USING GENETIC ALGORITHMS TO EVOLVE CHARACTER BEHAVIOURS IN MODERN VIDEO GAMES

T. Bullen and M. Katchabaw
Department of Computer Science
The University of Western Ontario
London, Ontario, Canada N6A 5B7
tbullen@uwo.ca, katchab@csd.uwo.ca

KEYWORDS

Artificial intelligence, bots, genetic algorithms, evolutionary algorithms, computer and video games

ABSTRACT

Artificial intelligence is an important aspect to nearly every modern video game. Providing this, however, is all too often an arduous task, even for the most expert developers. The behaviours of non-player characters in a game are typically defined and guided by a large collection of parameters; it is usually quite difficult to determine the best values for these parameters to achieve the desired behaviour considering the state of the game and the player involved in playing it. Some form of adaptation to adjust and tune these behavioural parameters would be extremely useful in addressing this problem.

This paper examines the use of genetic algorithms to adapt and refine character behaviours in video games. In doing so, non-player characters can be evolved to a fitness level appropriate to the game and its player, providing a more enjoyable experience in the end. This paper discusses our approach to using genetic algorithms, and describes a prototype system built using the Unreal Engine that implements this approach in its non-player characters. This paper also presents experimental results from using this prototype system; to date, these results have been quite positive, demonstrating great promise for the future.

INTRODUCTION

In recent years, artificial intelligence has increasingly become one of the most critical factors in determining the success or failure of a video game (Tozour 2002). This trend is expected to continue, with some saying that the key to more entertaining, enjoyable, and immersive games in the future lies in the artificial intelligence contained within them (Bourg and Seemann 2004).

Unfortunately, developing the artificial intelligence for a game is one of the most challenging tasks a programmer can undertake (Rabin 2002). Indeed, creating non-player characters that behave in a believable and realistic fashion, while working in the game to provide an appropriate challenge to the player, is incredibly difficult (Baillie-de Byl 2004). Such characters are typically defined and guided by a large collection of behavioural parameters whose interactions and dependencies can be complex and difficult to predict (Laramée 2002; Sweetser 2004; Thomas 2006). Configuring all of these parameters for game characters manually is a tedious, expensive (in terms of time and money), and potentially error prone process. Consequently, an approach is necessary to automate behavioural parameter configuration, to adapt and refine character behaviours as necessary for a game.

Our current work explores this problem through the use of genetic algorithms to develop non-player characters. This is done by using evolutionary processes to adapt behavioural parameters of the characters to a level of fitness suitable for the game context and player in question. Doing so has the potential to provide the player with a more enjoyable and appropriately challenging experience, without the problems and costs that are usually associated with the manual tuning and configuration of these behavioural parameters (Laramée 2002; Sweetser 2004; Thomas 2006). With evolution and adaptation already identified as highly important directions to the future of artificial intelligence for non-player characters in video games (Bourg and Seemann 2004), now is the time to study and explore this area further.

To this end, we have developed a mutator module for Epic's Unreal Engine (Epic Games 2005) that applies genetic algorithms to its non-player characters, also known as bots. This mutator enables Unreal bots to evolve as the game is played to adapt to their surroundings, the rules of the game, and the opponents they are facing. Our mutator module was then used in a series of experiments conducted using Unreal Tournament 2004 (Digital Extremes 2004) to investigate and determine the effects of the genetic processes put in place within the bots.

This paper presents the results of our current and on-going work in this area. We begin by providing background information on genetic algorithms and evolutionary computing, as well as a discussion of related work in this area. We then describe our approach to genetic algorithms to evolve character behaviours, and introduce our proof of concept system using the Unreal Engine. We then present experimental results from using this prototype to date, and discuss our experiences in using it so far. We then conclude this paper with a summary and a discussion of potential directions for continued research and development in the future.

BACKGROUND AND RELATED WORK

Genetic algorithms have been used in numerous contexts for quite some time, including artificial intelligence, as discussed in (Russell and Norvig 2003). Genetic algorithms have also been examined in the particular context of artificial intelligence for video games, including (Baillie-de Byl 2004; Bourg and Seemann 2004; Buckland 2002; Laramée 2002; Sweetser 2004) and several others, although much of this attention is relatively recent.

A Brief Overview of Genetic Algorithms

The essence of a genetic algorithm is much the same across domains: computational problems are encoded in such a way that natural evolutionary processes can be applied to them to produce optimal or near-optimal solutions (Laramée 2002). The general flow of the process is depicted in Figure 1, and discussed in the sections that follow.

