Flexible Artificial Intelligence API

For Torque Game Engine

A Thesis Presented to

The Faculty of the Computer Science Program

California State University Channel Islands

In (Partial) Fulfillment

Of the Requirements for the Degree

Masters of Science in Computer Science

By

Dustin Stevens-Baier

December 2008

APPROVED FOR THE COMPUTER SCIENCE PROGRAM

_____

Advisor: Dr. Andrzej Bieszczad Date

_____

Advisor: Dr. William Wolfe Date

_____

Advisor: Dr. Ron Rieger Date

APPROVED FOR THE UNIVERSITY

_____

Dr. Gary A. Berg Date

Flexible Artificial Intelligence API

For Torque Game Engine

By
Dustin Stevens-Baier

Computer Science Program
California State University Channel Islands

Abstract

Artificial Intelligence in Computer Games is a rapidly growing field. The thesis presents a flexible Artificial Intelligence API. This API allows developers to setup Artificial Intelligence quickly and cleanly. The API is integrated into the Torque Game Engine, and is tested by setting up a blackboard system and a Genetic Algorithm. Furthermore the Genetic Algorithm is tested using both single agents and teams of agents.

Acknowledgements

I would like to thank my wife for her endless support and hours of proofreading. I would also like to thank Dr. Andrzej Bieszczad for his guidance and help pointing me in the right direction. Finally, I would like to thank my friends and family for putting up with my long hours and lack of availability.

Table of Contents

Table of Figures

Chapter 1 - Introduction

I. Introduction

This thesis presents a flexible Artificial Intelligence API. This API allows developers to setup Artificial Intelligence quickly and cleanly for a variety of agents and circumstances. The API was created as an extension to the Torque game engine distributed by garage games.

After reviewing the literature and through my observations of Game AI it has become apparent that group behavior is relatively uncharted territory when it comes to learning algorithms. There are communication techniques like blackboard architecture and command hierarchy as well as concepts like scripting that currently control group behavior. However, there appears to be little in the way of learning algorithms and groups of agents. As a result this thesis will focus on creating a blackboard extension so that a group of agents can communicate.

Using this addition this project will combine offline learning, training that occurs during the development of the game, using Genetic Algorithms and the Blackboard Architecture allowing the agents to communicate. This addition will hopefully allow developers to coordinate group artificial intelligence agents using genetic algorithms and the flexible API. The hypothesis is that, by using the API and offline training we will improve Game AI development in two areas. First, we will have faster development in terms of man hours since the API will allow for easy setup. Second, we will have less predictable group behavior than if we had used a state machine. The algorithm will hopefully produce an entire population that can be used by the game engine when the game is being played online.

II. Introduction to Artificial Intelligence in Games

Artificial Intelligence in games (Game AI) has been around since computer games were first created in the late 1960s early 1970s, albeit in a simplified role. AI is used in computer games to accomplish a wide variety of things, including path finding, agent behavior, decision making and animation selection. Recently, Game AI has become an ever expanding field with more and more programmers becoming solely dedicated to creating Game AI from the beginning of a game's development.

As computation and storage space limitations have decreased, Game AI has continued to expand. AI in games was once considered an after-thought. Now, however,

it is playing a major role in the most successful commercial games. Computer graphics used to use so much processing power that there was little left for the AI calculations. Now that we have separate GPUs and our CPUs have even more power, Game AI is able to use more resources. Ultimately, end users, the people who play the games, want believable, realistic behavior that is challenging but, ultimately, allows them to win. As game developers have realized this, artificial intelligence in games has developed into its own area of expertise.

In order to understand the development of AI, it is necessary to understand the distinction between Academic AI and Game AI. In Academic AI, we are trying to understand the nature of the human brain and then use this understanding to solve real-world problems, whereas Game AI is all about the enjoyment of the end user. The success of Game AI is not measured in wins and losses. For example, we cannot simply make AI that wins every time. This would not be a good experience for the player. The goal, therefore, is to incorporate some of these Academic AI ideas and add them into Game AI while keeping, or increasing, the resemblance of the computer agents to end users.

Recent Game AI incorporates Academic AI techniques more frequently into both the development of the game and into the game play itself [Rabin 2004]. These techniques have allowed computer controlled agents to more closely resemble real end users. This resemblance leads to better game play and ultimately end user satisfaction. Overly-predictable game agents or wildly unpredictable agents lead to end user confusion and frustration.

A Genetic Algorithm is a learning technique that is sometimes used to create the most effective and realistic agents. This technique comes from the process of natural selection, essentially mimicking the process of evolution in an effort to produce a close to optimal solution. Some games already use Genetic Algorithms in an effort to have agents learn behavior, one such example is *Black & White*.

III. History of Game AI

When the first computer games came out in the 1960s and early 1970s there wasn't much behind the artificial intelligence. The games of this time period required two players or had no enemy agents. This allowed the developers to avoid having to worry about Game AI for the most part. One such game to illustrate this concept is *Pong*. *Pong* was created in 1972 by Nolan Bushnell. Pong was a simple game that succeeded because it allowed the youth of that generation to compete electronically [Morrison 2005].

Game AI gradually progressed as the popularity of video games increased. In the 1970s and early 1980s, games began to add enemy agents. Most of the video games at this time simply had preprogrammed patterns to follow. One of the most popular games was *Space Invaders* which used very complex patterns, for its time. *Space Invaders* was

so popular that Japan had a coin shortage and had to increase its supply of yen [Morrison 2005]. Another popular game was *Pac-man* which had different patterns for each of the Ghosts movement. This also included a handful of very simple random decision making to make the behavior less predictable [Morrison 2005].

In the late 1980s and early 1990s, the first state machines started to really develop. This led to the first Real Time Strategy (RTS) games which needed elaborate path finding for many agents. It also led to the first, First Person Shooter (FPS) games which have more complex state machines and much improved randomized behavior.

In 1990s and early 2000s, we saw the introduction of the first neural nets, genetic algorithms and heavy scripting. We also saw the first game that highlighted artificial intelligence as its selling point in *Black & White. Black & White* was built entirely around the idea of training the user's creature. The user could train the creature to be force of good or evil depending on whether the user punished or rewarded the creature for its behavior [Schwab 2004].

IV. Remaining Chapters

In the second chapter, we will analyze some of the different artificial intelligence techniques being used in the game industry today. We will inspect how learning algorithm's are used with individual agents in games. We will also take a look at the how the gaming industry deals with group agent behavior. Some basic AI techniques will also be discussed in this chapter.

In the third chapter, we will discuss genetic algorithms and the advantages and disadvantages of using this particular learning algorithm. We will look into the different elements that go into a genetic learning algorithm and how one might be able to fine tune it to fit different game genres.

The fourth chapter will discuss how we set up the game from a visual and game engine standpoint. We will discuss the environment being used, and how the enemy agents and the player were set up in Torque.

In the fifth chapter, we will begin to discuss the API that was created, specifically the Blackboard System API that we created for this project. We will also discuss how the blackboard architecture is used and why it useful in our system.

The sixth chapter will be a look at the Genetic Algorithm API which will also encompass how the data is learned, the different variables needed for the genetic algorithm to work and how the data is output so the developer knows what agents have been developed and will look specifically at how a genetic algorithm can be implemented in our system and the different ways it can be modified.

The seventh and final chapter will include a summary of the project's results. This chapter will also compare and contrast different learning techniques in games with our Genetic Algorithm. What remains unsolved and what could be added to further enhance the project will also be discussed.

V. Key Terms

Game AI - Artificial intelligence in video games.

Agent – An object in the game that is controlled by the AI system.

Ranged enemy - Game term for agent that attacks from a distance

Ranged damage – The amount of damage that the agent contributes from a distance.

Melee enemy - Game term for agent that attacks from close by.

Melee damage - The amount of damage that the agent contributes from up close.

Attack Speed – How fast the agent attacks.

Aggressiveness – How likely the agent is to attack and how quickly it starts attacking.

Favorite type of attack – which type of attack the agent prefers, melee or ranged.

Finite State Machines (FSMs) - A model of behavior that consists of a finite set of states, transitions between these states and actions.

Fuzzy State Machine (FuSMs) - Allow the possibility of being in more than one state at time unlike Finite State Machines

Message Based Systems – Instead of an agent checking for changes in information every time. Other agents will tell the agent that its information has changed

Scripting - The behaviors of the group are written out beforehand and any reactions are dealt with as a State Machine.

Neural Networks - Find solutions by using a method grounded in how the brain works both functionally, and organizationally. They pattern match and predict trends in data.

Flocking - A technique for coordinating a group of agents' movement.

Genetic Algorithms – Use principles of evolution to find solutions. Uses many variations of crossover and mutation along with a fitness function to make a determination of what should be reproduced. There are a variety of ways to choose the

best data from this group a roulette wheel, stochastic universal selection, and tournament selection.

Chromosomes – A string of bits that represents an individual in the system.

Parent – The individuals that pass their chromosomes onto the next generation.

Child - The individuals who receive chromosomes when creating the next generation**.**

Crossover – The process of combining the two sets of genes from one parent to create two children.

Population – The group of individuals that is used in each generation.

Generation – One pass through the genetic algorithm.

Single-point Crossover – A form of crossover where a single crossover point is defined. This point is then used to create the two children.

Multi-point Crossover – A form of crossover where every bit has a chance of crossover.

Mutation – A process where each bit of the child has a chance of being modified.

Fitness proportional selection method (roulette wheel) – A method of selecting parents where the probability of a parents' chromosome being selected for reproduction is proportional to its fitness score.

Hamming Distance – A form of measuring fitness where you add the number of similar bits between the parent chromosomes and the desired chromosome.

Grafting – The technique of taking a node of pone parent and connecting it to the node of another parent.

Elitism – A technique were a certain percentage of parents are passed directly onto their children.

Blackboard Architecture – a method that allows groups of agents to communicate based of a physical black board and classroom system.

Blackboard – The area in which the readable and writeable data for the Blackboard architecture system is stored.

Knowledge Source – The components that read and write to the blackboard.

Arbiter – The control that decides what the knowledge sources will do based off the information on the blackboard.

Chapter 2 - Field Overview

There are quite a few different techniques that are used when creating Game AI. These include, but are not limited to, finite-state machines, fuzzy state-machines, scripting systems, neural networks and flocking. All of these technologies are in current use to different degrees and all can be used to deal with the issue of group AI in games.

I. Finite-State Machines

The most used technique when creating Game AI is the Finite-State Machine (FSM). A FSM is a rule-based system in which a finite number of states are connected in a directed graph by transitions between states [Schwab 2008]. There are multiple reasons that this technique is used so frequently. One major reason is time constraints. In some situations, developers are trying to finish their projects so they choose a technique that is easy to debug, understand and is comfortable. Another reason to choose a FSM is because of the breadth of problems it can deal with, from dialogue selection to individual agent states [Schwab 2008].

For example, a simple state machine could consist of an agent who has four different states: (1) fighting (2) running (3) walking and (4) fleeing. During these different states, a corresponding animation is chosen. When the agent is fighting, the fighting animation is chosen. When the agent is running, the running animation is chosen. When the agent is walking, the walking animation is chosen and, finally, when the agent is fleeing, the fleeing animation is chosen. Implementing this is usually done with a “case” statement or by using “if else” statements. A simple FSM illustration can be seen in Figure 1.

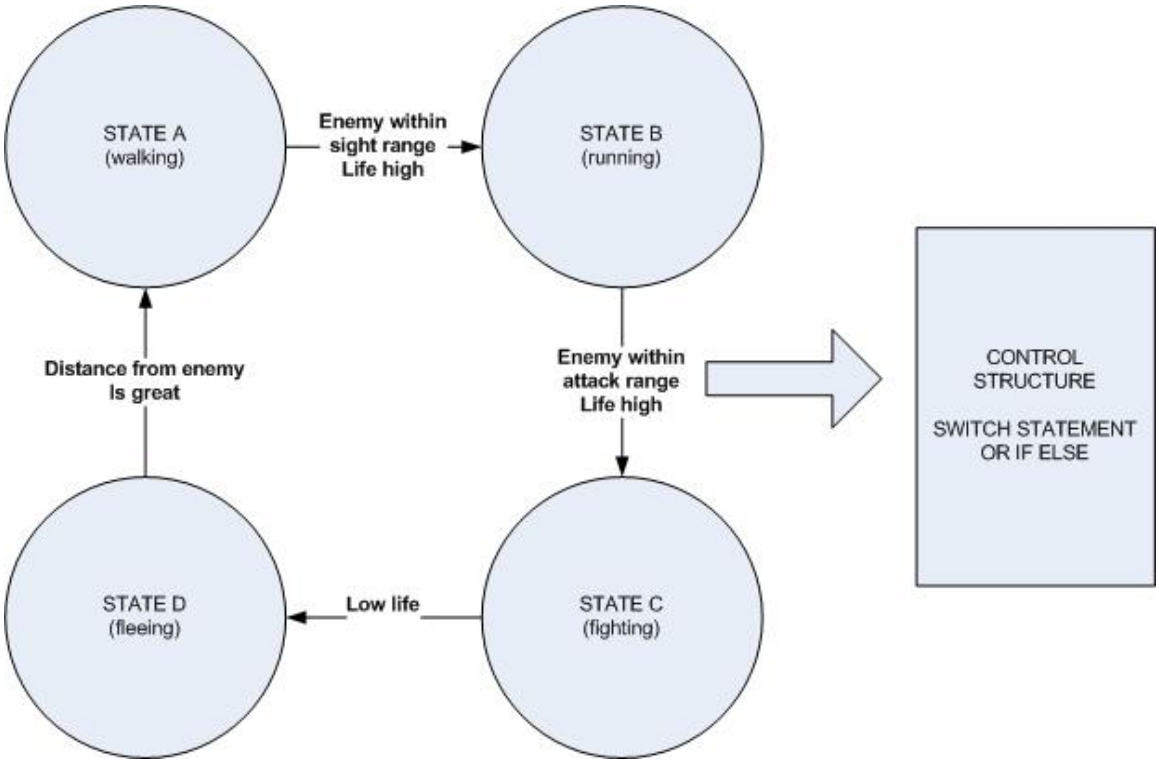

Figure 1- Traditional Finite State Machine with four states

FSMs are most commonly used to deal with single agents. However, this technique has been used when dealing with multiple agents, both in cooperative and competing situations. A drawback to dealing with FSMs is that, when too many states and agents are introduced, it can be difficult to keep track of what state every agent is in. This is especially true if the development was done in multiple parts. FSMs tend to respond poorly to multiple iterations of development. Each layer of development adds more specialized states and can lead to increasingly complex state diagrams [Schwab 2008].

