use in this project, there needs to be criteria to select the game engine that will be the most suitable for this project.

The first piece of criteria will be that the game engine is free to use. This project requires the game engine be free to use, wither that being an open source game engine or a professional engine that is free to use for academic use.

The next piece of criteria is that it is quick to learn. Due to the scope of this project and the time limit available, the author feels that in choosing a game engine that will take 6 months to learn how to code for is not applicable for this project. Therefore the game engine must be straightforward to develop for.

Next is the level of access available to the developers to the game engine. In order for an interface to sit between the game engine and an external application the developer will need access to some of the lower level functionality of the game engine. This could include things like networking features, restricting certain override functions. This will be needed when it comes to synchronising between the interface and the game engine.

Criteria of the game engine that this project is not concerned with are features like if the game engine is 2D or 3D, the overall look of the end game (graphics), sound capabilities and release platforms. These features are not exactly needed for this project therefore they should not be taken into consideration when deciding upon a game engine.

Based on these criteria the following game engines have been selected:

### Unreal Engine

The Unreal Engine was first developed by Epic games in 1998. Currently on its third version which was released in 2007. This is a professional game engine that a lot of industry game developers use for AAA titles. Such games include Batman: Arkham series (2009) and the BioShock series (2007) were created in this engine. This engine is free to use for non-commercial use (Games), meaning that it can be used for free in this project. This engine uses its own scripting language called UnrealScript. This game engine is highly optimised and has a wide range of documentation available. This engine also allows for development on a wide number of platforms; such as PlayStation 3 and Xbox 360. While this engine is one of the industry standards, the fact that it uses its own language that the author will have to learn as well as the engines inner workings, makes this engine an unlikely choice due to time constraints.

### Cry-Engine

This engine was developed by Crytek and has been featured in many AAA titles, such as the Crysis series (2007). This game engine has scripting in LUA and has C++ in the game engine. While these are both great languages, which are used in professional game development, the time it will take to learn not just the engine but the languages as well makes this unlikely a choice. The cry-engine is also free to use, for none commercial use (Crytek). Since this project will not be released then this fully complies with their licensing. While this is a fully valid choice for this project. Having the author learn new languages and a game engine isn’t practical. Therefore this engine will be unlikely to be chosen.

### Unity3D

Unity3D is a game engine that has been recently became a wide hit with the indie game development community (McKleinfeld, 2012). This is due to its ease of programming for and the fact that it is free to use. There are two versions of this game engine, free and pro. The pro version allows developers to use the more advanced features and removes watermarks (Technologies). The game engine is a full professional game engine; it was created by professionals, not just an open source game engine that a group of people have hacked together. Along with the pro version, developers can buy licences for certain platforms such as Android, Xbox 360 and PlayStation 3 to name a few. As for languages the game engine supports three natively. These are C#, JavaScript and Boo (language based on python). All three of these languages are relatively simple to develop in.

### Blender

Blender is an open source 3D modelling tool that has a game engine built in (Foundation). Since it is open source then that means that this meets the free to use criteria. Also it allows the developer to access the lower features of the game engine. It is written in python, which is a relatively simple language compared to other game engines. With above features it makes it a strong contender for this project. The only drawback is the fact that blender is a 3D modelling tool with a game engine inside it. Other options are a fully-fledged game engine, whereas this contains nowhere near as much functionality as the others.

### Writing a game engine

The author could choose to write their own game engine. This is a fully possible. This has a number of drawbacks and has a number of positives that can come of this. Firstly the author would know all the functionality that the game engine has. The game engine could also be created with the objectives in mind, allowing for easier development later. These are two valid reasons why to create a game engine. There are, however, a large number of drawbacks. First one being time, the project is already pretty ambitious. Creating a fully working game engine would take up a large amount of time. Next is features, this would be far lacking in features compared to the commercial ones. With only the basics inside the game engine some things can be hard to do. Speed would be another problem, even with an optimised engine this project could slow the game down. Having an already slow game engine would just make matters worse.

With all of these in mind this project is not likely to have a game engine written for it, instead it will use a pre made one.