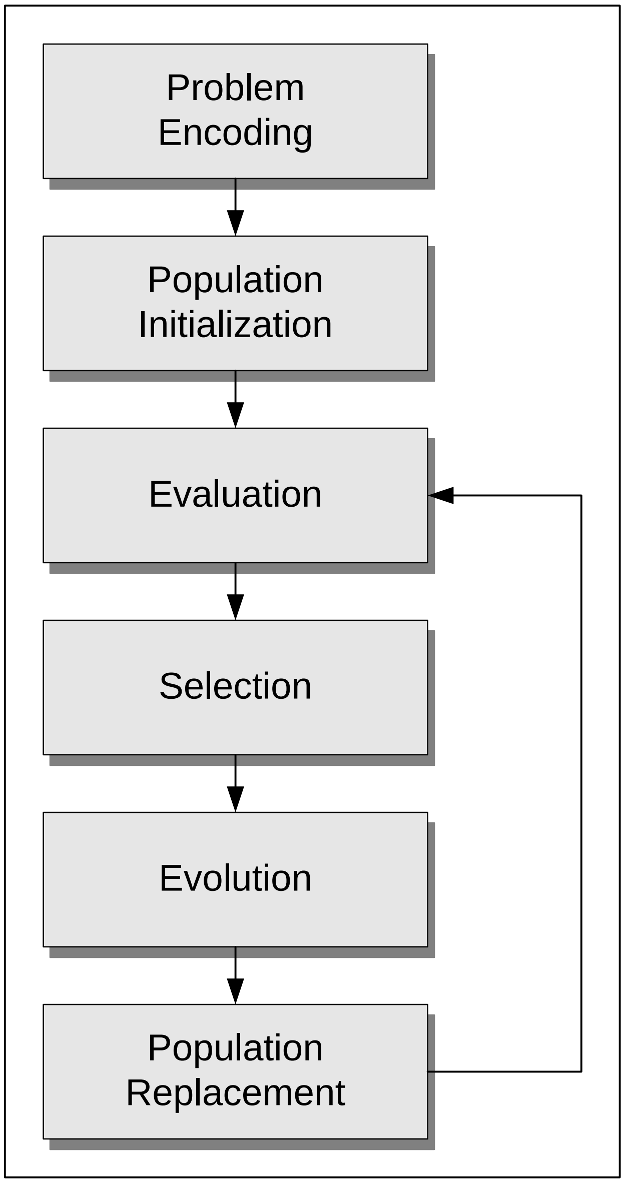


Figure 1: The Flow of a Genetic Algorithm

Problem Encoding

To use genetic algorithms to solve a problem, we must think of our problems from a genetics perspective. For our purposes, genetics concentrates on the transmission of traits from parents to offspring (Baillie-de Byl 2004). These traits are determined by the genes present in the chromosomes of the entities in question. In the end, these traits define the various characteristics and capabilities of an individual.

When dealing with genetic algorithms, we encode problems in this fashion, defining the various traits of a problem and its solutions through the use of genes. Typically, genes tend to be data variables containing values representing the traits in question, although it is possible for them to be elements of logic or code instead (Laramée 2002).

Population Initialization

To begin the process, we need a population of individuals, with each individual a potential candidate for solving the problem at hand. Each individual is defined by generating the collection of genes that determine its various traits. This can be done using some form of random process, or by some more informed process that creates individuals that should be inherently better suited to solving the problem at hand than a randomly generated one.

The latter of these options, however, must be used with care, as it could create a population that lacks the genetic diversity to contain the best solution to the problem, as sometimes the best solution comes from the most unlikely of candidates. With care though, a more informed population generation process can lead to a more efficient execution overall, in some cases.

Evaluation

The evaluation process determines which individuals in the population are the most successful. Typically, this is done through the application of a fitness function that assigns a score to each individual in the population. The closer an individual is to solving the problem, the higher its assigned fitness score. Naturally, the fitness function is very problem specific, and the overall success of the genetic algorithm is heavily dependent on the selection of an appropriate fitness function (Sweetser 2004).

Selection

After a fitness score has been assigned to each individual in the population, a mechanism is needed to select which individuals will become parents and reproduce to create offspring for the next generation of the population. There are many approaches to this selection process, as discussed in the literature listed earlier in this section.

Evolution

During reproduction, each parent transmits a portion of their genetic material to their offspring. The process is not simply one of copying, but usually involves other activities, most importantly crossover and mutation (Laramée 2002). Crossover involves mixing gene components from the chromosomes of each parent so that the resulting offspring has a combination of traits from the parents involved in its creation. Mutation is a random change to a gene that creates variation in the offspring so that, in some respects, the offspring can be unlike its parents. This prevents stagnation and premature convergence in a population, but care must also be taken to avoid too many changes that make the genetic algorithm too random and too inefficient (Sweetser 2004).

Population Replacement

When a new generation of individuals has been created as described above, they enter the population, potentially displacing and replacing individuals from previous generations. Depending on the genetic algorithm in question, this may be a total replacement of all individuals, or some select individuals from previous generations may be allowed to survive. Once the new population has been assembled, the process repeats. After sufficient repetitions, the population will evolve and a suitable solution to the problem will hopefully be found amongst the population during evaluation.