Another negative when working with FSMs is the oscillation problem. Oscillation occurs when an agent can switch back and forth between two states [Schwab 2008]. This leads to a player being trapped between two states, for example between attacking and fleeing. The result of this problem is frustrating game play, either being annoyed that the animations of the agent don't work appropriately or being unable to complete certain parts of the game.

FSMs also tend to be very predictable. They only respond to changes in states and, as such, users can exploit their repetitive nature as well as possible missing states [Rabin 2002]. This can be a negative in some situations, for instance, if a certain football play always works when you pass to a specific player. However, this predictability can also be built into the game play. For example, the user might need to avoid being seen and, therefore, needs to time the agent's path in order to run around a corner.

II. Fuzzy State Machines

Fuzzy-State Machines (FuSM) are state machines that use the concept of fuzzy logic to represent degrees of membership in different states that range between 0 and 1 [Schwab 2008]. An example would be an agent who is both in a fleeing state and attacking state just too different degrees, which usually are represented by a number between 0 and 1. This allows a gradual progression from state to state. The agent can start by only attacking when his attacking activation level is greater than a predetermined number. This agent then begins fleeing while still attacking when the fleeing activation number is greater than a predetermined number. Finally, the attack activation level could be dwarfed by the fleeing, which would result in the agent concentrating only on fleeing. One can also have degrees of being in one state. Another example would be an agent going from walking to jogging to running all based on the same state activation level. For this reason, FuSMs are useful for systems that can be in more than one state at a time and in one state to different degrees [Schwab 2008].

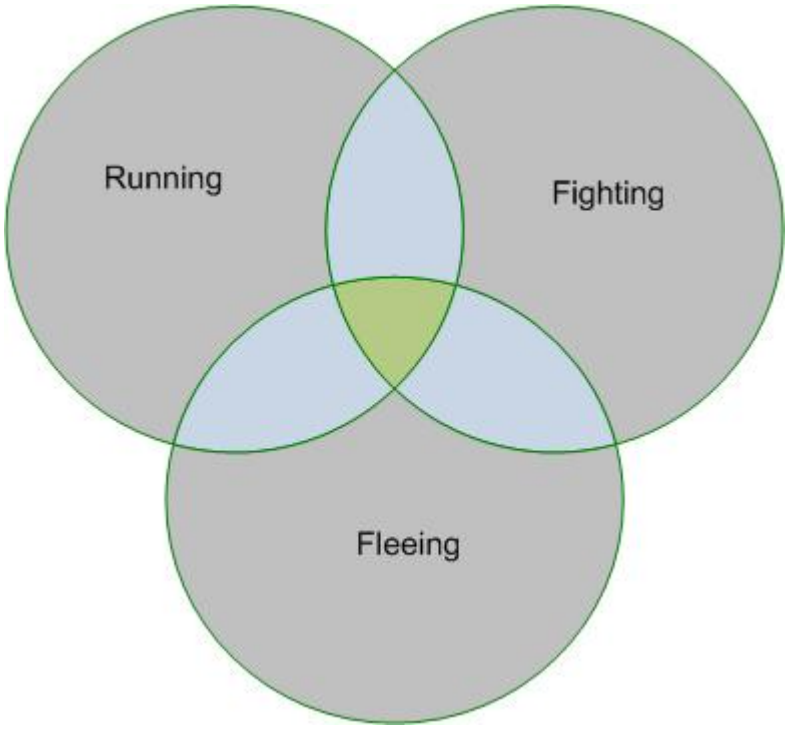

Figure 2- Venn Diagram of a FuSM shows that an agent can be running, fleeing, and fighting all at the same time.

The ability to be in more than one state at a time, while using a FuSM, removes the oscillation issue discussed earlier. One no longer flips back and forth between two states. Instead, the agents transition from one state to both and then to one. This allows for smoother game play and animations.

The concept of FuSMs has also been used to deal with group behavior. For instance, if a group needs to decide if it attacks or hides, it can actually use all the different members to make this decision. Let's say that a group has five members and we use an uneven number for the simplicity of breaking a tie. If three agents want to attack and two agents want to flee, the group will say that it has a .4 activation level for fleeing (2 fleeing agents/5 total agents) and a .6 activation level for attacking (3 fighting agents /5 total agents). This allows the group as a unit to make decisions based on its ability to be in more than one state at a time. After attacking, one of the agents could decide it wants to flee, which subsequently changes the activation level of the entire group.

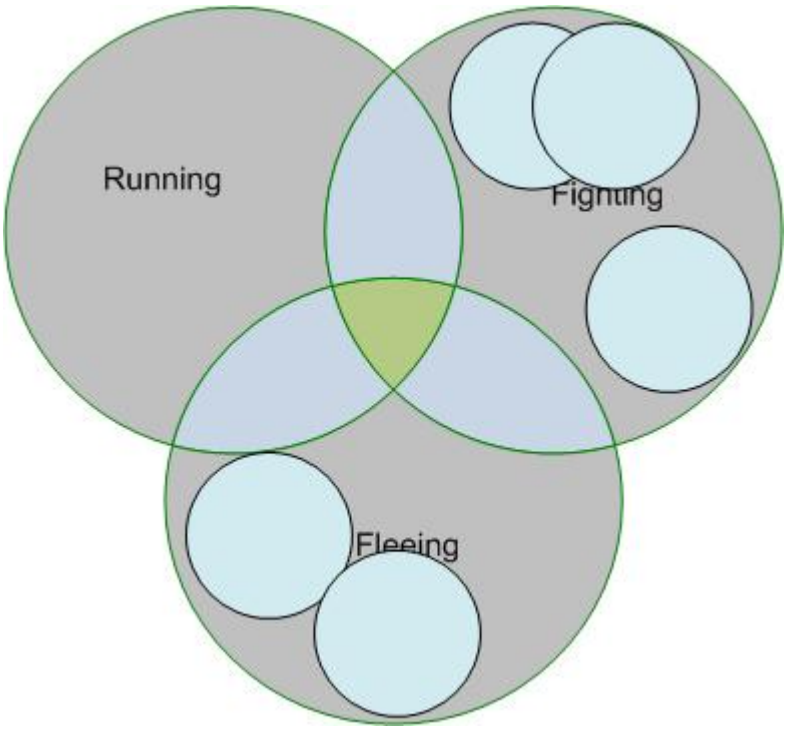

Figure 3- A Venn diagram that shows a group with three agents in the fighting state and two agents in the fleeing state

Even though the individual agents might have finite states of either fleeing or attacking, the group has fuzzy states. This allows the group to exist in more than one state at a time. This solution can be very life-like as groups of people tend to have their own opinions on how to do things but can work together for a common goal.

FuSMs are being used more frequently now that the predictability of FSMs is becoming undesirable [Schwab 2008]. This technique requires more planning since one can be in more than one state at a time, which leads to harder-to-read code and diagrams. A once very simple diagram map of a FSM now has to take into account which states can exit together and which cannot.

The implementation of a FuSM can be even easier than a FSM because transitions aren't as difficult to deal with [Schwab 2008]. The transitions take care of themselves

since one can be in more than one state at a time. Another benefit is that FuSMs are easier to extend than FSMs, given that one can simply add a state that can exist along with the other states [Schwab 2008]. This is much easier than having to create a new state and to find the right place for it and the correct transitions. One can also control degrees of behavior by increasing the activation levels. For instance, if one wants one group of agents to hide more frequently, one can increase the hiding activation levels.

III. Scripting

Another technique that is used a lot when creating Game AI is Scripting. This technique involves using a simplified programming language to build AI elements, logic, or behaviors [Schwab 2008]. Scripting is used to allow others, besides the programmers, to help in the creation of Game AI. This can be very time-intensive, as the scripting language has to be created or one has to allow the use of a language that is already available, like Lua a scripting language that is used in a lot of games on the market. One has to weigh the benefits of allowing others to assist in the creation of Game AI and the time it takes to set up this feature.

Scripting can also be used when dealing with group behavior in that it allows the developer to very tightly control the way individuals within the group respond to certain actions. For instance, the boss agent can wait until all his minions die before attacking the user. A problem with scripting, however, is that it can lead to repetitive behavior. If a user has to play a specific section a few times and every time the section contains the exact same agent behavior, then it can become boring and even annoying if the user cannot get past this section.

One significant benefit of scripting is the ability to rapid prototype [Schwab 2008]. Level designers and other developers can use the scripting language to get a demo up and running very quickly. Scripting allows modifications to AI behavior to occur at a faster pace [Schwab 2008]. Since we are making modifications at a higher level than the code itself, we don't necessarily have to link and compile again. This means developers can easily make changes and test drive their changes.

A drawback of scripting is that any scripting language is going to be slower to execute than a program that is compiled into machine code. This means we don't want to run path finding algorithms for an entire army in a scripting language. Another issue with scripting is debugging. Debugging tools are not very good for scripting languages because they tend to lack the full capability contained within the development tools for other types of languages [Rabin 2002]. From a user's perspective, the biggest issue is the look and feel of heavily scripted games. When a game is heavily scripted, it doesn't allow for different actions and the users tend to get tired of the exact same actions and agent behavior.

IV. Neural Networks

A Neural Network is a learning technique based on the architecture of neural interconnections in animal brains and nervous systems [Schwab 2008]. The concept of a neural network is an attempt to learn things from real life and apply them to AI. This is very similar to the goals of genetic algorithms and artificial immune systems [Schwab 2008]. There are three different layers to a back propagation neural network, one of many Neural Networks that has been used in games. These are: (1) input layer (2) hidden layer and (3) output layer. The input layer is where outside data enters the neural network. The hidden layer is where the data is stored and on which operations are performed. Finally, the output layer is where the desired data is located.

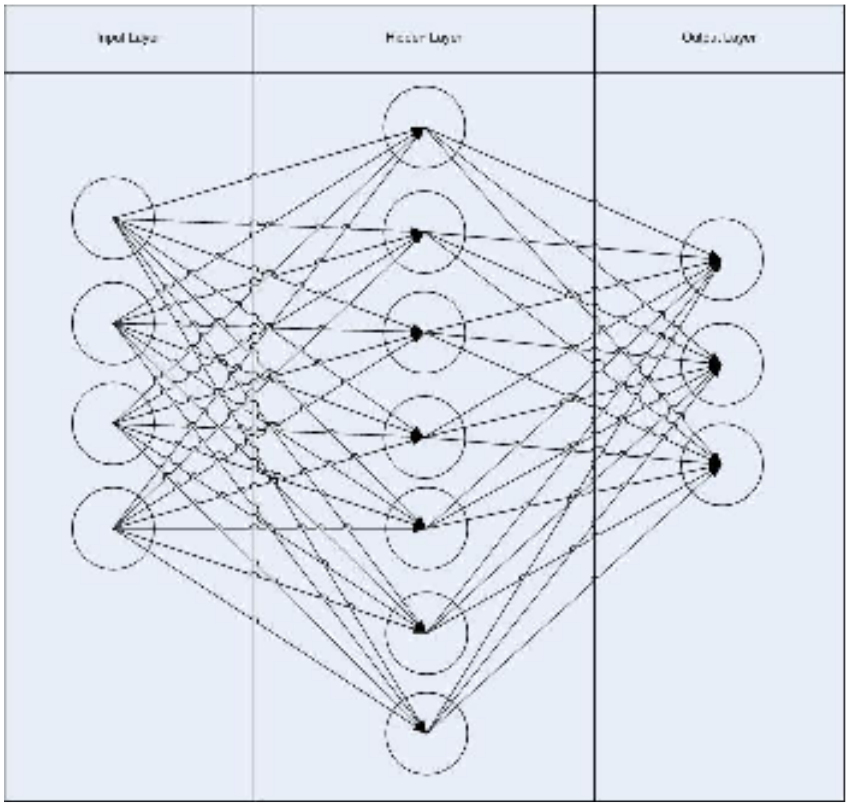

Figure 4- An example of a simple neural network with four input nodes and three output nodes

Neural Networks can conceivably be used to train groups to work together while offline. This can be done in the same way that individual agents are trained. One could input all the different agents' values into a network and look for one output to represent the group decision. Neural Networks have been used in Game AI, usually to train individual agents to react, and can be helpful in reducing the number of man hours needed for trial and error. Additionally, neural networks allow for tweaking some of the training data and then letting the system start training, which can free up the developer to do something else while waiting for the data to be trained.

A major problem, however, with neural networks is figuring out how to train them [Castor 2006]. Another issue is that a neural network is hard to debug since the data inside the neural network is often impossible to comprehend. Since neural networks can take lots of training data it is difficult to make this concept work in a game that is live. One such possibility might be a Massive Multiplayer Online Role-Playing Game (MMORPG) where lots of players are online all the time.

V. Flocking, Herding and Schools

Flocking is one of the few techniques discussed that is never used to deal with just one agent but, instead, deals solely with groups of agents. Flocking is a technique for coordinated movement that allows the groups of agents to move in life-like fashion. This is typically represented as herds, schools of fish and flocks of birds. This can be an incredibly complex task if trying to control a group of agents individually. One would have to keep track of every agent in relation to all other agents as well as objects in the gaming environment.

Flocking is accomplished through three simple steering behaviors [Castor 2006]. The first is collision avoidance and separation, which is the individual agent's attempt to avoid hitting another agent in the group [Castor 2006].

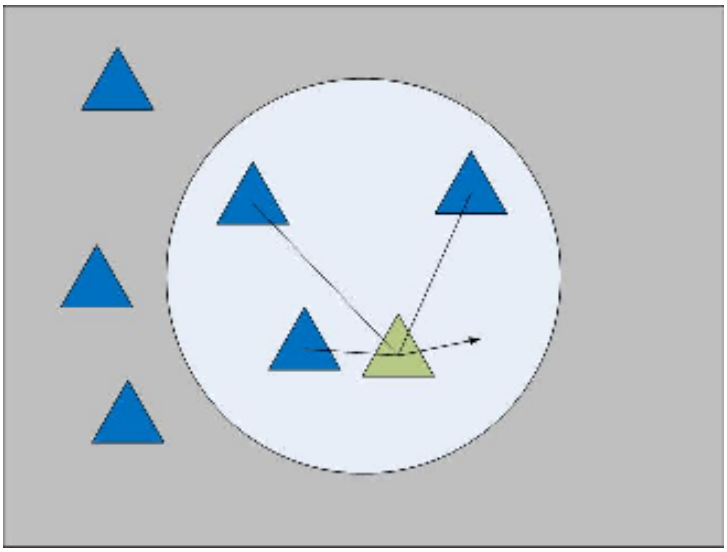

Figure 5- Example of collision avoidance in a flocking system

The second steering behavior is velocity matching and alignment, which is the individual agent's attempts to match the other agents in direction and speed [Castor 2006].