### Game Engine Conclusion

Out of the four game engines listed only two are likely to be used within this project. These are Blender and Unity3D. Unreal and the Cry-Engine are more focused on cutting edge software and therefore have a steep learning curve. Also both of these are in high performance languages in terms of speed, which the author would need to lean. Learning both a new language and a new engine is not really applicable for this project. This is solely due to time constraints. Therefore the two game engines to choose from are Unity3D and Blender. While Blender is a valid option, it is at its heart a 3D modelling tool not a game engine. While it has one featured inside it, it is nowhere as detailed and optimised as the Unity3D game engine. The Unity game engine, while missing the advanced features as the industry standard engines, is still a powerful engine.

## Current Game Standards

Artificial intelligence has always taken a back seat within games industry. The drive of the industry is better looking graphics (Handrahan, 2011). Game AI: The State of the Industry (Woodcock, 1998), while this article is dated, it shows how little resources the games industry dedicated to AI. This article shows that AI gets around 10% of the total CPU cycles.

At the present there are two parties in artificial intelligence, game developers and academic researchers as defined in the book Artificial Intelligence for Games (Millington and Funge, 2009). The book goes on to define that game developers are only interested in the engineering side, making hacks to make characters appear to be life like. Academic AI on the other hand is based on solving problems; this can be nature based, psychology based or engineering based.

Currently games industry uses Pathfinding, finite state machines and steering behaviours(Sweetser and Wiles, 2002), and that is about it. More advance techniques are not used, such as bio inspired techniques. This is due to a number of reasons, mainly due to developers focus. In the games industry there is one main focus, graphics.

Another reason is processor constraints as mentioned in Current AI in Games: A review (Sweetser and Wiles, 2002). This paper goes on to mention the drawbacks of using more advanced AI techniques within games. The paper states that game developers are reluctant to produce games that have learning techniques, such as neural networks and genetic algorithms, in case they develop/learn stupid behaviours. Also in the case of genetic algorithms they are very computationally expensive, something the game cannot have due to the amount of other tasks that need to be carried out.

While most modern games only take advantage of steering behaviours, state machines and A\*, there have been a few commercial games that have been released with more advanced AI techniques.

These games include the Black & White series (2001) which feature the player praising or punishing the in game character based on the characters actions. For example if the creature attacks someone then you can punish it, therefore it knows that attacking people is wrong. Both these games were reviewed positively, the first game getting a 90/100 on Metacritic(2001).

## Evolutionary Games

As discussed above, current game developers are reluctant to use more advanced artificial intelligence techniques in their games. Although some academic researchers have tried to prove that these techniques can be used within games.

While most of these do not go on sale, they instead become freeware, they are still games.

### Galactic Arms Race

(Hastings et al., 2009)

Galactic Arms Race is a project created by students at the University of Central Florida. This project was aimed to “automatically generate complex graphic and game content in real-time through an evolutionary algorithm based on the content players liked” (2009). This was achieved through the game Galactic Arm Race using their cgNEAT algorithm.

The game features weapons that evolve to the players’ preference, the player can only have three weapons at a time and they have the ability to throw away/pick up new weapons. Each weapon fires particles, the number and strength of each particle remains constant in every weapon. Each weapon is also a neural network, and at each frame of the weapons firing animation, parameters are passed through to control the animation and the colour of the particles.

When the player fires a weapon, its fitness increases by one and all the other weapons fitness’s decrease by one. This is to stop the generation of previously favoured weapons from previous generations having extremely high fitness’s and therefore always being selected during crossover.

Since the player selects the weapons they want in the population then the algorithm does not have to worry about replacing members of the population.

Therefore the project was considered a success. The project not only successfully generated content in real time but it also generated content to the player’s preference.

This results in strange animations that the developers never even thought of. The figure below shows just one example of a weapon and the children it creates.



Figure One weapons evolution at various generations. The above image shows how one weapon evolves from generation to generation. Image was taken from (Hastings et al., 2009) paper.

### Nero

The paper Evolving Neural Network Agents in the NERO Video Game (Stanley et al., 2005) aims to show that they can evolve the agents within the games behaviours at real time. To show that they can they use the game NERO as a test bed.

They use an offset of the NEAT algorithm for evolving neural networks called rtNEAT. The NEAT algorithm will be explained in 3.4 Neural Networks in more detail.