Related Work

As mentioned earlier, genetic algorithms have been applied to artificial intelligence for video games in the literature before, including (Baillie-de Byl 2004; Bourg and Seemann 2004; Buckland 2002; Laramée 2002; Sweetser 2004). While this work has done an excellent job of introducing genetic algorithms in this context, applications of genetic algorithms in this work have been quite limited to rather simplistic characters and scenarios, without examining games of a commercial scope or magnitude. It is also unclear how much experimentation was conducted in this work, as presentation of results was also rather sparse for the most part.

Further work in this area has examined more advanced genetic algorithms for game artificial intelligence, including (Buckland 2004; Laramée 2004; Thomas 2004; Thomas 2006). While presenting some rather interesting and practically useful techniques, this work is again limited in terms of its proof of concept and experimental results.

More rigorous application of genetic algorithms to video games is starting to appear in the literature, however, with (Spronck and Ponsen 2008) being a notable example. This work uses genetic algorithms to generate strategies for real-time strategy games. While there are many caveats to this work, as described in (Spronck and Ponsen 2008), the work is quite promising and demonstrates the potential for using genetic algorithms in this area.

There has also been interesting applications of genetic algorithms in commercial video games, as discussed in (Sweetser 2004), including Cloak, Dagger, and DNA, the Creature series, Return Fire II, and Sigma. Spore, developed by Maxis for Electronic Arts and expected to be released in late 2008, also makes use of genetic algorithms and evolutionary computing in a variety of ways. Unfortunately, the extent to which these approaches have been used in these and other commercial games, as well as their ultimate success, is unclear.

Consequently, while there has been considerable discussion on using genetic algorithms for artificial intelligence in video games, there is also considerable room for additional research, development, and experimentation to explore this area further.

OUR USE OF GENETIC ALGORITHMS

In our current work, we are studying the use of genetic algorithms to evolve character behaviours in video games. Consequently, our population will consist of non-player characters with their traits and characteristics encoded as the genes used during evolution.

Using the Unreal Engine as a Research Platform

Instead of creating our own simple game or game scenarios to explore genetic algorithms in this way, we instead chose to use a commercial game system as our research platform. This allows us to focus on issues and experiments related to genetic algorithms, as opposed to the construction of the game itself and its characters. For this purpose, we chose to use Epic's Unreal Engine (Epic Games 2005). The Unreal Engine is a fairly popular engine among developers and hobbyists, providing a reasonably large collection of games suitable for study. This, and our own prior experience with the Unreal Engine, made it an ideal candidate for use in our current work.

Since we were using the Unreal Engine in this work, our system for genetic evolution was developed using UnrealScript. While a C or C++ approach is preferable to provide support across a variety of games and game engines in the long term, most game engines used in industry do not provide code-level access to their engines or only do so in a cost-prohibitive fashion, including the Unreal Engine. UnrealScript fortunately provided all the access that was required for our current work.

Adding genetic evolution to the Unreal Engine involved manipulations of its non-player characters, known as bots, as well as its game rules, as shown in Figure 2. Each Unreal game type has a Game Info object that defines the game in question. Among other things, this object contains a collection of game rules defining various aspects of how the game is played, and a collection of mutators. Mutators, in essence, allow modifications to a game and gameplay at run-time while keeping the core elements and game rules intact.

In our case, we developed a Genetic Evolution Mutator to bootstrap the genetic evolution code within the Unreal Engine. Upon loading, this mutator instantiates a collection of Evolution Rules and adds them to the list of game rules in the engine to control the evolutionary process depending on the configuration of the mutator. This mutator also modifies the Pawn class from which all Unreal bots are derived, to remove its reference to the default artificial intelligence controller and replace it with one to a new bot controller that contains a genetic algorithm. Making this change forces all newly constructed Unreal bots to use the