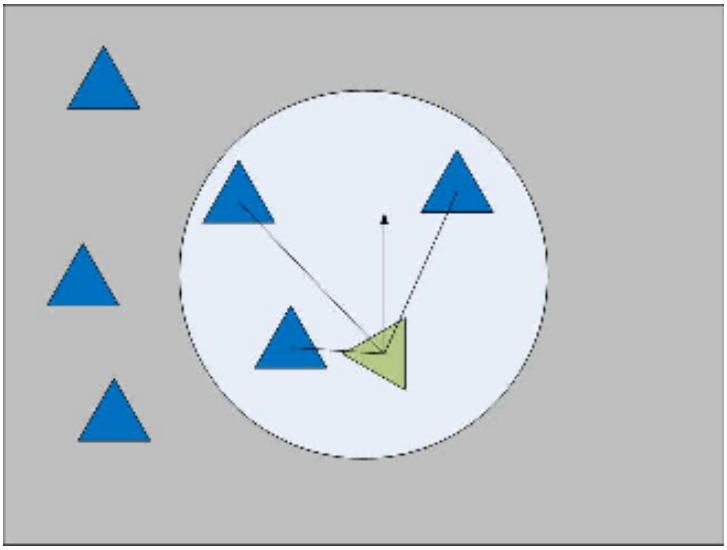

Figure 6- Velocity matching in a flocking system

Finally, the last element in the steering behavior is centering, which consists of the individual agent trying to stay near the average position of all of the agents surrounding it [Castor 2006].

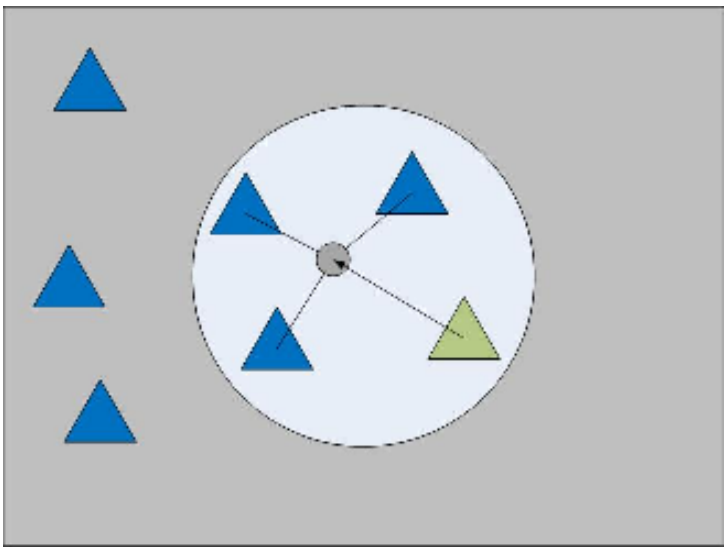

Figure 7- Centering in a flocking system

One can see how flocking plays an important role in group behavior. Although the initial concept of flocking does not deal with anything besides motion, it can easily be adapted to deal with other things, such as level of agitation for the herd. For example, a deer that gets upset when a lion can be seen by one of the herd would indicate this concept. Flocking could be brought over to a military group, as well in which the entire group could be affected by lack of food or gradual running out of ammunition.

Flocking is most commonly used in feature films, although it has started to make its way into more and more gaming applications. Films use flocking to show stampedes, large battle sequences and to make sure they look life-like without having to coordinate thousands of animals or men. For animated movies, the creation of a few creatures and then the replication of them over and over allows the flocking system to control their behavior instead of having to draw or pose multiple agents everywhere repeatedly.

Chapter 3 - Genetic Algorithms

I. Genetic Algorithms Overview

A Genetic Algorithm is a type of evolutionary algorithm that models the evolutionary process and uses a vocabulary borrowed from natural genetics [Castor 2006]. This process usually has the steps illustrated below in figure 3.1. The first step is the setup of the population which is usually done by randomly creating individuals, which are represented by chromosomes. Then, these chromosomes are evaluated in order to choose the parents, individuals that pass their chromosomes on to the next generation. After this, the parents are used to create the next generation, also called children. This is usually done through genetic crossover but can sometimes use grafting. Then, the children are sent through the mutation process and finally they replace the old generation.

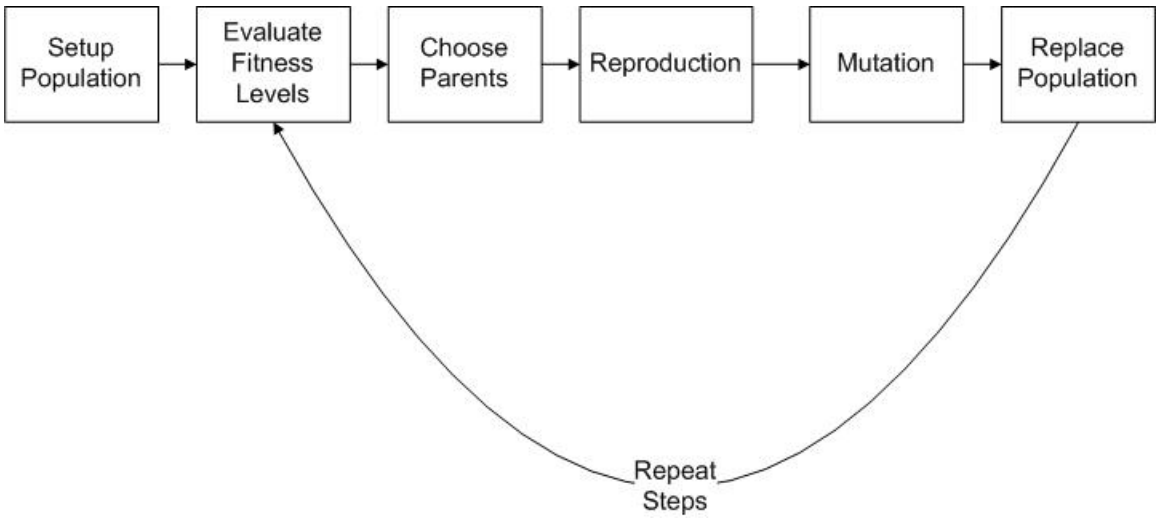

Figure 8- Standard flow chart of a Genetic Algorithm

To choose which parents reproduce, one usually uses the survival of the fittest method, which occurs when a species as a whole tends to choose the fittest parents, or the ones with the best genes, to reproduce. One such technique that is used to accomplish this is called fitness proportional selection method, or the roulette wheel selection method. In this method, the probability of selecting a chromosome is directly proportional to its fitness [Castor 2006]. This concept essentially maps each chromosome to a portion of a roulette wheel. The size of each section is proportional to that chromosome's fitness value, which is a rating for good the chromosome is for reproduction. Spinning the wheel one time for each parent needed produces the next generation [Castor 2006].

There are variations of this concept. One such variation allows one to select the same individual many times over. To fix this, one can put a limitation on the number of times a selection can be chosen before it is removed as an option from the wheel.

Another adaptation to the selection method is elitism. In this method, a certain percentage of the parents with the highest level of fitness have their chromosomes directly passed down to the children. This method helps to ensure the fittest genes make it to the next generation.

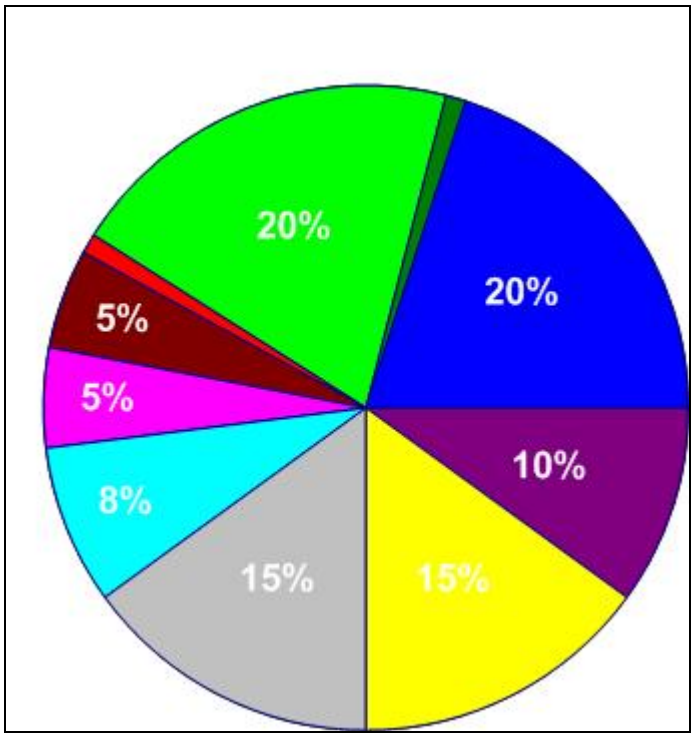

Figure 9- Roulette Wheel with fitness being the measurement for percentage of the wheel.

One of the most commonly used measures for determining fitness is the Hamming distance. This method consists of adding the number of similar bits in the parent chromosome as compared to the desired chromosome. For example, if one has 6 out of 8 bits matching, the number of different bits is 2 and the Hamming distance is 6. In this case, the ideal individual would have a fitness score of 8. This concept could be expanded upon in two ways (1) by having individuals consist of three sets of chromosomes or (2) taking each individual on the team and adding up the scores so that a fitness level for the entire group could be used.

Once the parents have been decided, on the next decision to make is which bits get passed on to which children. Crossover is the process of combining the two sets of genes from the parents to create two children. This is done by defining a crossover point, usually halfway through the chromosome, and then taking the first half of one of the parents and the second half of the other parent to create the first child. Then, the second half of the first parent and the first half of the second parent is used to create the second child. This concept is called single point crossover [Rabin 2004]. Since we don't know which genes make the parent the fittest, when performing crossover, it is possible to create children that are less fit than their parents [Castor 2006].

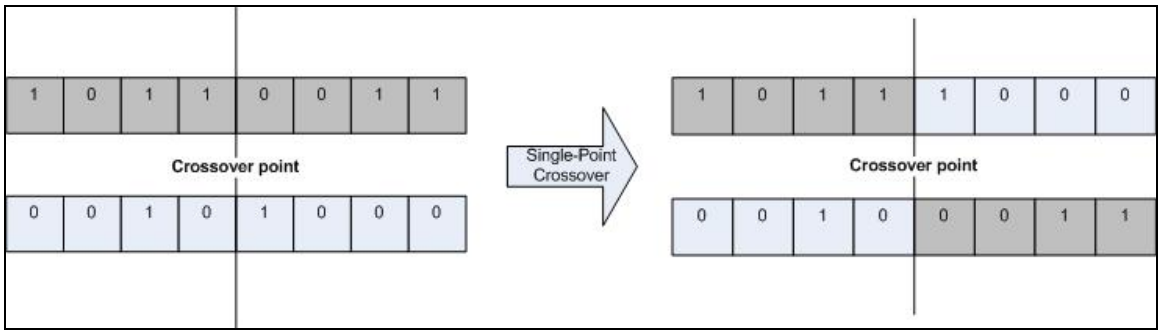

Figure 10- example of single point crossover

Single point crossover is not the only way to deal with reproduction. Multiple point crossover occurs when one child could have the first third and last third of one parent and the second third of the other parent. In this situation the second child would have the opposite genes. Additionally, in single point crossover, the crossover point doesn't necessarily have to be fixed. For example, the point could be located halfway for one set of parents and then one third of the way for the next set of parents. There can also be a certain percentage chance for a crossover at each point.

Another portion of the Genetic Algorithm is mutation. To illustrate, consider that a chromosome is represented by series eight bits, either one or zero. When a mutation occurs, one of these bits is chosen randomly and changes from zero to one or from one to zero. This can be accomplished in a couple of ways. First, we could force a percentage of chromosomes to mutate and then randomly select which bits get changed. The second way, the most common way, every bit in a chromosome has a very small chance of mutation [Schwab 2004].

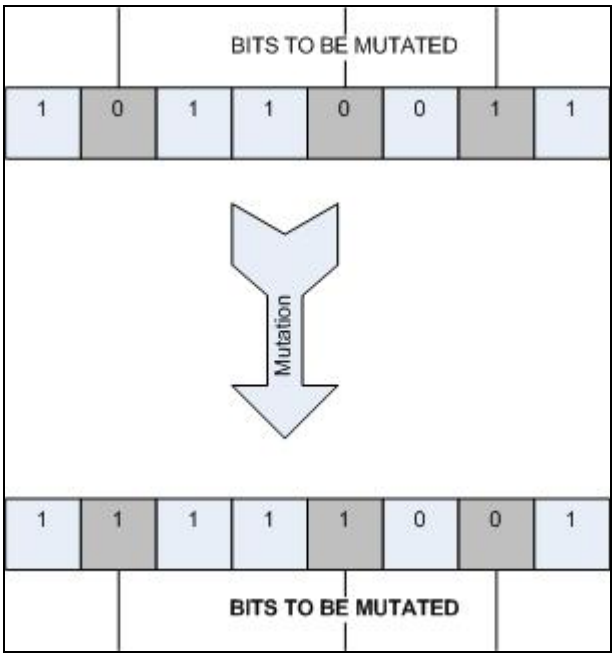

Figure 11- Example of mutation

A practical application of this concept is in pattern recognition. One can make a matrix representation of a particular number or letter and then teach a group of individuals to recognize the character by performing the above mentioned steps [Castro 2006]. After enough iterations, the Genetic Algorithm, can successfully detect the character.

Genetic Algorithms have been adapted to fit games as well. The concept has broken into two parts: the set up of the behavior and the usage of the learned data. The setup of the behavior is typically done offline; this involves setting up the controls for how the agents will behave. The second part is the usage of the learned data; this usually does not happen online and typically does not involve any modification of the data or behaviors. [Sims 1994].

One major reason why Genetic Algorithms are almost exclusively done offline is because they are computationally expensive. This is because in order to get the most desired results, one has to run many iterations of a large population. Another reason for offline work is the random factor. Since Genetic Algorithms have some element of randomness when selecting the next generation, this can cause the algorithm to find local maximums instead of global maximums. Since most of the time developers don't want this behavior in a released game this is why most of the behavior modification and learning is done offline. If the algorithm is to be run online, it needs boundaries to reign in the random factor and unwanted behavior.

II. Field Overview of Genetic Algorithms

In order to more fully understand Genetic Algorithms, it is necessary to understand their history and how they have been used in gaming. To accomplish this we will look at [Sims 1994 3D], [Sims 1994 Virtual],[Grand 1996], [Grand 1997] and [Lassabe 2007].