In game the player is given sliders, these sliders relate to the behaviours that the player wants. The slide selects how much praise/punishment to give the agent for their behaviours in game. For example if the player wants the agents to move in close to the enemy then the slider for distance to the enemy will be at maximum. If the player wants the agents to move far away from the enemy and shoot them, then the distance from enemy slider will be at maximum punishment but the shoot enemy slider will be at maximum praise. It’s with these sliders that the player can evolve complex behaviours.

The player can alter the environment during while training the agents. This could include placing walls that the agents must move around, placing enemy soldiers etc. These are used by the player to try and get the desired behaviour from the agents.

It’s with these sliders that relate to the fitness of the agent. The fitness is determined by the player.

When training the agents, the replacement of agents happens constantly. It doesn’t destroy almost every member at once like normal genetic algorithms; instead it constantly replaces agents with lower fitness’s with an offspring of two of the fitter agents.

With the rtNEAT algorithms flexibility behaviours can be altered in at real time in training mode. In battle mode the player selects their evolved population and battles another evolved population. During the battle no evolution happens, the agents do not learn during the battle. It is more of a test to see who has the better army of agents.

### Conclusion

While both of the above games are great examples of evolution in games they both suffer from the same drawback, the time it takes to evolve. While both of these games keep the player engaged during the evolution process finding the optimum solution takes an extremely long time. No player wants to play for a large number of generations to wait to get the optimum weapon/agent.

## Neural Networks

The games discussed above in section 3.3 Evolutionary Games both games used an Artificial Neural Network. Artificial Neural Networks are a widely used tool for learning in computing.

Artificial neural networks are inspired by the brain. Simply the brain is made up of neurons and the connections between them. This is what ANN is trying to mimic. There are many different architectures for neural networks. Some of these will be described below

### Single-layer Feedforward Architecture

Single-layered feedforward architecture is a simple neural network. There are only two types of neurons; input neurons and output neurons. In this architecture all input neurons are connected to output neurons. A neuron also contains an activation function. When the neuron receives values from all of its inputs, it sums them all up. The activation function will pass a value to the output connections based on the output of the activation function. For example if a step function was used as the activation function, if the total input value was greater than the threshold then the function would output 1, but if it didn’t meet the threshold then it would return 0. The connections between the neurons have a weight associated with it. It is with these weights that the neural network can learn. By altering the weight of a connection, it can change the output from the activation function.

### Multi-layered Feedforward Architecture

This architecture is similar to the single-layer architecture described above. There is one key difference, the hidden layer. In this architecture there are three types of neurons; input, output and hidden. The difference between these two architectures is that instead of all the inputs feeding directly into the outputs, they feed into the hidden layer. The hidden layer contains hidden neurons. There can be multiple hidden layers in this network. All inputs feed into the hidden layer then into the output neurons.

Figure 3 below shows the basic layout of a multi-layer feedforward ANN. The inputs feed into the hidden layer. The hidden layer outputs to the output layer. Every neuron is connected to every neuron in the layer above it. Every connection also has a weight (not shown in figure).



Figure 2 Basic Multi-layer ANN layout. The topmost layer is the output neurons. The middle layer is the hidden layer and the bottom layer is the inputs. Image taken from AI Techniques for Game Programming (Buckland, 2002)

The same as the architecture above, all neurons have an activation function and all connections have a weight.

Learning can be accomplished by using a Genetic Algorithm (GA) to evolve the weights of the connections. The fitness of the GA can be measured on what the output is compared to what the desired result is.

### NEAT algorithm

This algorithm was created by Ken Stanley and RitsoMiikkulainen in 2002, the paper Evolving Neural Networks through Augmenting Topologies (Stanley and Miikkulainen, 2002b) describes this.

The simplest description of the NEAT algorithm was found in AI Techniques for Game Programming (Buckland, 2002). Buckland explains it simply and clearly to the reader. The genome for a possible solution is made up of two parts, the list of neuron genes and a list of link genes. It is these link genes that contain the connections between the neurons. It also contains data about the connection, such as its weights, if it is active and an innovation number. The neuron cells have data about what type of neuron they are, an input, output or a neuron in the hidden layer.

The chromosome contains all the neuron genes and the link genes. The evolution is similar to the normal evolution of a neural network but there are a lot more parameters that can be altered. This includes adding new connections and neurons to the network. During evolution connections can be disabled, meaning that when running the neural network nothing will be sent through that connection.