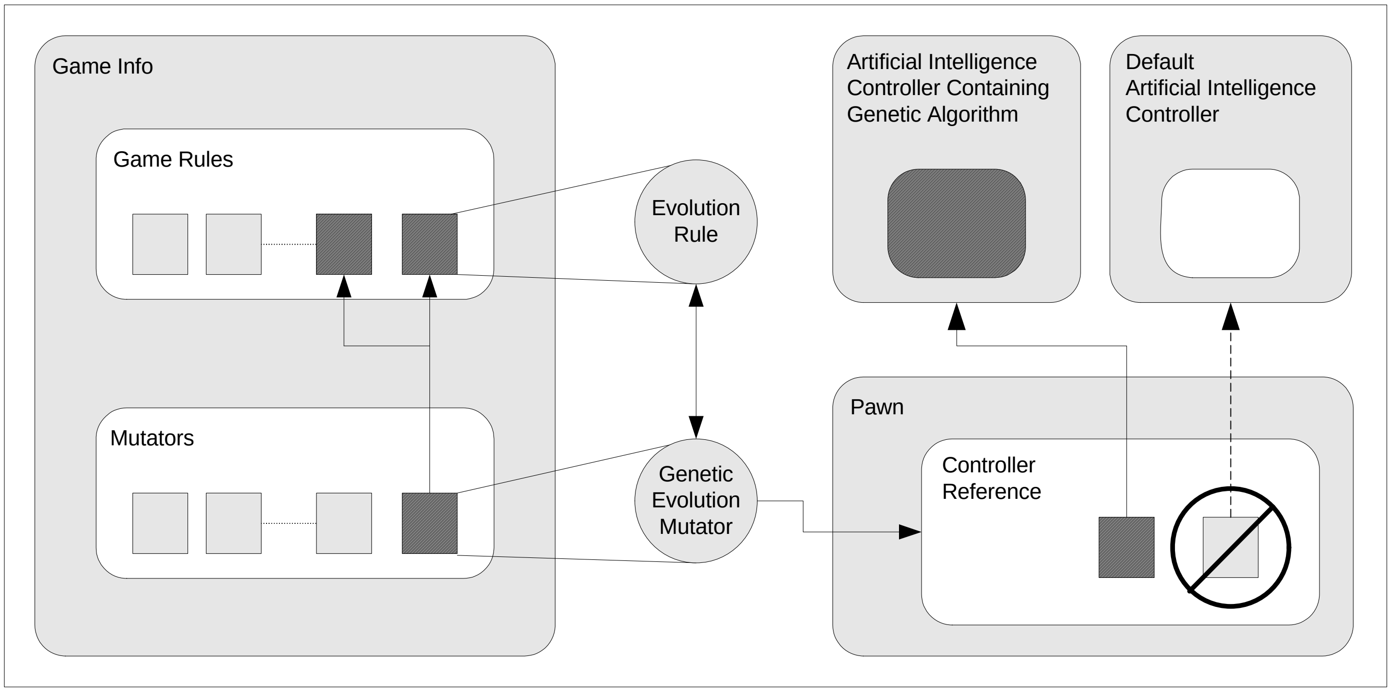

Game Info
Game Rules
Mutators
Evolution Rule
Genetic Evolution Mutator
Artificial Intelligence Controller Containing Genetic Algorithm
Default Artificial Intelligence Controller
Pawn
Controller Reference

Figure 2: Additions and Modifications to the Unreal Engine to Support Genetic Algorithms

new controller instead of the default one. This new controller determines the behaviour of the bots making use of the controller, and consults the Evolution Rules to control the genetic evolution of the bots to refine and adapt their behaviour. In doing things in this fashion, we do not need to make changes to the core of the Unreal Engine code, and only need to deploy our mutator to enable genetic evolution in the Unreal bots.

Using Genetic Algorithms in Unreal Tournament 2004

In adding to and modifying the Unreal Engine as described in the previous section, we can now use genetic algorithms in Unreal-based games. The selection of chromosomes, genes, fitness functions, selection criteria, and other elements of genetic algorithms as discussed earlier in this paper, however, is dependent on the particular game making use of this engine.

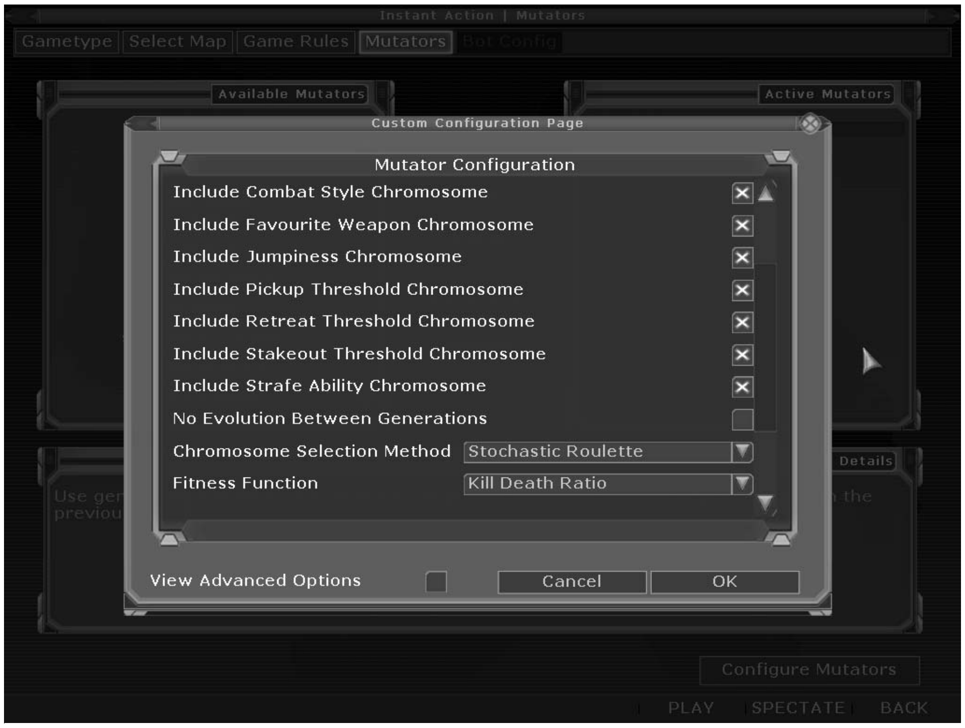
For our purposes, we used Unreal Tournament 2004 (Digital Extremes 2004), as it is one of the most popular Unreal-based games, and it was readily available at our disposal. Unreal Tournament 2004 is a first-person shooter game that supports a wide variety of different game types and sets of game rules, individual and team-based games, and single player, multiplayer, and spectator modes of play. (In spectator mode, games can be played with no human players, and the game's display is used to observe the game's progress.) Consequently, there are many gameplay options provided within this game, enabling a wide variety of experimentation with genetic algorithms using just this single package.