A. Karl Sims

Karl Sims described a system for creating virtual creatures in his white paper, *Evolving Virtual Creatures*. Sims discusses the entire process of his virtual creatures from morphologies to evolution. However, we will focus on his description of creature evolution since this is more directly related to our project. Sims first created an initial population of genotypes, usually by randomly creating them. Sims arranged for genotypes with a fitness rating under a certain threshold would be replaced with randomly-generated genotypes. This was accomplished by first changing the fitness rating of these genotypes to zero, which eliminated possible further reproduction in his system [Sims 1994]. The next step in his evolutionary process was to create the next generation. First, he used asexual reproduction to determine the amount of the

population that will directly survive to the next generation; a process sometimes called elitism. After elitism he then moved onto the mating part of his process.

Sims used two mating techniques to create his offspring. First, he used the crossover technique described earlier to create offspring. He uses both single point crossover and multiple point crossover to create different combinations of offspring. When creating the offspring using this method, he started by copying one parent. Then, when the algorithm calls for a crossover, the other parent's node is taken along with the node connections. He deals with the concept of uneven nodes by randomly reassigning node connections that no longer have nodes.

Sims also used a technique he calls grafting. This technique takes a node of one parent and connects it to the node of another parent. He chose the first parent, randomly took one of the connections, and randomly assigned it a node of the other parent. The now unconnected nodes of each parent are then removed. When finalizing the reproduction, he uses the three techniques with this percentage, (1) 40% asexual, (2) 30% crossovers, and (3) 30% grafting [Sims 1994].

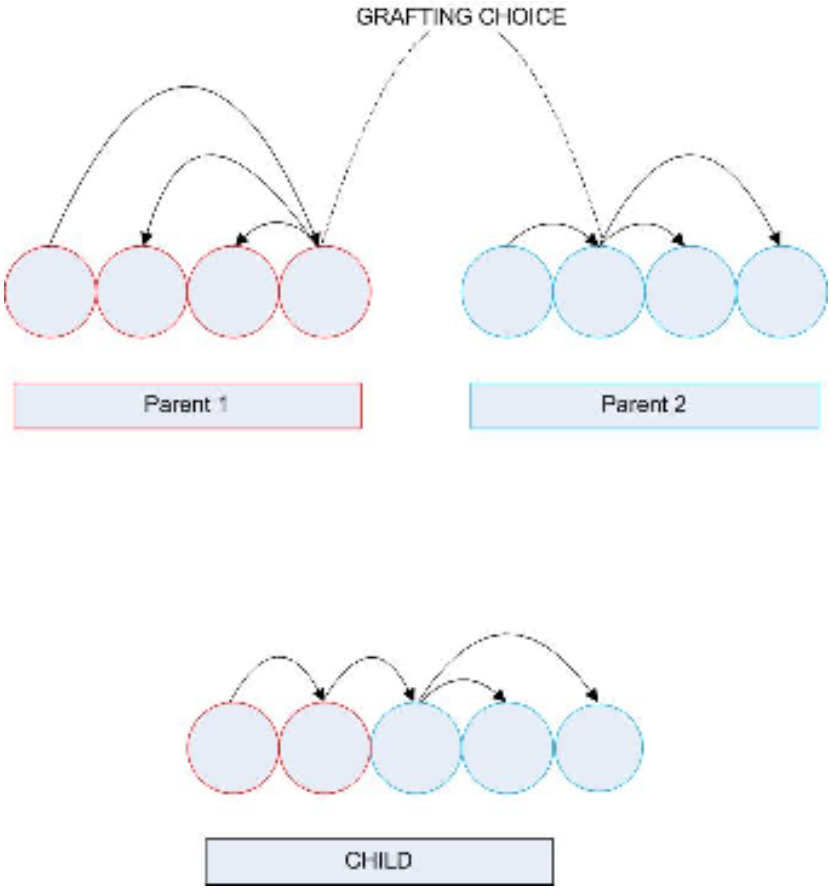

Figure 12- Example of grafting

B. Creatures

The computer game, *Creatures*, provides an example of Genetic algorithms being used in computer games. The game, *Creatures,* sold over half a million units and is widely thought of as a breakthrough in artificial life [Grand 1997]. These accolades come from a lot of different areas of Artificial Intelligence present in the game. *Creatures*, the computer game, uses a genetic algorithm to determine the evolution of the creatures. The new creatures are created based off the parents' genetic traits. There are a few steps taken in order to successfully control the online learning.

The first step was identifying the properties of the creatures that could be mutated. This includes types of dendrites, the expression used to calculate the state value, the threshold for calculating the output and the relaxation state [Grand 1996]. These limits are set up to control what properties of the parent can be mutated when creating the child. This process is necessary in order to ensure the system will not crash or drastically change to unwanted behavior.

The second step was to make the crossover and mutation processes safe. The developers needed to make sure they prevented the creature's brain from stopping after crossover or mutation. To do this, the developers needed to explicitly control what operations (omission, duplication, and mutation) can be performed on the genes representing the creature's brain [Grand 1996]." This control allows the developers use the genetic algorithms online instead of being limited to offline learning.

Figure 13- snapshot of creatures game

C. A New Step for Artificial Creatures

A natural progression of evolving creatures is to combine complex creatures with complex environments. Lassabe, Luga and Duthen propose such a system in their white paper, *A New Step for Artificial Creatures*. The goal of their paper is to show how complex, evolved creatures are able to cope with different situations in a complex environment [Lassabe 2007]. [Lassabe 2007] use similar creature morphologies to those used by Karl Sims. [Lassabe 2007] allowed their morphology to be dynamic because they wanted to keep the ability of generating various life forms inside their ecosystem. To evaluate the fitness of their creatures, they used different options. First, they wanted them to learn to walk so they created a function that measures distance over time. Second, they tried to encourage a direction so they tried the distance on the x-axis over time. After walking, they moved onto more complex environments, such as jumping over trenches and climbing up stairs [Lassabe 2007].

The group agent cooperation created by [Lassabe 2007] is their final test. In this test, the fitness is measured by the success the agents have in pushing a block in the right direction. The block has been defined as too big for just one agent to push. Their example is simplistic in nature as they use just two agents, one cloned from the other. Since the bigger agents are stronger, the groups of two that have a larger size are always the best at this test [Rabin 2002].

Chapter 4 - Game and API Design

I. Game and API Design

To setup the AI API that was the goal of the project we first had to select a game engine to add our API to. To do this we researched a few different engines, Ogre, which had a large community for support, and Torque which had some extra functionality like built in scripting language. We ultimately choose Torque because of the built in scripting language that allowed us to demonstrate the use of the API a little easier. We could show how the engine components can be completely separated from the game components.

After choosing an engine to work with we decided on a battle simulation that would involve a simulated player versus three agents communicating together. We decided on a simulated player so that we could have as controlled an environment as possible. Next, we decided on having the enemy team be made up of three agents. We chose three agents because one agent does not constitute a team and two agents can communicate directly with each other pretty easily, but three agents becomes more difficult, and we wanted to demonstrate our Blackboard API. The process of creating the blackboard necessitated adding interface functions and control variables so the developer can set up a blackboard object and access it appropriately when necessary. Developers will need to add missions, delete missions, move agents to different missions etc…

The design step was to setup our Genetic Algorithm specifications. The same idea that we applied to the Blackboard API we applied to the Genetic Algorithm API. The extension needs to allow developers the flexibility to control variables like number of generations, mutation rate, and crossover rate. Also the developers need to be able to easily change the way the genetic algorithm's success is measured. We decided that all of this would be done through defines and or get/set function calls. After making these design decisions the next step was to start setting up our game using the Torque game engine.

II. Torque Setup

The first step in setting up our game was to set up the map using Torque's development tools. We created a map that consisted of a few mountains, grass, sand and some trees. The idea was to create an area that has some objects that could get in the way as well allow for different terrain for the battles. A snapshot of the map is seen in Figure 14.

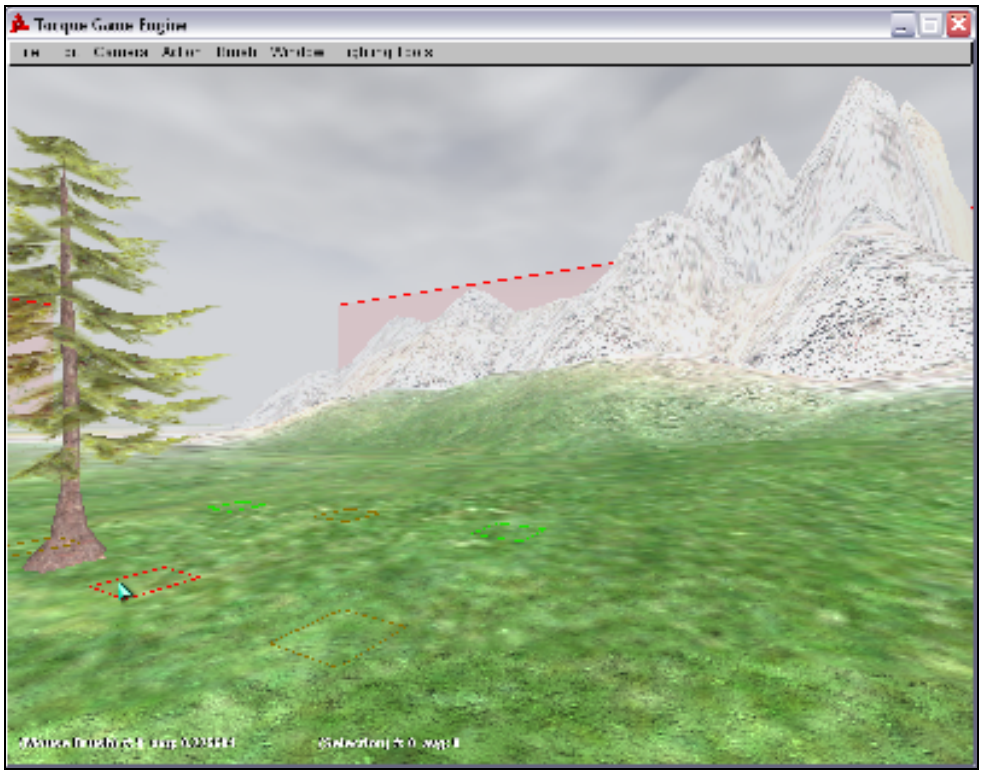

Figure 14 - Example of game map being set up, inside Torque environment.

A few different types of trees were placed around the primary battle area. The trees' role in the game was to present the agents with something to avoid when in battle with the player. The Torque game engine dealt with the collision detection and the necessary AI adjustments for the players to avoid such objects. A picture of the trees in the Torque World Editor seen in Figure 15.

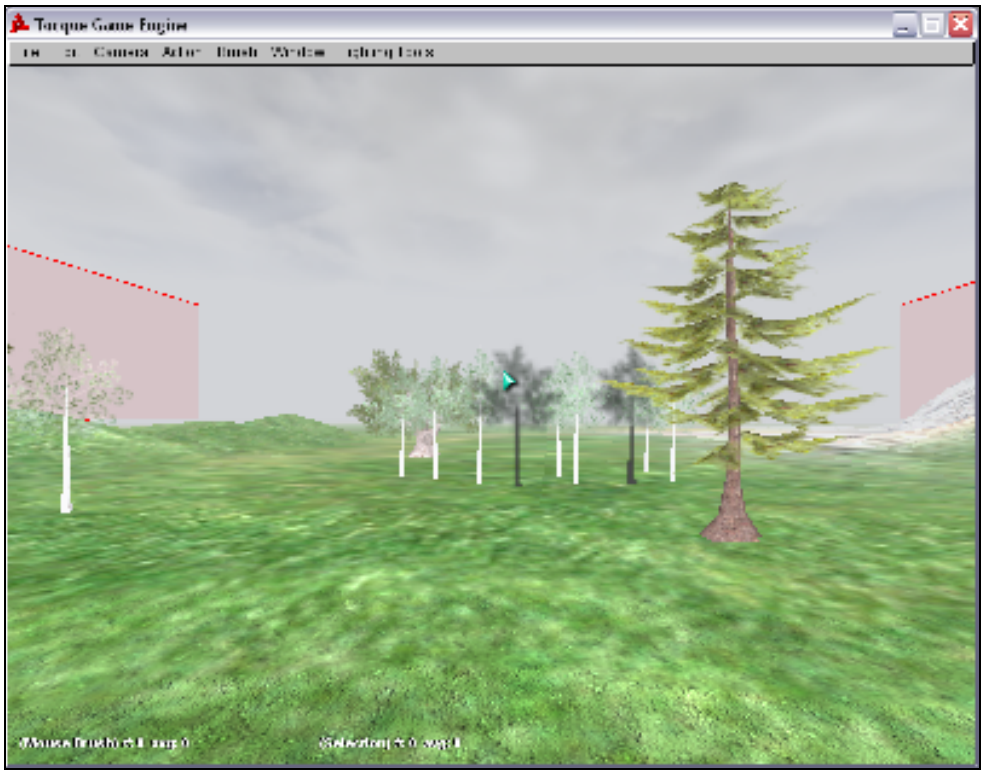

Figure 15 - An example of trees in the torque environment.

III. Simulated Player Setup

The player was placed facing the location where the enemy agents would start. This was right on the fringe of the grassy area facing the sandy area. The reason was to give the battle a sizeable amount of room to fight in and around the environment. A bird's eye view of the player facing the direction where the enemy agents will spawn is seen in Figure 16.

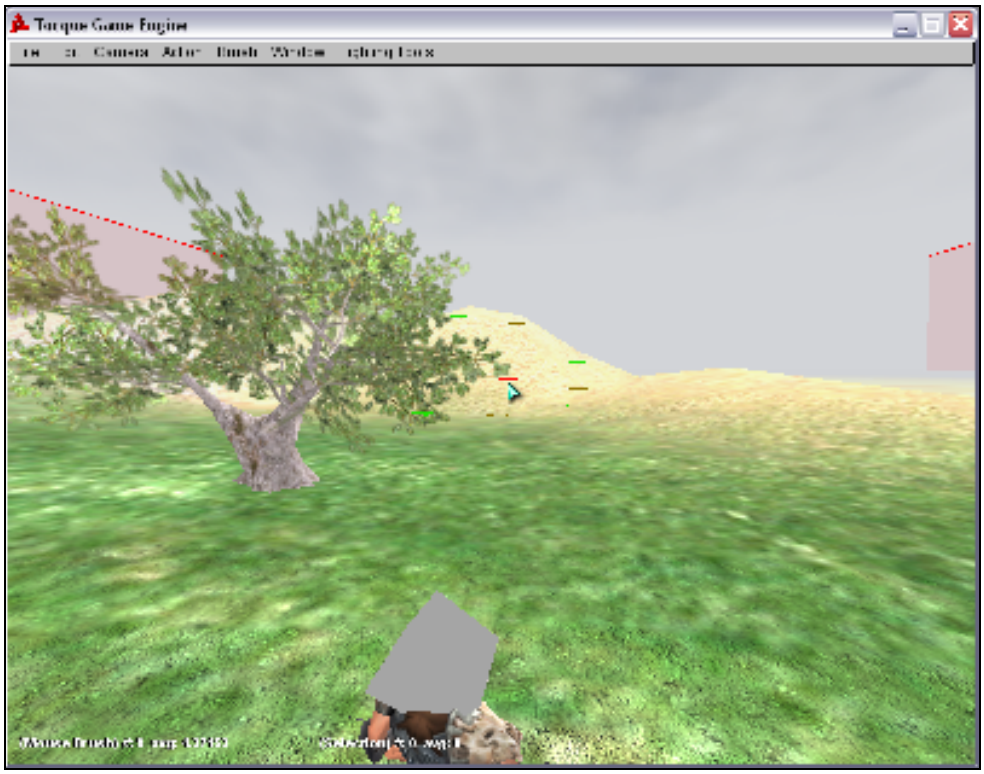

Figure 16 - An example of the view from the player facing the area the enemy agents will be.