This algorithm was used in both of the two projects mentioned in section 3.3 Evolutionary Games. This is because it is a powerful algorithm for evolving neural networks.



Figure : An example of how two parents combine to make a child. Image taken from (Stanley and Miikkulainen, 2002a)

## Interfacing In-between Games

### ACI EAI

This topic is new to the games industry. There is only one other project like this and that is Atlantis Cyberspace Inc.’s Engine Agnostic Interface. This is a piece of middleware that sits in-between the game engine and the simulation software. The key difference between this project and their middleware tool is context; this project is aimed at games, whereas they are aimed at simulations. The information obtained from their website provides little in the way of detail of the system. Since this project costs money and no documentation can be found the author cannot detail this system any further.

## Literature Review Conclusion

As discussed above, this project will need a game engine. The chosen game engine will be the Unity3D game engine. This is due to its ease to develop for. The game itself will act as a test bed for the interface and the neural network. Therefore the game engine will need to be quick and easy to develop a game in. Unity3D meets all the requirements stated above in section 3.1 Game Engines. The last feature that swayed the authors choice was the amount of documentation available for this engine. The amount of documentation given by the developers is large and in-depth. Also there is a strong developer community with forums and wiki pages devoted to game development in Unity3D.

Chapter 3.3 Evolutionary Games shows that games with evolutionary artificial intelligence techniques can be created. These games not only work but the show that these techniques can be used in real time in games.

Section 3.2 Current Game Standards discusses the current standard of the artificial intelligence in the games industry. This section was aimed to show how much of a difference there is between the techniques currently being used in artificial intelligence and the ones being used in the games industry.

Section 3.4 Neural Networks discussed the basics of a neural network. A simple multi-layer feedforward neural network will be implemented in this project. This section also detailed the NEAT algorithm. This will not be implemented in this project, but if time constraints allow it, it may be created. This would solely be used to testing to see if this approach made any dramatic change to the results.

# Methodology

## Prototype Method

This project will take the prototype development approach to development. This means that over the course of development many separate pieces of the overall project will be created. The first prototype will be very basic but the prototypes will increase in difficulty and complexity. This allows the author to focus on smaller parts of the project one at a time, instead of trying to do the full thing from the start. It also gives the author something to fall back on if the end product cannot be done. This was a concern for this project as no other projects like this was found.

### Prototype One

Prototype one will not feature the interface. Instead it will be built within the game engine. This will feature a single bot within an environment that will move around using the wander steering behaviour. The bot moves forward at a constant speed but a random amount of rotation is added to it. The amount of rotation is limited to a specific range, -10 to 10, to give a more fluid behaviour. If the rotation values were larger the bot could move around extremely jerky. These values are a product of trial an error. The reason the minimum is a negative value is due to the fact that the bot must be able to rotate left, not just right. If the minimum value was limited to 0 then the bot would only move forward or right. The random numbers generated for the behaviour are generated by the game engine itself.

### Prototype Two

Prototype Two will be the first prototype that will feature the interface. This prototype will have basic interaction between the game engine and the interface. This is to test if communication is possible between the two. This prototype features the same environment and bot as prototype one. The bot does not have the wander behaviour and will be static throughout. The new feature is the button on the screen in the game. When this is clicked it sends a message to the interface and the interface will respond to this. A simple counter to keep track of how many times the button was clicked was chosen as the author needed to test how variables were stored on the server. The button will send the message to the server, the server will increase the current counter then it will return he value to the game, which is then printed into the console.

### Prototype Three

Prototype three is a mixture of the two previous prototypes. The bot in the environment will use the wander behaviour again but this time the random values will be generated with the interface. The range will remain the same but the interface will be providing this data. The main reason behind this prototype was to test the interface to see if it can keep up with the game engines frame rate. In the game engine the rate is 30 frames per second. Therefore the interface must receive this function call, process the data and return a value before the next frame happens.

### Prototype Four

This prototype will use an artificial neural network to control the bot within the environment. The ANN will be stored within the interface and it will be run in synchronisation with the game engine.

#### Level One

Level one will feature the bot learning through the ANN to follow another bot using the wander behaviour. This shows that the interface and ANN combination can be used co-operatively between bots. The evaluation of this is how long the bot faces the player.