Problem Encoding

Since Unreal Tournament 2004 is a first person shooter, gameplay primarily revolves around killing other players (both humans and bots) while trying to stay alive yourself. Consequently, most player activity focuses around completing these objectives, as well as collecting items that facilitate these objectives (such as weapons, ammunition, health packs, armor, and so on). Some game types supported by Unreal Tournament 2004 have additional objectives as well, such as capturing a flag from your opponent's base, controlling critical points in the game world, and so on. These gameplay objectives represent the problem that we are trying to solving using genetic algorithms.

The bots in the game form the population, and their various characteristics and traits collectively form the chromosomes and individually can be considered the genes for our genetic algorithm. Since we are primarily interested in refining the behaviour of these bots, we focus on traits that influence a bot's decision making processes and have an impact on the outcome of the game, as opposed to traits that only affect their visual appearance or voice within the game. As a result, we consider the following traits of Unreal bots in the set of genes and chromosomes within our genetic algorithm:

- Accuracy: Determines how good a bot is at hitting its target when shooting at it.
- Alertness: Determines how aware a bot is of changes to their surroundings.
- Aggression: Determines how engaged a bot is during combat and how they react to combat.
- Jumpiness: Determines how much a bot will use jumping, especially as an evasive maneuver.
- Strafe Ability: Determines how much a bot will use strafing, especially as an evasive maneuver.
- Combat Style: Determines how a bot engages in combat, either up close or far away, or somewhere in between.



Mutator Configuration

Include Combat Style Chromosome	✓
Include Favourite Weapon Chromosome	✓
Include Jumpiness Chromosome	✓
Include Pickup Threshold Chromosome	✓
Include Retreat Threshold Chromosome	✓
Include Stakeout Threshold Chromosome	✓
Include Strafe Ability Chromosome	✓
No Evolution Between Generations	☐
Chromosome Selection Method	Stochastic Roulette
Fitness Function	Kill Death Ratio
View Advanced Options	☐

Figure 3: Configuration Screen for Genetic Evolution Mutator

- Reaction Time: Determines how quickly a bot responds to changes to their surroundings.
- Favourite Weapon: Determines which weapon a bot will prefer to use, given the choice.
- Retreat Threshold: Determines how likely a bot is to disengage from combat when facing a stronger opponent.
- Pickup Threshold: Determines how likely a bot is to seek out a better weapon than the one it is currently using.
- Stakeout Threshold: Determines how long a bot will continue to hunt for an opponent outside its field of vision.

There are other traits that a bot possesses, but their effects are not documented, and so they are currently being studied further before inclusion within our genetic algorithm. Our mutator can be configured at run-time to determine which traits to include or exclude from evolution, as shown in Figure 3, providing a great deal of flexibility and control over the process.

Population Initialization

The initial population of bots to use in our genetic algorithm is generated through a random selection from all of the available bots within the game. This, of course, is a subset of all of the bots that are possible through a completely random assignment of all trait values.

This population initialization decision was made as a great number of the bots possible in the game are extremely ineffective at playing the game well, and these bots needed to be culled for efficiency reasons. Since additional arbitrary bots can be easily added to the bot roster for the game, there can still be as much diversity as needed in the initial population used by the genetic algorithm.

Evaluation

For evaluation purposes, we have defined a number of fitness functions, primarily aimed at assessing a bot's success in killing its opponents and/or avoiding its own death. These include the following:

- Gross Kills: Fitness is determined by the total number of opponents killed during the game. This will favour bots that tend to kill opponents, regardless of the consequences.
- Deaths: Fitness is determined by the number of times the bot was killed during the game. This will favour bots that are survivalists, regardless of how many opponents they kill in the end.
- Net Kills: Fitness is determined by the total number of opponents killed, minus the number of deaths incurred in doing so. This will favour more balanced and cautious bots.
- Kill/Death Ratio: Fitness is determined by a weighted ratio of kills to deaths. This is calculated so as to favour killing activity during the game, although this can be easily tuned. This fitness function was introduced as an improvement over the Net Kills fitness function, as this function would rate a bot with 0 kills and 0 deaths the same as a bot with 10 kills and 10 deaths, even though the latter was more actively participating in the game.