With regard to perspective the player has a few different views it can use. Although not necessary for the project it is interesting to use these views to watch the battles. The first view is in first person which is demonstrated Figure 17. The direction of the player is shown with a plus sign which is where the player is aiming with its sword. This allows the developer to see where the player is facing because, when it is simulated, the player cannot be controlled by the mouse.

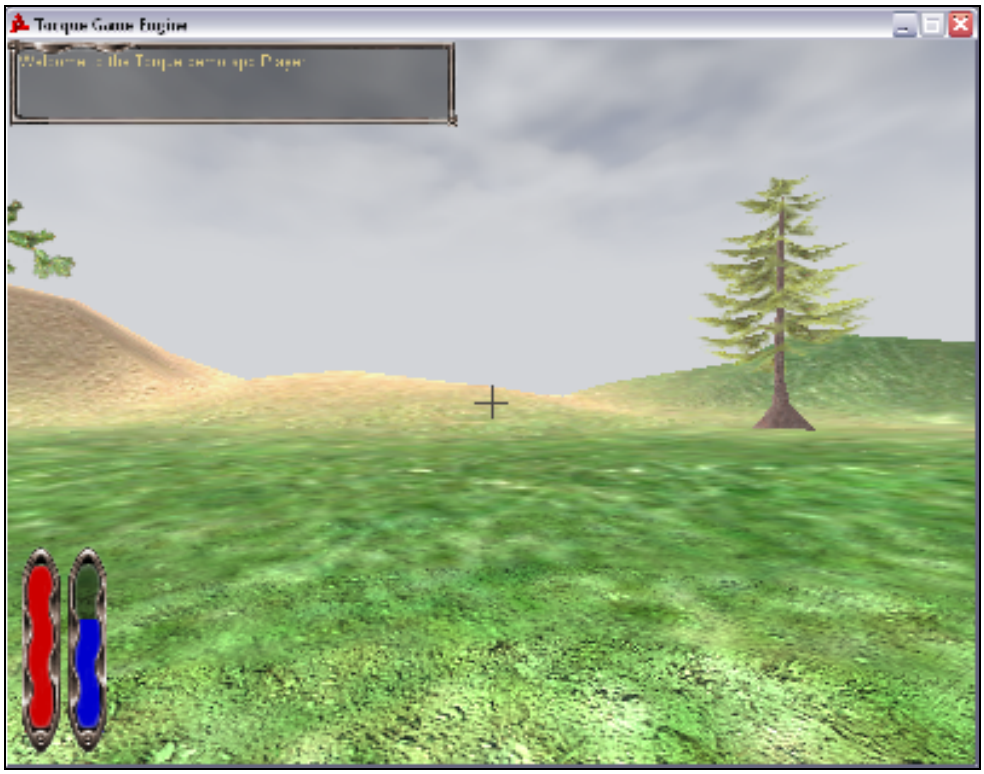

Figure 17- First person view from player.

Another view is third person which does not have the aim location given that the whole player is visible as is in Figure 18. This view gives a little better visual of what is happening in the battles and is used as the default view. This view was also used during the debugging process to best view the attacking between simulated player and agents.

Figure 18- An example of the simulated player.

The player that can be controlled by the user or simulated, as is done for most of the project, only has a sword to attack with. The player also looks just like the enemies, as the same model was used for both. A front view of the player with the sword is shown in Figure 19.

Figure 19- Front view of the simulated player with a sword.

IV. Head-Up Display (HUD) Setup

In order to keep track of vital information necessary for the player to play the game the developers usually use a *head-up display*, or *HUD*. A HUD is typically a transparent display that is on top of the game so the user can be presented with all of the vital data without obstructing their view. In our project we used two bars in the HUD one was a red bar that showed the amount of life of the player. The second was a blue bar that displayed the amount of energy the player has left. An example of the HUD can be seen in Figure 20:

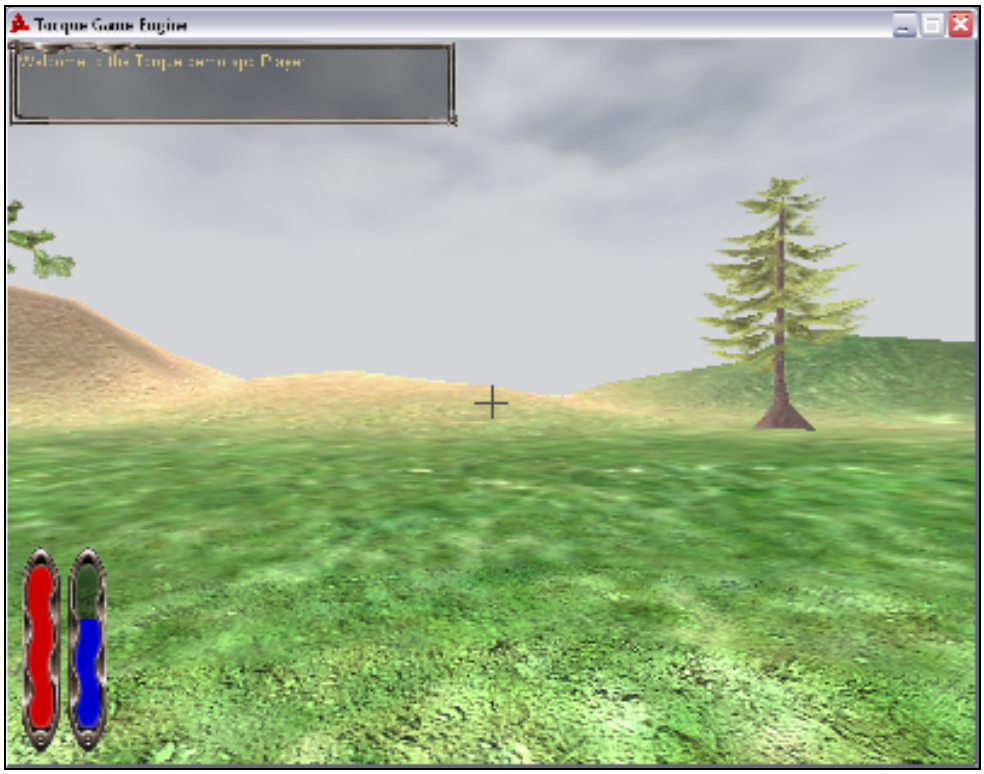

Figure 20 - an example of the HUD used to display player life and energy.

By turning the Genetic Algorithm HUD on, the developer can see the chromosome data broken up into variables and displayed so the developer can see the variables change as the different battles take place in the top right-hand-corner of the screen. An example of the Genetic Algorithm HUD is shown in Figure 21.

Figure 21- Genetic Algorithm HUD

This data gets updated every time the next three agents are created. The developer can also see the current generation and the current chromosome in the top left-hand-corner of the screen. This data is only visible when the genetic algorithm HUD is turned on.

V. Enemy Agent Setup

In addition to attacking with a sword, the enemy agents can also attack with a crossbow. These agents use the chromosomes and Genetic Algorithm talked about in chapter 6 to make decisions on what weapon to use. The three agents individually decide An example of an attacking enemy agent with a crossbow can be seen in Figure 22.

Figure 22- An enemy agent attacking with a crossbow.

When the battle begins, the three agents start at about an equal distance from the player. The agents start facing away from the player, although they scan the area for players to attack depending on their parameters. If they decide to attack, a decision that is made based off of parameters that are discussed later in the paper, they start towards the player. The default setting of each agent is HOLD, a state where they wait and scan for the player. Once an agent sees the player, a mission will be added to the blackboard to ATTACK. The blackboard does not allow the same mission to be posted more than once. Each agent can ask the blackboard for specific information through function calls. Every time through the agent calls its own application routine. This routine looks at the

different missions and sees if there is one for which it thinks it should apply. Once the agent applies for the mission the blackboard will either accept or deny the application, depending on how many agents are currently on the mission. Below in Figure 23 is an example of three nearby agents scanning the environment and awaiting orders they currently start off with the skills: (1) HOLD, (2) ATTACK_BOW, (3) ATTACK_SWORD, and (4) DEFEND. An example of the agents in their starting positions is seen below:

Figure 23- Players before battle begins.

When the enemy agents see the player, they start attacking. In our project, we also set up a simulated player that starts attacking. The player attacks the nearest agent first until it dies and then looks for the next nearest, etc. An example of this can be seen in Figure 24, as the player targets the enemy and the enemy starts running at the player.

Figure 24 - Enemy agent entering combat

Once combat has begun and the enemies have started damaging the player, a few things start to occur. First, the player's life is reduced, which can be monitored by red bar. Second, the screen briefly flashes red to notify the player that they have been hurt. Finally, if the damage is extensive enough, the player dies. This amount of damage that it takes to be killed is a parameter that can be adjusted during the course of the project. An example of damage occurring right before the player dies can be seen in Figure 25.

Figure 25 -Note the lack of the red bar signifies that the simulated player is near death

Chapter 5 - Blackboard Architecture

I. Blackboard System

Since this project involves teams of agents, there needs to be some form of communication system that coordinates the activity of the agents. For the purposes of this project, we are only coordinating three agents' actions. However, the system was designed to allow more agents as desired. Blackboard Architecture was chosen to accomplish this goal for the project due to the ease in which it can be implemented, combined with the power it provides the developer in player coordination. This addition allows developers to use the API we developed when adding blackboard system to their game. The Blackboard Architecture is our first extension of the Torque game environment. The Blackboard Architecture is done using C++ and is on the game engine side, while the API calls are made using Torque script and are game specific code.

The blackboard approach is based off the action of using a physical blackboard as a means to brainstorm and problem solve [Rabin 2002]. The blackboard system allows a shared space for agents to communicate. Blackboard Architecture consists of three main components: (1) blackboard, (2) knowledge sources, (3) arbiter. A blackboard is the area in which readable and writeable data is stored and accessed. This also controls the organization of this data, whether the data is grouped by type, time asserted, or no grouping. Knowledge sources are the components that can read and write to the blackboard [Rabin 2002]. These are the agents that are cooperating to try and solve a problem. In our project, these knowledge sources are the AIGuard agents that the player is fighting. The a*rbiter* is the control that decides what the knowledge sources will do based off the information on the blackboard.

The process of creating the blackboard system involved creating modules on the engine side of Torque and then adding interface functions and variables so that the developer could set up a blackboard object and access it appropriately when necessary.

There are four modules that were used to create the blackboard system as well as some interface functions that were added to the agent's module. Each of these modules corresponds to the four components used in the blackboard system: (1) skill type which is defined as the type of tasks that an agent can participate in, (2) missions which are defined as tasks that require certain skill types to participate,[1] (3) agents which includes how agents create missions and skill types, (4) blackboard control which controls each agent's current mission and what missions are active and/or closed.

---

[1] The concept of breaking the blackboard system up into skills, missions, and applications was derived from Chp 7.1 of A.I. Game Programming Wisdom.

II. Skill Type Module

The first step in using the blackboard system is to create skill types for the agents. This allows any developer to set his own set of skills for each agent. In the case of our project, we had the following skilltypes: (1) Holding, (2) Defense, (3) Attackbow, and (4) Attacksword. All agents were able to hold or defend and each agent had one attack skill, but some had both.

The steps demonstrated below enable the ability for the agent (represented as %obj in the code) to participate in missions that require the skill type "Attackbow." When setting up the skill type, the agent needs the name of the skill type (Attackbow); it also needs the reference to the agent object (%obj) and the proficiency of that agent with this skill type (.8). The proficiency is later used when determining the best course of action for every agent by the blackboard control. The developer can use this when allowing agents to switch back and forth from different missions and skill types. An agent might be more proficient in attacking with a bow than with a sword. For the purposes of our project, we assumed the agents were equally efficient because our focus was on the Genetic Algorithm rather than on the different skill types and how it affects the communication system. The script example below shows how a developer will add a skill type. The example uses Torque script which is used by game developers to keep their game specific code separate from their game engine code. These are API calls to create a skill type in the case Attackbow which later will be used in to match agents with missions.

```
//agent now has a crossbow add the ability to attack with
a bow
%skillbow = new AISkillType();
MissionCleanup.add(%skillbow );
%skillbow.setAISkillType("Attackbow", %obj, .8);
%obj.addSkill(%skillbow);
```

Figure 26 - API calls that setup skill type.

III. Mission Module

The next step when using the blackboard system is to create and add a mission. In order to create a mission, the agent first has to determine what skill type will be necessary for the mission to be completed. For example, if the agent currently has a bow, he tries to create a mission that attacks with the bow. If the agent has a sword, he tries to create a mission that attacks with a sword. The endcase for the mission is either the agents dying or the player being killed. The intention of this project was to focus on the battle between the three agents and the player resulting in one group is defeat.

In the example below, one can see that we create a skill type the same way we do for an agent. This time, however, we also create a mission and we set the mission to have a certain skill type needed in order for the other agents to participate. Two other arguments are passed into the function when setting up the mission. The first argument is the number of agents that can participate in this mission. For this project, we used three as the number of agents because one agent is not a team and two agents can communicate directly with one another. On the maximum side we had time constraints so we could not test for larger and larger teams. The second argument is the priority of this mission. We chose .8 as the default priority of all missions because we are focusing on the growth of the agents in the Genetic Algorithm and, as such, did not create missions that have different priorites. This could be used by the developer to create missions that matter more than others. For example, an army attacking a castle could have two missions: (1) To kill the king with a very low priority to start and (2) To get inside the castle with a high priority to start. Initially, most of the agents put their attention towards getting inside the castle. Once the agents get inside, their attention would shift to the king because it is the next highest priority mission. An example of how to set up a mission using the API from game side is seen below.

```
//setup skillType first
%skilltype = new AISkillType();
MissionCleanup.add(%skilltype );
%skilltype.setAISkillType("Attacksword", %obj, .8);

//create new mission
%mission = new AIMission();
MissionCleanup.add(%mission);
%mission.setAIMission(%skilltype, 3, .8);
```

Figure 27 – Example of API being used to setup a mission with a skill type.