#### Level Two

Level two demonstrates that the ANN can be used competitively against another bot. The environment is full of objects that the bot must collect. The behaviour of the bot again is controlled by an ANN. The difference is that the evaluation of the neural network is dependent upon how many items that the bot picks up.

### Final Product

The final product will be the interface and the tutorial explained below. The interface must be well documented as it will/may be used by developers in the future. Therefore they must know all the features that it offers and what it can/cannot do. Therefore an API or document must be written to explain these details to developers.

## Tutorial

Since this project was aimed to aid developers the author decided to create a tutorial on how to do the basics in with this tool. The tutorial shows the reader all the necessary steps to get the interface working with the game engine. This includes setting up the server, connecting the game objects to the server and describes the basics of the interface and how to expand upon it. The tutorial will appear in the appendix.

## Synchronisation

Keeping everything synchronised is a key part of this project. Making sure that things don’t overlap themselves is desperately needed. Therefore a method of delaying the next function call was implemented. This could be used to stop animations overlapping or getting the neural network to wait until everything has been set up correctly.

# Results

## Prototype One

Prototype one was a full success. The wander behaviour was correctly written with correct parameters.

## Prototype Two

Prototype two was also a success as the server game could contact the server during run time.

## Prototype Three

Prototype three was a success. There was a slight change of plans when I could not get a second server up. One server would act as the interface, the other being the code that should run. The two servers should communicate with each other and should return a product to the game. But this was difficult to achieve. Therefore the code was integrated into the interface.

## Prototype Four

Prototype four was very difficult to achieve. Similarly to prototype three the original plan was to have a separate server holding the code that deals with the neural network, and that the interface would contact this server and receive data, which will then be passed back to the game engine. This was not achieved. Therefore a solution was to have the code integrated into the interface. This removes the need for a second server. This is not an ideal solution but it worked for the time being.

### Level one

Level one proved to be difficult for the ANN to be trained. The bot has three sensors that read data from certain angles in front of it. This data is then passed onto the ANN, this data serves as the input of the neural network. The bots behaviours are based upon its perception of the environment. In order to train the network into getting the correct behaviour a method of evaluating the ANN is needed. This is called the fitness function. The desired behaviour was for the bot to follow the other bot in the environment. The more the middle sensor was targeting the target indicated that it was facing the target. Therefore the fitness will be dependent upon how much the sensor read the target in the given run time. This did not appear to work. After a number of generations training the network it appeared not to obtain the desired behaviour. Therefore the bots movement was removed. The bot was to stay still during training. This approach never worked either. Therefore something else needed to be taken into account for the fitness. The distance between the two bots served as a good item to take into account. The closer the bot is the better the fitness. Therefore the fitness function was altered to also incorporate the distance between the two bots. The fitness function contained both the distance and the reading from the middle sensor.

### Level Two

Level two features an ANN that should collect all of the items in the environment. As with the level two the code for the neural network aimed to be in its own server. But due to difficulties connecting two servers the code ended up being inside the interface. This ANN like any other ANN needed a fitness function in order to be trained. The fitness for this is relatively simple. The fitness is dependent upon the amount of collectables it collects in the given time. The collectables reset during each iteration of the training, but in random positions.

## Balancing the neural network

Balancing the neural network required a lot of time and effort. Since there is no golden rule for them you have to use your judgement.

Setting up the architexture of the neural network was the first step. This involved selecting how many layers of hidden nodes there are, how many hidden nodes there will be in a layer. Also selecting what goes into the neural network and what should come out. This is all things that involved trial an error rather than having a set solution.

For the first level the first experiment was using a ANN with a single layer hidden layer containing five nodes. The input nodes took the data from the raycasts that are taken from the game. Originally the output node would feed out just the rotation for the bot to take. This gave poor results; therefore an extra output node was added to the architecture. This extra node would control how much the bot was to rotate. This extra node gave significantly better results.

Figure 4 Shown below

Show fitness results of single output node.

Level two again worked better with the new two output node architecture. Since the first level didn’t succeed with a single output, the second level didn’t even test the one node architecture.

Time taken is another step that was needed to be balanced. If the time given for each chromosome, in each generation, was large then the experiment would have taken a long time to finish, also overlearning could have happened. This is where the ANN learns too much and then can’t learn anything else. It gets too focus on one certain thing rather than learning the whole thing.

# Discussion

# Conclussion

## Future work