It is not obvious which fitness function results in bots that provide the most enjoyable experience to the player. Furthermore, it is unclear how well these functions apply to games with objectives beyond a simple kill-or-be-killed deathmatch, or when team play is involved. Experimentation is needed to study these issues and explore them further.

Selection

A number of methods, as described in (Baillie-de Byl 2004), have been defined for selecting bots to be parents to generate offspring in our genetic algorithm. Each of these selection methods makes use of either the raw fitness score from the evaluation process, or a fitness ratio, which is the individual's fitness divided by the population's total fitness. These methods include the following:

- Stochastic Roulette: Each potential parent from the population is allocated a portion of a circular roulette wheel, the size of which represents its fitness ratio. A parent is selected for mating by conceptually spinning the wheel and picking the parent on which the wheel stops. The fitter parents have a bigger portion of the roulette wheel and so have a better chance of being selected to produce offspring.
- Remainder Stochastic: A parent is selected for mating based on its fitness ratio, converted to an integer on a scale from 0 to 100. This value determines the number of times the potential parent is allowed to mate.
- Ranking Mating: In this simple approach, potential parents are ordered based on their fitness; parents near the top of the order are selected to produce offspring more times than those lower down. A cut-off point can be configured with this method, below which bots are not allowed to mate due to their poor performance during evaluation.

As with traits and fitness functions, the selection method used in our genetic algorithm can be adjusted by configuring our mutator, as shown in Figure 3.

Evolution

The genetic algorithm used in this work employs both crossover and mutation in creating offspring from parents selected using one of the above methods. Crossover is accomplished by swapping segments of chromosomes from parents using a random process when constructing offspring. Mutations occur randomly in offspring, with the offspring receiving traits that were not from one of their parents, but were instead randomly generated. The probability of mutation occurring is again a parameter configurable in our Unreal mutator.

Population Replacement

In our genetic algorithm, population replacement is again configurable in our mutator. By default, the entire population is replaced by offspring after evolution has occurred. Options exist, however, to keep bots selected either by fitness or randomly from one generation to the next.

EXPERIMENTAL RESULTS AND EXPERIENCES

Using the Unreal-based prototype system described in the previous section, a series of experiments was conducted to study the use of genetic algorithms in evolving bot behaviour in Unreal Tournament 2004. This section presents highlights of results from this experimentation, and discusses some of the observations made and insights gained in the process.

Experimental Environment

Our experimental environment consisted of a lab of 20 workstations, allowing us to conduct multiple experiments in parallel. Each test system in the lab was a dual-core 3.0GHz Pentium D system, with 2GB RAM, a 250GB hard drive, and an ATI X800 graphics accelerator card. The operating system in this case was Microsoft Windows XP SP2. As such, the test systems greatly exceeded the recommended system requirements for Unreal Tournament 2004.

Deathmatch Experiments

In this experimentation, we studied our prototype system with bots playing a standard deathmatch game. The game was set in one of the largest levels provided in Unreal Tournament 2004, Tokara Forest, to allow the largest possible number of bots in the game at once.

In total, 32 bots were allowed in the game, split into two groups of 16 bots each. The first group of bots made use of the genetic algorithm as described in the previous section to evolve over time. The second group of bots was a fixed control group that did not evolve over time. Both groups were selected randomly at the beginning of each repetition of the experiment; there were five repetitions in total, providing five different starting points for evolution against five different control groups.

All bots were configured to be of a "masterful skill" level. The genetic algorithm was configured to allow all of the traits discussed earlier to be affected by evolution, with a 0.2% chance of mutation. Fitness was calculated using the Kill/Death Ratio, and parent selection was done using the Stochastic Roulette method.

The game itself was configured to run until either 20 minutes had elapsed, or a target kill level of 100 kills was achieved by one of the bots. The experiment was then configured to repeat through 25 generations of evolved bots, with evolution occurring after each game was completed and before a new game was started.

Figure 4 presents results from this set of experiments, plotting the fitness difference between the evolving bot population and the control population through each generation of evolved bots. This fitness difference was calculated as the mean evolved bot fitness minus the mean control bot fitness across all replications of the experiment. As the bots using the genetic algorithm evolved, the fitness difference increased, indicating that the evolved bots improved against the control group over time. To make this trend easier to see, Figure 5 sums the fitness differences from Figure 4 into fifths. (The first bar in the graph in Figure 5 is the sum of the first five fitness differences from Figure 4, and so on.) From Figure 5, an improvement in evolved bot fitness is quite apparent over time.