Once a mission has been set up by an agent, the mission needs to be added to the blackboard so other agents can see it and apply for it. Because each agent has a reference to the blackboard object on which they are a communicating, whenever that agent wants to add a mission, the agent (%obj) uses its reference to the blackboard to add a mission from the script side. This is illustrated below in another example of our API being used.

```
//add Mission to blackboard
%obj.blackboard.addMission(%mission);
```

Figure 28 – Example of adding a mission to the blackboard using API functions calls.

Adding the mission to the blackboard allows the agents to apply for what missions make the most sense for them. The blackboard arbiter then decides what missions to grant based on what makes the most sense for the group. In our project, the agents only want to change missions if the priority of a holding mission or a different attack mission is higher. The reason for this is we wanted the agents to focus on the battle and not risk having them take off to perform other tasks.

IV. Agent Interface

When interfacing with the blackboard system, every agent uses his reference to the blackboard control to apply for missions and to set actions. Every time the agent runs through its thinking function, the agent updates its current mission. The first step is to save the old action. Then, the agent calls its actionSelect function.[2] This function sets up the mission it will be on next. The agent then needs to see if it has a current mission. Otherwise, it does its default action, which is holding. This is illustrated in the script example below:

```
%prevaction=%obj.action;  //save the previous action

obj.actionSelect();     //apply for new missions everytime through

//if agent has been placed on a mission update action state
 if(%obj.isMissionActive())
    %obj.action = %obj.currentSkill.getSkillName();
 else
    %obj.action = "Holding";
```

Figure 29 – Example of actionSelect API function call from script side.

The actionSelect function is an API call to our extension inside the Torque Game engine it loops through finding the highest priority mission for the agent. The function then checks three things: (1) has the agent completed its current mission? (2) Is the agent's current mission's priority 0, which is the lowest priority? (3) Does the agent have a current mission? If (1) or (2) is true or if (3) is false then the agent is given the highest priority mission that the selection process determined earlier. This process is demonstrated in our game engine extension code below, which shows what happens when a developer has an actionSelect API call.

[2] The actionSelect function is a call to the engine which determines what mission makes the most sense for the agent to be on.

```
//Find the highest-priority relevant mission on the blackboard.
for(iter = skills.begin(); iter< skills.end(); iter++)
{
   relevantMissions = blackboard->getMissionsForSkillType(*iter);

   for (d=0; d < relevantMissions.size(); d++)
   {
      mission = relevantMissions[d];
      temp = calcPriority(mission, *iter);

      if (temp > highestPriority)
      {
        highestPriority = temp;
        highestPriorityMission = mission;
      }
   }
 }
```

Figure 30- Internal code inside actionSelect function call

V. Blackboard Module

The blackboard object consists of a map object which contains all the open missions and some interface functions. The interface functions allow the developer to add missions to the blackboard, get missions that fit a certain skill type, remove agents from missions and take the agent's applications for missions. When adding a mission as described above, the script interface calls the following engine function. The engine function is the extension to the engine that we created.

```
/*************************************************************
 *                          addMission
 *
 * Used by producers of missions to put it on the blackboard.
 * Only place on blackboard if missions not already created.
 * Insert call to map will not allow us to duplicate mission
 * numbers we can however have missions that require the same skill
 *************************************************************/
 void AIBlackBoard::addMission(AIMission* mission)
 {
      if(!missionIsPosted(mission))
      {
         //set Mission number
          mission->setMissionNumber(getTotalMissionNumber());
         //need make_pair becuase map stores values that are in a pair
         openMissions.insert(make_pair(mission->getMissionNumber(),
               mission));
      }

 }
```

Figure 31- addMission API function inside the Torque game engine

The engine checks to make sure that we don't already have this exact mission posted. Then, it adds the mission to the map that contains all active missions. This is called by the agents in all the situations the developer has set up to start missions. In our game, this is called when the agents see the human for the first time.

Every time through the AIManager cycle, the blackboard updates the mission assignments. This update is called from the script side but interfaces with an engine function that resides in the blackboard module. This function looks at all the requests for missions and grants them as long as they are not full. If the mission is full, the blackboard removes the mission from the blackboard and denies all requests. This is illustrated in the figure below.

```
void AIBlackBoard::updateMissionAssignments()
{
  map<int, AIMission*>::iterator it;

  //exit if we have no missions
  if(getTotalMissionNumber()<=0)
     return;

  for(it = openMissions.begin(); it!=openMissions.end(); )
  {
    AIMission* aimission =  dynamic_cast<AIMission*>((*it).second);

    //Remove the mission from the blackboard if it is complete.
    if(aimission->getMissionComplete())
      openMissions.erase(aimission->getMissionNumber());

    //Start granting applications
    AIApplication* curApp = aimission->applications;

    while(curApp!=NULL && !aimission->isFull())
    {
      //grant mission request
      curApp->aiguard->grantRequest();
      //set current mission
      curApp->aiguard->setCurrentMission(aimission);
      //add member to mission set
      aimission->addMember(curApp->aiguard);
      curApp = curApp->nextApplication;  //move onto next
application
    }

    //While we have more applications start denying applications
    while(curApp!=NULL)
    {
      curApp->aiguard->denyRequest();
      curApp = curApp->nextApplication;
    }

    //When the mission is full remove it from the blackboard
    //the blackboard ONLY contains open missions.
    //Also increment iterator one way or another
    if(aimission->isFull())
   {
      it = removeMissionFromBlackboard(aimission);
   }
   else
      it++;

   aimission->applications = NULL;
 }
```

Figure 32- A part of the Torque Game Engine extension for a Blackboard System.

Once the agent's mission has been switched from "mission pending" to "current mission," it will start to perform whatever action matches the mission criteria. If the agent dies, it will remove itself from the mission. This action forces the blackboard to clean up the mission map. First, the member gets removed from the mission list.

Second, if the mission is not full or completed, the function adds the mission back to the blackboard so that another agent can apply for the newly created vacancy. The remove mission function is shown below:

```
/*************************************************************
 *                    AIBlackBoard::removeFromMission
 *
 *  An AIGuard is requesting to be removed from a mission, either it
 *  has found a more important mission to attend to or it died.
 *  In this function I make the decision of whether or not we want
 *  to recycle the mission or just let it expire. The general idea
 *  is that as long the mission is still incomplete and is not full
 *  we repost it.
 *
 *************************************************************/
void AIBlackBoard::removeAgentFromMission(AIGuard* aiguard, AIMission*
mission)
{
  myMissions::iterator iter;
 //error checking
 if ((aiguard==NULL) || (mission==NULL) )
    return;

 mission->removeMember(aiguard);

 //make sure that mission is not full or complete before posting
 if (!mission->isFull() && !mission->getMissionComplete())
 {
      iter = openMissions.find(mission->getMissionNumber());
      if (iter!= openMissions.end())
      {
           addMission(mission);
      }
  }
 }
```

Figure 33- API function removeFromMission used on the script side to remove an agent from a Mission.

Using the blackboard architecture allows the developer to control group behavior relatively easily. By adding new skill types later, the developer can extend the functionality to include a hunter or gatherer agent, or an agent that does karate instead of using a sword. Using this framework also allows for an easy increase in priorities. If a group of agents is behaving as desired, an increase or decrease in the priorities can create the desired behavior. This functionality can also be used to allow agents to switch back and forth as certain missions become higher priority. An example of this could be if an agent is fighting one enemy but a higher priority enemy appears so the agent creates a new mission and applies for it. The blackboard arbiter then tells the agent to change its mission which results in the agent switching to attacking the new enemy.

Chapter 6 - Group Genetic Algorithm

I. Evolving Agents in a Group Environment

A. Genetic Algorithm Outline

We have chosen to use Genetic Algorithms to train the most effective groups of agents. This task will be split into two sections, (1) Game engine code which will be API functions that are created on the game engine side so they can be used by developers creating any similar game, and (2) game specific code which will be the API function calls used by developers to interface to their game to the game engine AI components.

The Genetic Algorithm we are using has seven steps. The first step is population setup, followed by evaluating the population with the fitness equation. After every member of the population has been given a fitness score, we go into choosing our parents that will create the next generation. From there, we use elitism and crossover to create our children. After all the children of the next generation have been created, we then pass each child through our mutation process. Once the next generation has been finalized, it replaces the previous generation and the entire process is repeated.

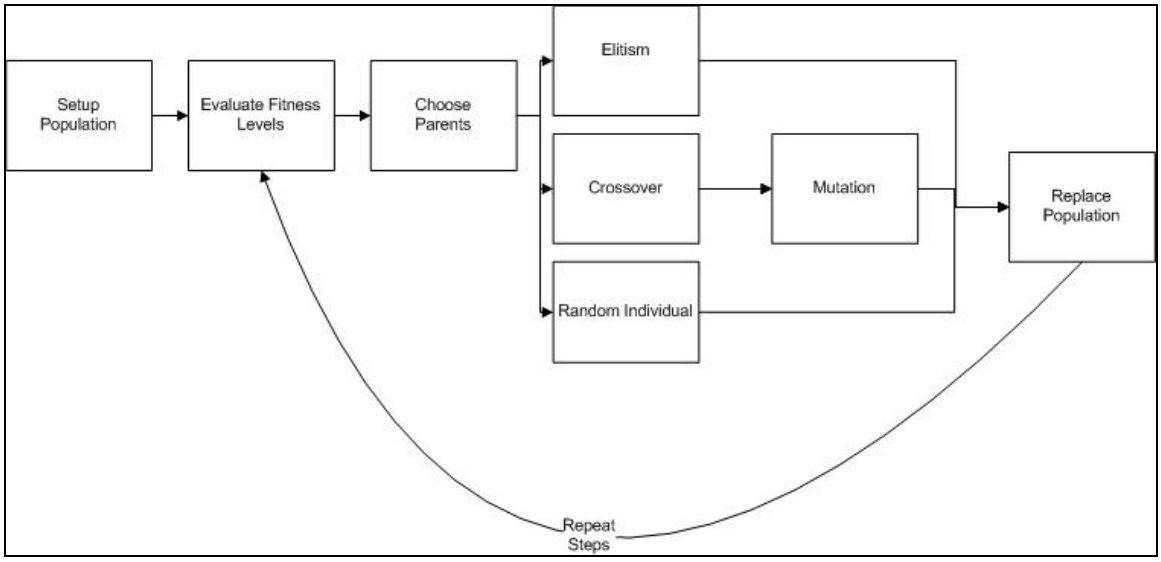

Figure 34- Flowchart of the genetic algorithm that was used in our project

B. Setup of Population

In the initial population setup, we are randomly creating a group of parents that will get our genetic algorithm started. The parents will each have the following chromosome:

1. Ranged Enemy	0 or 1	represented by one bit
2. Damage per ranged attack	1 to 15	represented by 4 bits
3. Melee Enemy	0 or 1	represented by one bit
4. Damage per melee attack	1 to 15	represented by 4 bits

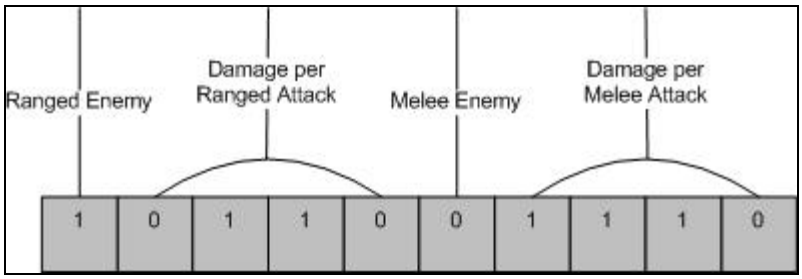

Figure 35- Part 1 of the Chromosome that represents every agent in our system.

The bit that symbolizes ranged enemy shows whether or not this agent can be a ranged enemy. The following four bits show the range of damage the agent can cause if the ranged enemy bit is a 1. The bit that symbolizes the melee enemy shows whether or not this agent can be a melee agent. The following four bits show the range of damage that the agent can cause if the melee enemy bit is a 1.

1. Attention Level (aggressiveness)	1 to 15	represented by 4 bits
2. Attack Speed	1 to 15	represented by 4 bits
3. Favorite type of attack	1 to 15	represented by 4 bits

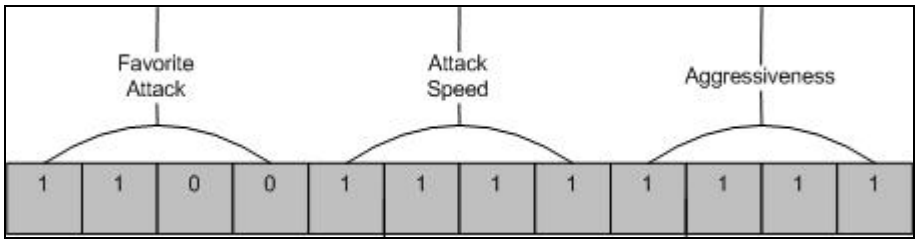

Figure 36- Part 2 of the Chromosome that represents every agent in our system.

The first four bits represent the range of the favorite type of attack this particular agent favors. If both Melee Enemy and Ranged Enemy bits are chosen, then this group of bits determines how often the agent favors one or the other type of attacks. The second four bits represent the range of attack speed that the agent uses. Attack speed is defined as how quickly the agent attacks after a previous attack. The final four bits of this chromosome represent the aggressiveness of the agent. Aggressiveness is defined as how likely the agent will start attacking as well as how likely it will start to flee.