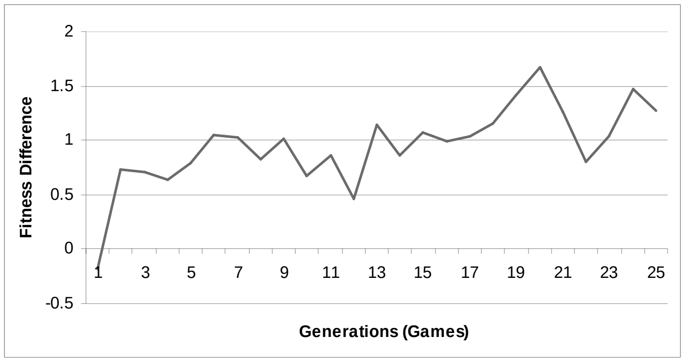


Figure 4: Fitness Differences Between Evolved and Control Bots in Deathmatch Play

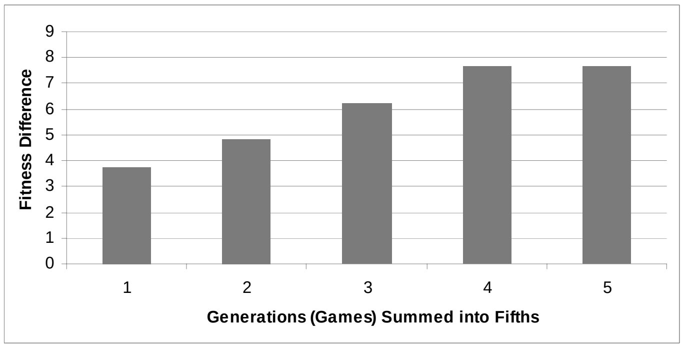


Figure 5: Fitness Differences Between Evolved and Control Bots in Deathmatch Play, Summed into Fifths

Team Deathmatch Experiments

Following the success of the pure deathmatch experimentation as described above, we conducted a similar set of experiments except that the bots were organized into teams. While the best overall team score determines the victor in this type of game, the best strategy for success is to largely play the same as a pure deathmatch, with a few exceptions (Suit et al. 2007).

Consequently, our team deathmatch experiments were conducted with the same configuration as our pure deathmatch experiments, except that the evolved bots formed one team and the control bots formed the other. The teams then competed against one another following the same rules as before. Figure 6 presents the fitness differences measured in this case.

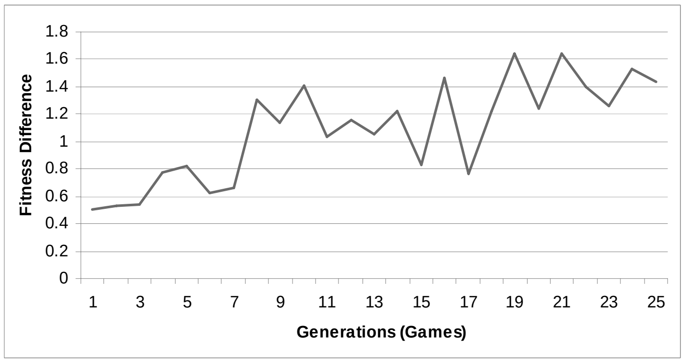


Figure 6: Fitness Differences Between Evolved and Control Bots in Team Deathmatch Play

Once again, the evolved bots demonstrated an improved fitness over time compared to the control group. This trend is readily apparent in Figure 7, which sums the fitness differences from Figure 6 into fifths.

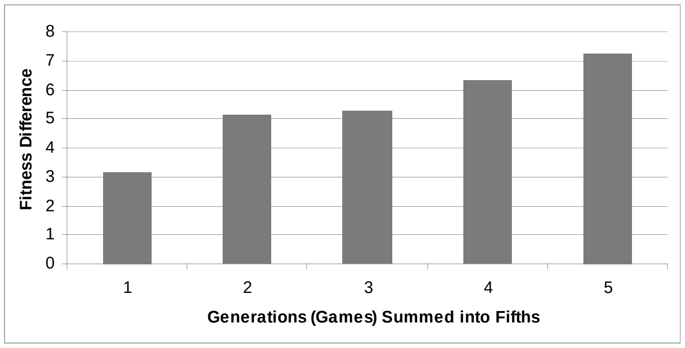


Figure 7: Fitness Differences Between Evolved and Control Bots in Team Deathmatch Play, Summed into Fifths

As indicated in (Suit et al. 2007), taking the same strategy in the team deathmatch as used in a pure deathmatch was a reasonably successful approach. A more highly tuned fitness function to take into consideration some of the exceptions to this strategy in team play is under development, and might produce even better results in the future.

Other Observations and Comments

Experimentation in both of the above cases showed little improvement in evolved bot performance past 25 or 30 generations. At that point in time there was simply not much genetic diversity left in the population.

To assess the general playing ability of the evolved bots once evolution showed little additional improvement, we played additional games with the fully evolved bots. In one scenario, we pitted the fully evolved bots against the same control group in a different Unreal Tournament 2004 level. In another scenario, we pitted the fully evolved bots against an entirely different control group in the same level in which evolution took place. In both cases, there was still a difference in fitness between the evolved and control groups, indicating that evolution still retained some of its benefits, but the difference was between 10 to 30% smaller than before, depending on the scenario. This suggests that evolution in this case is at least somewhat dependent on the context.

So, while bots can be evolved during game production using genetic algorithms for efficiency reasons, these bots will still require further online adaptation to become better suited to the individual player of the game. Improvements in fitness were observed after 10 to 15 generations, which might be acceptable to some players, but could be too long for others. As a result, we may need to accelerate the evolution process, perhaps by having multiple generations of bots in each game played, as opposed to only one generation per game. This possibility needs to be explored in further experimentation, as forcing evolution prematurely might not result in the improvements in bot performance desired.

It was also observed during experimentation that evolved bots almost universally maximized their accuracy trait. This makes sense, since improved accuracy in shooting at opponents only has benefits to the bots, without any negative consequences. While this might challenge a player, it could do so in a way that is rather frustrating, as a bot could succeed by making nearly impossible shots in a super-human fashion, while a human player could not possibly do the same regardless of their skill. Consequently, we are currently conducting further experiments that do not allow the accuracy trait to be adjusted, forcing bots to improve in other ways that could produce more rewarding gameplay to the player. Initial results are quite promising.

CONCLUDING REMARKS

With artificial intelligence becoming increasingly critical to the success of modern video games, it is important to study methods of improving non-player character behaviour in games to produce a more rewarding experience for the player. Our current work represents an important step in this direction, using genetic algorithms to evolve and adapt character behaviours.

This paper presents the results from our work, describing an Unreal-based prototype system for genetic evolution of Unreal bots, and presenting experiments conducted using Unreal Tournament 2004 to assess the suitability of genetic algorithms to improve game artificial intelligence. Results to date have been quite promising, encouraging further research in this area.

There are several possible directions for continued research in the future, including the following:

- Additional experimentation is clearly beneficial to further research in this area. The experiments presented in this paper only scratch the surface of what can be done using our prototype system. There are still many configuration options to be explored more fully, including the traits used during evolution, the fitness functions used, and the method used to select parents for generating offspring.

- User testing during experimentation is also important. So far, the success of evolved bots has been measured only in terms of their fitness. In the end, it is important to also determine if the evolved bots deliver a more enjoyable and satisfying experience to a human player.

- It is also important to study the use of our prototype system in other Unreal-based games. This may include porting our system to Epic's Unreal Engine 3.0, the most recent version of the engine in release.

- Applying our approach to games based on other game engines would also be interesting, and would provide additional platforms for further research, development, and experimentation in this area.

REFERENCES

Baillie-de Byl, P. 2004. *Programming Believable Characters for Computer Games.* Charles River Media.

Bourg, D. and Seemann, G. 2004, *AI for Game Developers*. O'Reilly Media Inc.

Buckland, M. 2002. "Genetic Algorithms in Plain English". *Available online at http://www.ai-junkie.com*. Last accessed June 2008.

Buckland, M. 2004. "Building Better Genetic Algorithms". *Appeared in AI Game Programming Wisdom 2.* Charles River Media.

Digital Extremes. 2004. *Unreal Tournament 2004 – Editor's Choice*. (August).

Epic Games. 2005. *Unreal Engine 2, Patch-level 3369*. (December).

Laramée, F. 2002. "Genetic Algorithms: Evolving the Perfect Troll". *Appeared in AI Game Programming Wisdom.* Charles River Media.

Laramée, F. 2004. "Advanced Genetic Programming: New Lessons from Biology". *Appeared in AI Game Programming Wisdom 2.* Charles River Media.

Rabin, S.. 2002. "Preface to AI Game Programming Wisdom" *Appeared in AI Game Programming Wisdom.* Charles River Media.

Russell, S. and Norvig, P. 2003. *Artificial Intelligence: A Modern Approach.* Second Edition. Pearson Education, Inc.

Spronck, P. and Ponsen, M. 2008. "Automatic Generation of Strategies". *Appeared in AI Game Programming Wisdom 4.* Charles River Media.

Sweetser, P. 2004. "How to Build Evolutionary Algorithms for Games". *Appeared in AI Game Programming Wisdom 2.* Charles River Media.

Suit, B. et al. 2007. "Unreal Tournament 2004/Team Deathmatch". *Appears in Strategy Wiki: The Free Strategy Guide and Walkthrough Wiki. Accessible online at: http://strategywiki.org/wiki/Unreal_Tournament_2004/Team_Deathmatch.* Last accessed June 2008.

Thomas, D. 2004. "The Importance of Growth in Genetic Algorithms". *Appeared in AI Game Programming Wisdom 2.* Charles River Media.

Thomas, D. 2006. "Encoding Schemes and Fitness Functions for Genetic Algorithms". *Appeared in AI Game Programming Wisdom 3.* Charles River Media.

Tozour, P. 2002. "The Evolution of Game AI." *Appeared in AI Game Programming Wisdom.* Charles River Media.