To create the initial population, we loop through the entire population size, creating random chromosomes that are designed as stated above. The code sample below demonstrates this process further, which takes place in the Torque game engine extension we created for Genetic Algorithms.

```
/*****************************************************************
 *    AIGeneticAlgorithm::createInitialPopulation()
 *
 * Create as many random individuals as the population size
 * dictates
 *****************************************************************/
void AIGeneticAlgorithm::createInitialPopulation()
{

      for(S32 i=0; i < GA_POPULATION_SIZE; i++)
      {
            population[i].setStrand(createRandomChromosome());
      }
}
```

Figure 37- API function call for Genetic Algorithm extension

Creating a chromosome strand is done by following all the rules described above. The first step is randomly generating a one or zero for both the ranged and melee bits. Then, every other group of bits is generated by creating a random number between one and fifteen. After all of the numbers have been created, the bits are shifted to the desired location and the OR operation is performed to make them one strand. This process is pictured below:

```
/*****************************************************************
 *          AIGeneticAlgorithm::createRandomChromosome
 *
 * Create a random individual.  This is done by a strict set of rules-
 * we must have a weapon all values must be atleast 1.
 *
 * population value is a 22 bit value
 * 1st bit - ranged enemy  1 = ability to shoot crossbow,
 * 0 = cannot shoot crossbow
 * 2nd-5th bit ranged damage per attack
 * 6th bit - melee enemy   1 = ability to use sword,
 * 0 means no ability to use sword
 * 7th-10th bit melee damage per attack
 * 11th-14th bit favorite attack type = 0 favors melee
 * 15 favors crossbow
 * 15th-18th bit attack speed - how fast the agents attacks
 * 19th-22nd bit attentionlevel
 *****************************************************************/
S32 AIGeneticAlgorithm::createRandomChromosome()
{
      S32 chromosome, weapon;
      S32 melee, ranged;
      S32 rangedDamage =0;
      S32 meleeDamage =0;
      S32 attentionLevel, attackSpeed, favoriteAttack;

      ranged = gRandGen.randI(0 , 1);

      if(ranged == 1)
            melee = gRandGen.randI(0 , 1);
      else
            melee = 1;//must have a melee weapon if we have no other
      //we need to have atleast a minimal amount of damage
      rangedDamage = gRandGen.randI(1 , 15);
      meleeDamage = gRandGen.randI(1 , 15);
      attentionLevel = gRandGen.randI(1 , 15);
      attackSpeed = gRandGen.randI(1 , 15);
      favoriteAttack = gRandGen.randI(1 , 15);
      ranged<<=21;
      rangedDamage<<=17;
      melee<<=16;
      meleeDamage<<=12;
      attentionLevel<<=8;
      attackSpeed<<=4;

    chromosome = ranged | rangedDamage | melee | meleeDamage |
             attentionLevel | attackSpeed |favoriteAttack;

      return chromosome;
}
```

Figure 38- API function that creates a random Chromosome

C. Selecting Parents

There are two ways to choose parents. The first is through elitism, which means a certain percentage of parents are passed on directly to their children. The second way to choose parents is through the roulette wheel method. After every generation, a roulette wheel is created with each member of the generation getting a portion of the roulette wheel that matches their fitness. Then, a parameter is used to determine how much of the next generation will be filled using the roulette wheel.

```
/*****************************************************************
 *                AIGeneticAlgorithm::rouletteWheelSetUp()
 *
 * Setups the roulette wheel so that all population gets a
 * portion of the wheel that is proportional to their fitness.
 *****************************************************************/
void AIGeneticAlgorithm::rouletteWheelSetUp()
{
  S32 totalFitness = 0;
  S32 tempRoulette =0;

  for(S32 i = 0; i < GA_POPULATION_SIZE ; i++)
  {
    totalFitness += population[i].getFitness();
  }

  //now setup wheel
  for(S32 i = 0; i < GA_POPULATION_SIZE ; i++)
  {
    tempRoulette += (population[i].getFitness()*360)  / totalFitness;
    population[i].setRouletteEnd(tempRoulette);
  }

  //make sure that we keep the edge case
  if(population[GA_POPULATION_SIZE-1].getRouletteEnd()<360)
      population[GA_POPULATION_SIZE-1].setRouletteEnd(360);
}
```

Figure 39- roulette wheel setup function used to run the roulette wheel in the genetic algorithm.

D. Creating Children

When creating children, the first step is to use elitism, as described in the previous section. The elitism parameter controls the percentage of parents that gets passed directly onto the next generation, meaning they do not go through crossover or mutation. For example, if the number is 10% and we are looking for 100 children, then 10 children will be direct copies of their parents. The process of directly copying the parents into the next generation is illustrated below:

```
int topPercentLevel = GA_TOP_PERCENTAGE * GA_POPULATION_SIZE;

//we want to keep the top percentage so start the next generation by
keeping these
for(idx=0;;idx < topPercentLevel; idx++)
{
      nextGeneration[idx] = population[idx];
}
```

Figure 40- Code demonstrates direct copy of parents to next generation.

The second step is to use the roulette wheel that was setup earlier. This is done by randomly generating a number and seeing where it falls on the roulette wheel. This gives us the first parent. We repeat this step to get the second parent. First, we choose to use multipoint crossover in which every gene has a certain chance of being chosen as a crossover point. An example of multipoint crossover can be seen below:

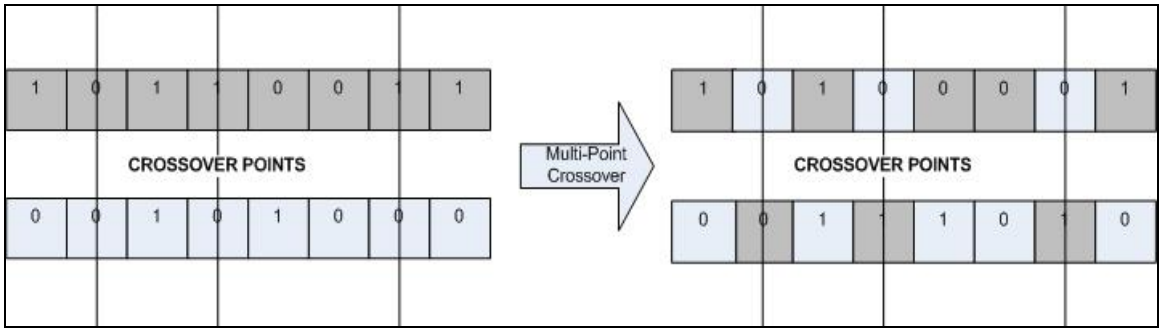

Figure 41- Multi Point Crossover example

If chosen as a crossover point, the algorithm swaps one parent's bit with the other parent's bit. The bits only get swapped if the parent bits are different. The chance that a crossover occurs is based off a parameter called GA_CROSSOVER_RATE. Using this algorithm, we generate two children by looping through the entire strand, as is illustrated in the figure below:

```
/*****************************************************************
 *                      AIGeneticAlgorithm::crossover()
 *
 * Crossover can happen for any bit based off a random chance that
 * crossover rate is less than the number that is randomly choosen.
 *****************************************************************/
void AIGeneticAlgorithm::crossover( AIChromosome * chromosome1,
         AIChromosome * chromosome2, S32 popStrand1, S32 popStrand2 )
{
  S32 random, mask, tempStrand1, tempStrand2;

  for(S32 i =0; i < GA_MAX_BITS; i++)
  {
    random = gRandGen.randI(0, 100);

    if(random > GA_CROSSOVER_RATE)
    {
       mask = 1<<i; //get the location of crossover

       //get the value of the bit in chromsome 1
       tempStrand1 = (popStrand1) & mask;

       //get the value of the bit in chromsome 1
       tempStrand2 = (popStrand2) & mask;

       if(tempStrand2==0 && tempStrand1>=1)
       {
          chromosome1->setStrand(popStrand1 & ~mask);
          chromosome2->setStrand(popStrand2 | mask);
       }
       else if(tempStrand2>=1 && tempStrand1==0)
       {
          chromosome1->setStrand(popStrand1 | mask);
          chromosome2->setStrand(popStrand2 & ~mask);
       }
       //otherwise the bit has the same value for both starnds so do
       //nothing
    }
 }
}
```

Figure 42- crossover function that inside the Torque engine extension for Genetic Algorithm.

After we ran the Genetic Algorithm a couple of times, we also tried single point crossover. The point we chose was after the weapons and damage bits right before the favorite attack bits. Single point crossover swaps the bits after the point of "parent one" with the bits after the point of "parent two". This is done by creating a mask for the upper 10 bits and the lower 12 bits and then combining the upper 10 bits of parent one with the lower 12 bits of parent two by using the OR operator. The second child is made by using the upper 10 bits of parent 2 and the lower 12 bits of parent one. The code that demonstrates this is shown below:

```
/*****************************************************************
 *                      AIGeneticAlgorithm::singleCrossover()
 *
 *  Crossover can happen for any bit based off a random chance that
 * crossover rate is less than the number that is randomly choosen.
 *****************************************************************/
void AIGeneticAlgorithm::singleCrossover( AIChromosome * chromosome1,
AIChromosome * chromosome2, S32 popStrand1, S32 popStrand2 )
{
      S32 random, mask, mask2, tempStrand1, tempStrand2;
      S32 tempStrand3, tempStrand4;

      mask = 4095; //this is a 1 for the bottom 12 bits
      mask2=1023;  //this is a 1 for the top 10 bits
      mask2<<=12;

      //get the bottom 12 of chromsome 1 & 2
      tempStrand1 = (popStrand1) & mask;
      tempStrand2 = (popStrand2) & mask;

      //get the top 10 of chromsome 1 & 2
      tempStrand3 = (popStrand1) & mask2;
      tempStrand4 = (popStrand2) & mask2;

      //set the new chromosomes
      chromosome1->setStrand(tempStrand2 | tempStrand3);
      chromosome2->setStrand(tempStrand1 | tempStrand4);

}
```

Figure 43- single crossover function that inside Torque engine extension.

After crossover, every pair of chromosomes goes through mutation. This process consists of every bit getting a chance to be changed from one to zero or from zero to one. This is done using an XOR operation on every bit that gets mutated. This chance is based on a changeable parameter that can be adjusted as the project dictates. The function used is pictured below:

```
/*****************************************************************
**
 *                AIGeneticAlgorithm::mutateChromosome()
 *
 * Mutates the chromosome based of a random chance number that is
 * compared to the mutation rate for very bit.

*****************************************************************
/
void AIGeneticAlgorithm::mutateChromosome( AIChromosome *
chromosome)
{
  S32 random, mask;
  S8 strand1[] = "00000000000000000000";

  for(S32 i =0; i < GA_MAX_BITS; i++)
  {
      random = gRandGen.randI(0, 100);

      if(random > GA_MUTATION_RATE)
      {
        mask = 1<<i; //get the location of mutation
        chromosome->setStrand((chromosome->getStrand()) ^ mask);
      }
   }
}
```

Figure 44- mutate chromosome function that is inside genetic algorithm extension. Not accessible directly on script side.

After every set of parents has gone through the crossover and mutation steps, the new chromosomes are checked for validity. This is a process where we make sure the new chromosome meets certain mandatory data for it to work in our system. For the purposes of this project, every chromosome must have a 1 in either Ranged Weapon or Melee Weapon. Every other set of four bits must fall in the range of 1 to 15. For example, a value of "0" in the Melee Damage bits would be invalid. If the mandatory criterion for an agent is not met, that agent will be eliminated immediately and replaced with a random chromosome. The validity process is demonstrated in the following code sample:

```
/*****************************************************************
 *                      AIGeneticAlgorithm::checkStrand()
 *
 * checks the chromsome strand to make sure that it is valid
 * it needs melee or bow to be 1 and all other values to be more than
 * 1.
 *****************************************************************/
bool AIGeneticAlgorithm::checkStrand(S32 strand)
{
   S32 maskRanged             = 1<<21;  //mask for ranged
   S32 maskMelee              = 1<<16;
   S32 maskMeleeDamage        = 15<<12;
   S32 maskRangedDamage       = 15<<12;
   S32 attentionMask          = 15<<8;
   S32 attackSpeedMask        = 15<<4;
   S32 favoriteAttackMask     = 15;

   //we have an ivalid starnd right off the bat
   if(strand<=1)
     return false;

   //we have no weapon invalid strand
   if(!(maskRanged & strand)  && !(maskMelee & strand))
     return false;

   //make sure that we have a value >= 1 in the strand for damage
   if(!(maskMeleeDamage & strand) || !(maskRangedDamage & strand))
     return false;

   //make sure that we have a value >= 1 for attack speed and
   //attention level
   if(!(attentionMask & strand) || !(attackSpeedMask & strand) ||
      !(favoriteAttackMask & strand))
       {
         return false;
       }

      //we made it through so we are fine
        return true;
}
```

Figure 45- check strand function used to verify that mutated chromosomes are still valid.

The final step in filling up our next generation with children is to make sure that we have a full set of agents. If we have a total of 80 agents and each generation calls for 100 agents, we need to fill the next 20 with random chromosomes. This is done in the same manner as the initial setup, except for a much smaller number of agents.

E. Evaluating Fitness Levels

Success is measured based on the amount of the player's life left after each battle. The end of the battle is defined as: (1) either all three agents being defeated or (2) the

player being defeated. The fitness score is determined by each agent, starting with a number of points and then subtracting from that the absolute value of the desired player life and the actual player life. This means that the damage for each agent will be added together and then the absolute value of the difference between this total damage and the desired total damage will be subtracted from the maximum number of points. This process is shown in the code below:

```
void AIGeneticAlgorithm::evaluateFitness()
{
  S32 currentFitness;
  double percentage;
  char nextLine[256];

  dSprintf(nextLine, sizeof(nextLine), "RESULTS FOR GENERATION:  %d ",
    currentGeneration);

  fileobj.writeLine((const U8*)"-------------------------------------
      --------------------------------");

  fileobj.writeLine((const U8*)  "     Enemy Life
      Fitness       %");

  fileobj.writeLine((const U8*)"-------------------------------------
      --------------------------------");

  fileobj.writeLine((const U8*)"");
  fileobj.writeLine((const U8*)nextLine);
  fileobj.writeLine((const U8*)"");
  fileobj.writeLine((const U8*)"-------------------------------------
      --------------------------------");

  for(S32 i = 0; i < GA_POPULATION_SIZE ; i++)
  {
    currentFitness =0;
    currentFitness += GA_POINTS_LIFE -
        abs(population[i].getLifeOfEnemy()-desiredLife);

    //want a minmum fitness of zero
    if(currentFitness<0)
      currentFitness =0;

     population[i].setFitness(currentFitness);
     percentage = ((double)population[i].getFitness() /(double)
         (GA_MAX_POINTS)) * 100;

     dSprintf(nextLine, sizeof(nextLine), "%d          %d           %d
          %d%", i, population[i].getLifeOfEnemy(),
          population[i].getFitness(),  (S32)percentage);

     fileobj.writeLine((const U8*)nextLine);
   }
}
```

Figure 46- output function puts the fitness data into a text file.

The reason we chose this method for evaluating fitness was to create a group that doesn't easily triumph over the user, but a group that can win if a player plays poorly but usually puts up a fight and loses. For example, we create a group of three and call the members Agent A, Agent B, and Agent C, we call the user Agent X and we have a desired total damage amount of 50% damage. If Agent X has 100% life to start and Agent A does 10% damage, Agent B does 15% damage and Agent C does 5% damage, the group as a whole did 30% damage. Since the goal was 50% damage, we take 30% subtract 50% and then take the absolute value and subtract it from the maximum total points. This gives us the fitness for the members of this group. If a group does 70% damage, its fitness would be the same as the previous example. We want the group to do as close to the desired amount of damage as possible.

II. Genetic Algorithm Scripting API

When setting up the Genetic Algorithm, there are some things that need to be done on the script side of the game. The script side is also thought of as the game specific code. This is code that is specific to this game and cannot be used in every game unlike the engine components of Genetic Algorithm described earlier. On this side of the code the developer will use API function calls to interface with our game engine code discussed in the previous section. To start, the genetic algorithm has to be created inside the AIManager[3] and passed on to agent loadEntities function where we set up our agents and the think function where the AI decisions are made. These steps are illustrated below:

```
//Create GA if turned on
%genetic_alg = new AIGeneticAlgorithm();
MissionCleanup.add(%genetic_alg);

//spawn all the ai agents
AIManager.loadEntities(%blackboard, %genetic_alg);

//start the control of the ai management system
AIManager.think(%blackboard, %genetic_alg);
```

Figure 47- example of Genetic Algorithm API being used on script side.

Inside the loadEntities function, the reference to the Genetic Algorithm object is used to get the next chromosome. This is an interface with the engine components discussed earlier so the agent can get the next chromosome and use this data to seed itself with the correct variables for the battle. Inside the creation function of each agent, we need to get three things, two of which are only necessary for the project and the third is necessary for using the Genetic Algorithm API. The first is the next chromosome which

[3] This is the script module that was created to control all the AI added to the torque game.

is necessary for everyone. The other two are current index and current generation, both of which used to display data to the screen so we know what generation we are on and what chromosome in the current population. The API for these three calls is demonstrated in the script below:

```
//need to get current index before next chromosome increments it
%chromosome = %genetic_alg.getNextChromosome();
%index      = %genetic_alg.getCurrentIndex();
%generation = %genetic_alg.getCurrentGeneration();
```

Figure 48- shows how the developer can get certain data from the genetic algorithm using various API calls.

In order to access the data on the chromosome, we need to set up a mask of the bits that we desire for every variable that is stored on the chromosome. Then, we need to shift the bits back down and use these values to set up the AIGuard with the desired variables. The code section below demonstrates this process for one variable:

```
%meleeDamageMask   =15<<12;
%meleeDamageLevel =  %chromosome & %meleeDamageMask;
%meleeDamageLevel>>=12;

//set damage amounts
%player.setMeleeDamageAmount(%meleeDamageLevel);
```

Figure 49 - example of developer getting bits of data from the chromosome.

This data is then used as the agents interact with the player in their small battles. After each one of these battles, we need to save the results to be used later in the creation of the next generation and fitness evaluation routines. The data that is used later is the player's life and is achieved with the following function call:

```
%genetic_alg.saveSuccessRateData(%player.getLifeLeft(), %damage,
      %time, %targetObject.storedIndex);
```

Figure 50- shows how the developer can save the data after every battle.

Chapter 7 - Thesis Results

I. API Extension

The primary goal of the thesis was to create an Artificial Intelligence extension to the Torque game engine. This goal was accomplished. We were able to add a series of API functions that allow developers to create agents, and then integrate them together so that they communicate via a blackboard system. The API allows for easy creation of varying missions and skill types. This setup allows for developers to have the freedom of creating whatever missions and skill types that fit into their game. The extension code which is on the game engine side is easily modified, this allows games with different styles and agent types to add more API functionality.

The second API extension that we created was a Genetic Algorithm API. This extension allows the developer to use the agent objects and setup a Genetic Algorithm. This extension worked successfully as we were able to create a Genetic Algorithm that went through the entire routine including crossover, mutation and fitness evaluation. However, with this extension we ran into problems when trying to learn from generation to generation, this will be discussed later in this chapter.

II. Genetic Algorithm Output

When outputting data, we need to be able to store our enemy agent chromosomes because developers would need this data when setting up the final version of the game. The developers want to pool the best scoring agents to choose from when the game is online which was accomplished by outputting all the data to a text file after every generation. For each agent in the generation, chromosome value, fitness score and the players' remaining life left were stored. The only exception to this occurred when scoring the agents individually instead of in a group, in which case, the damage each agent accomplished was used instead of the player's life. After each generation, an average fitness score for that generation is outputted to track the growth of the agents. An example of a small population, first generation with the maximum fitness score for this trial run of 200 is below:

```
-------------------------------------------------------
GENETIC ALGORITHM RESULTS
-------------------------------------------------------
Agent     Chromosome     Enemy Life    Fitness     %
-------------------------------------------------------
RESULTS FOR GENERATION:  0
-------------------------------------------------------
0         2968419        0              180        90
1         893784         0              180        90
2         4013803        0              180        90
3         2973457        56             164        82
4         746081         56             164        82
5         254696         56             164        82
AVERAGE FITNESS              86
```

Figure 51- Example of Genetic Algorithm results

III. Genetic Algorithm Results

A. Single Point Crossover vs. Multi Point Crossover

When first working on the project, we went with multi point crossover. This was chosen to give the system the most flexibility. Allowing the algorithm to crossover on any bit seemed like it would allow for the most diversity when creating groups of agents. However, after a couple of test runs, it was determined that allowing crossover at every point was not ideal. Crossing over so frequently, and at an unfixed position, allowed for invalid individuals to be created at a rate of 30%-35% of the population. Since invalid individuals are dealt with by creating another random valid individual, we are not maximizing learning at least 30% of every generation was created at random. This problem was dealt with by adding a single point crossover function and using it instead. The single point crossover function was fixed at the 12^{th} bit position. The 12^{th} bit position was chosen because all of the bits to the left of it go together like melee damage and meleeAttack. This modification reduced the number of invalid individuals to around 1% solely from mutation, a much more acceptable level of randomly created individuals per generation.

While using multi point crossover, the agents were getting very sporadic results because of the invalid individuals. For example, one generation might have a very reasonable 80% average fitness score and the next generation would have a 54% average fitness score. Ideally in a genetic algorithm you would continue to build on the success of a generation. However, we were unable to accomplish this because of the high turnover rate for each generation.

Single point crossover gave us a good distribution of agents in our initialization. However, single point crossover ran into its own complications. The generation fitness scores did not improve. We checked both the mode and the mean scores of fitness and the distribution was normal for almost every generation. This means there was no improvement. After further inspection, it became apparent that the same chromosome value in one generation does not score the same in the next generation. An example of this is seen below:

```
-------------------------------------------------------
GENETIC ALGORITHM RESULTS
-------------------------------------------------------
Agent       Chromosome     Enemy Life   Fitness       %
-------------------------------------------------------
RESULTS FOR GENERATION:  0
-------------------------------------------------------
0           1667862        81           199           99
-------------------------------------------------------
RESULTS FOR GENERATION:  1
-------------------------------------------------------
117         1667862        28           148           74
-------------------------------------------------------
```

Figure 52- example of two small generations output

In order to isolate this theory, we created a fitness evaluation that looks at total damage by each agent individually instead of the group.

B. Group Fitness vs. Individual Fitness

Since groups of agents were not successfully learning we changed the number of enemy agents in each from three to one. The goal was to isolate the problem in either the blackboard architecture or the Genetic Algorithm. However, after many tests it was determined that even individual agents fighting finished with the same non-deterministic results that their group counterparts finished with. An example can be seen below, where we have the same exact chromosome fighting in a one on one environment and the results are clearly different. With these scores we cannot successfully create a Genetic Algorithm that will allow the agents to learn no matter what the number of enemy agents is in each battle.

```
-------------------------------------------------------
GENETIC ALGORITHM RESULTS
-------------------------------------------------------
Agent       Chromosome     Enemy Life   Fitness       %
-------------------------------------------------------
RESULTS FOR GENERATION:  0
-------------------------------------------------------
0           1812083        50           150           75
-------------------------------------------------------

RESULTS FOR GENERATION:  1
-------------------------------------------------------
40          1812083        20           120           60
-------------------------------------------------------
```

Figure 53- Example of two single agent generations

C. Randomness of the Game Engine

In this project a significant problem we ran into is the randomness of the game. For instance if we have two identical chromosome values, one would expect the results of each chromosome to be the same or near the same. However, in the game environment, the fitness values varied significantly. The reason for this is the game engine has some elements of random chance built in. First, it is possible to block an attack by having both swords hit each other this can cause agents to change location by being pushed back. This built in feature can either put the agent at an advantage in the battle or a disadvantage. Second, the team members might have better or worse chromosome values which affect the performance of the individual agent. Third, the path finding might allow the agents to take slightly different routes which can result in a different agent position than the last time through the Genetic Algorithm.

Finally, the simulated player's performance also comes into consideration. The simulated player follows some basic rules: (1) find the closest enemy agent and attack (2) every so often the player does a side step to avoid getting stuck in battle and (3) when an enemy agent dies, it moves onto the next closest agent. This can cause a difference whenever the new group of agents is picked. If the agents are placed in different locations, then the simulated player might attack a different one to start with, which can cause different results. All of these concepts contributed to the randomness of each agent and group agents. This randomness ultimately made it unreasonable to expect deterministic results from the agents. Without the deterministic results no learning would be possible.

After piecing all of the information we gathered together it is our conclusion that in order to successfully create a Genetic Algorithm that works in a team or single enemy environment we need to create the gaming engine with this in mind. The reason for this is because of the lack of deterministic behavior that we received when using the Torque Game Engine environment. It has become apparent that the difference in timing mechanisms and possibly path finding algorithms from battle to the next, despite having the same identical agents causes us to get undesirable results.

IV. Future Work

A. Blackboard Architecture

There is a lot of room for future work within the Blackboard Architecture. One area is the API. Because the focus of the project was on Genetic Algorithms and group learning, the API for the blackboard system was limited. The system consists of a way to setup missions and apply for missions. As well as an arbiter that makes decisions on who gets to be assigned to what missions. More attention could be paid to different types of missions, such as, non combat missions like mining, or time sensitive missions. Non combat mission options would allow agents to go off and perform activities like mining for a designated time or until they reached a desired level of resource mined. This adaptation would be dealt with by the arbiter thus controlling what had the highest priority.

There is also room for future work when dealing with changing priorities and swapping missions. Priorities are easily changeable in this system. The only thing flexible about priorities in this project occurred when an agent applied for a mission and decided what priority it should be. In our system, the priority is set ahead of time by the developer. This could be integrated into the system to allow for variable priority depending on what other agents are doing and what this agent has done in the past.

Another area that the blackboard could be improved upon is scope. The blackboard could be expanded to allow for use in a real time strategy game. Blackboard Architecture is used in some real time strategy games. Adding to the engine this capability would be very useful because, it would allow armies to communicate and attack in unison. For example, a general could make a mission for 100 agents to attack from the left and also for 50 agents to attack from the right. Currently, the blackboard system handles this. However, the system does not support the general sending 10 agents on a side mission and, then when this is finished, going back to the original mission.

The blackboard in this system does not allow for filtering areas of information. This would allow agents to focus on their current behavior and only be interrupted when

there is information that pertains to them. They would not waste time digesting data that is not relevant to them.

The ability to section off the blackboard into communities of interest would also be a useful addition to the system. This would speed up communication as developers added more types of agents, missions and skills. This reason communication would improve is the agents that care about a specific area of information would only need to go through that area of the blackboard. This would speed up the whole process since agents would not look at unnecessary data and the arbiter could assign work quicker when the areas of interest are narrowed down.

B. Genetic Algorithm

The largest area of future work is to construct an entire game engine with the idea in mind that a Genetic Algorithm would be used. The reason for doing this would be to strictly control all data so we get the best Genetic Algorithm results. This control would allow us to define how long certain processes are allowed to take and control path finding more closely. The end result would be a deterministic system and then further tests could be done on team Genetic Algorithm.

There are a few other areas that could be looked into from a Genetic Algorithm standpoint. One possibility would be to add other pieces of data to the chromosome. For example, (1) the life of the enemy agents, (2) other weapons, (3) field of vision and (4) health regeneration could all be integrated into the chromosome. All of these could easily be integrated into the chromosome data. This would allow the developer to add many different parameters that the agent has to the chromosome and then run the Genetic Algorithm and get close to the ideal parameter values.

Another area of improvement would be to further explore the crossover issue discussed in the Results section. We tested out fixed single point crossover and random multi point crossover. However, no work was done on finding a fixed multi point crossover that might work. This process would include testing out which fixed points worked give the best results. To start, we would make a fixed point at each group of bits, ensuring that no partial group of bits or bit would be crossover without the rest of that group. This would ensure that we have a valid strand before mutation and allow for crossover of more groups of bits than the single point crossover.

C. Game Considerations

On the game side, there are quite a few concepts that could have more work done on them. For example, (1) more missions and skills, (2) more expansive blackboard, and (3) other learning algorithms also the need for a deterministic system, all of these things would increase the power and flexibility of the system. This increased control would

allow developers the ability to create an expansive blackboard infrastructure that a large group of agents can communicate through.

More missions and skills would make the blackboard system more diverse and allow the developers to create different types of games using the same structure. It also allows the system to deal with multiple types of agents: one could have an agent that only gathers and an agent that both attacks and gathers and an agent that only attacks. Setting up the system to easily handle the different types of agents, missions and skills would significantly improve the flexibility.

Other learning algorithms could also be added to the project in an effort to get a direct comparison of Genetic Algorithms, such as neural networks. This would allow the developer to try different approaches and see which ones fit the scope of the game the best.

References

Rabin, S., (2002) AI Game Programming Wisdom, Hingham, Massachusetts, 2002.

Rabin, S., (2004) AI Game Programming Wisdom 2, Hingham, Massachusetts, 2004.

Rabin, S., (2008) AI Game Programming Wisdom 4, Hingham, Massachusetts, 2008.

Schwab, B., (2004) AI Game Engine Programming, Boston, Massachusetts, 2004.

Morrison, M., (2005) Beginning Game Programming, Indianapolis, Indiana, 2005.

Leandro Nunes de Castro, (2006) Fundamentals of Natural Computing Basic Concepts, Algorithms, and Applications, Boca Raton, FL, 2006.

Lassabe, N., (2007) A New Step for Artificial Creatures, France 2007.

Grand, S., (1996) Creatures: Artificial Software Agents for Home Entertainment, Cambridge U.K. 1996.

Grand, S., (1997) Creatures: Entertainment Software Agents with Artificial Life, Cambridge U.K. 1997.

Sims, K., (1994) Evolving Virtual Creatures, Computer Graphics Annual Conference Series July 1994 pp.15-22.

Sims, K., (1994) Evolving 3D Morphology and Behavior by Competition, Artificial Life IV proceedings 1994, pp 28-39.