Gamebots Evolution for First Person Shooter (FPS) Games

Abhishek Reddy Yeruva
Department of Computer Science
University of New Mexico
reddyya@unm.edu

Introduction:

First Person Shooter (FPS) game genre focuses on the feel and perspective it gives to the player, the view and the feeling of the first person, just as if he was in the 3D world. In recent years, a significant rise in computing power and storage devices has generated enormous interest in data-driven approaches to studying all kinds of phenomena. Understanding and explaining social behavior can fundamentally alter the functioning of professional organizations, battlefield scenarios, and multi-player professional teams. The rapid advancement of data-driven techniques, combined with greater availability of data and greater capabilities in data storage have now allowed computers to better analyze and model social phenomena.

Typically, the FPS game shares common traits with other shooter games as weapons always used during the game plays which in turn fall under the heading of action game. The game 'Doom' was released on 1993and was considered as the trigger for the upcoming of FPS games. When Doom released, the main concern of the developer was the experience that the game engine gave to the player. Basically, the Artificial Intelligence (AI) bots that they created were simple, in the sense that they either run towards the player and attack or just walk around. The gaming patterns for the later released FPS games are almost similar, that is simple AI used during gameplay and it has never affected the gaming experience too much.

In the past decade, most of the gaming developers have spent their efforts and time in focusing on the design of 3D environment and details of the game engine in order to attract more gamers. For the non-player character (NPC) part, it has been taken lightly by the game creator, only a small amount of time and effort is used to create the AI. But later, some developers found that the gaming AI plays an important factor in order to the keep them continue their strike in the gaming industries especially for FPS game genre. As the demand for better AI bots is requested, the development on better and stronger gaming AI against human player has been crucial. The game developer may lose their clients if the AI used in the FPS game is too weak against human player as the gamer may feel boring after few plays. But, the clients also may not be interested if the AI bots used are too strong against the human players. Hence, this brings to the motivation of the FPS AI gaming research and the facts that how will the AI change the whole experience of playing the 3D FPS has been the recent concern.

This research work involves the discussions on the application of Evolutionary Programming (EP) used in evolving a set of parameters that act as the AI bots controllers in FPS game. In this work, there are two major discussions involved: (1) the proposed decision making structure used in this research, and (2) the Evolutionary Programming (EP) used during the optimization processes.

Motivation:

One of the major commercial applications of AI is in the rapidly expanding computer game industry. Virtually every game sold today has some sort of AI, from the "computer player" in a chess game to the machine-gun-toting enemies in a first-person shooter (FPS). Any virtual being that does not behave in a strictly pre-scripted manner has some sort of AI behind it. Sadly, however, the multi-billion dollar gaming industry has done very little to advance the field of AI. The industry has moved past the day when an AI opponent would not even react to the death of his teammate standing three feet away, but that does not mean these bots, as they are commonly called, have any serious thought processes.

A Visual History of 3D Game Engines [1]:

Doom might arguably was the most memorable (or at least the most popular) PC game of all time, and with good reason. Programmers found themselves in the stone age of game development. For the most part, building a game meant starting from scratch and compiling all new code, the advent of the game engine forever changed the PC gaming landscape.

It was Id Software's now legendary first-person shooter that pushed reusable 3D game engines as a viable programming model, and videogame development has never been the same since then. Id Software's Doom engine wasn't actually a true 3D engine at all, but a very well conceived two dimensional sector-based engine with 2D sprites representing objects, characters, and anything not tied down to the map. Because of this 2D limitation, rooms could not be stacked on top of one another, but this also allowed for faster rendering on the less powerful hardware of the time. All that was needed to run Doom was a 386 level PC (in low-detail mode) with a standard VGA video card capable of rendering texture-mapped environments.

Despite the underlying 2D nature, *Doom* was, and still is, considered a 3D title. Id created the illusion of 3D with height differences added separately to the environment, and later titles built around the Doom engine would even implement the ability to look up and down, with a distorted view.

Adding Smart Opponents To A First person Shooter Video Game Through Evolutionary Design [2]:

This paper demonstrates how a first-person shooter (FPS) video game can be made more fun and challenging by replacing the hard-wired behavior of opponents with behaviors evolved via an evolutionary algorithm. Using the open source first-person shooter game Cube as a platform, they replaced the agents (opponents) hard-wired behavior with binary “DNA” supporting a much richer variety of agent responses. Only those agents whose DNA allowed them to avoid being killed by the human player would continue on to the next "generation" (game). This was pretty much a property of survival-of the fittest. Mutating the DNA of the survivors provided enough variability in behavior to make the agent's actions unpredictable. The paper shows how this approach produces an increasingly challenging level of play, more fine-tuned to the skills of an individual human player than the traditional approach using pre-programmed levels of difficulty or simply adding more opponents.

Cube is an open-source FPS, written in C++, that is based on a very unorthodox engine that emphasizes simple 3D designs to provide a quickly and cleanly rendered environment. The Cube Engine is mostly targeted at reaching feature richness through simplicity of structure and brute force, rather than finely tuned complexity. The code for Cube is much shorter and simpler than that for many other FPS games.

The idea is that each of an agent's possible decisions are represented by a single value (true, false, or a probability) and all of these values combined determine its behavior. This string of values,

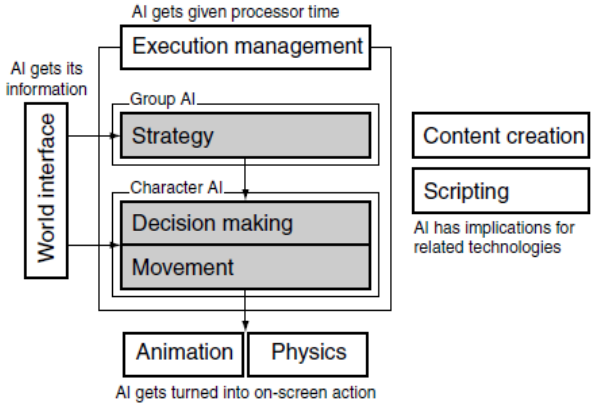
which they can call its DNA, is attached to an individual agent. Whenever the agent needs to make a decision, it consults both the usual criteria and the corresponding value in its DNA. (Strings are initially all zero at the start of the first game). At the end of a game, each agent is given a score based on how well it performed, and five of the ten agents are chosen based on their probability value to be "reborn" in the next game, by fitness proportionate selection. Two copies of each agent are passed on to the next game, but each of these is run through a mutation function that randomly alters a small fraction of the values in the DNA. This way, the agents are changing a little bit each game and the ones that perform the best will live to the next game. Just like biological evolution, this game became a survival-of-the-fittest environment where only the most well-adjusted agents survive and eventually, at least in theory, the agents become very good at surviving.

AI in Games [3]:

The AI in most modern games addresses three basic needs: the ability to move characters, the ability to make decisions about where to move, and the ability to think tactically or strategically. Even though we've gone from using state-based AI everywhere (they are still used in most places) to a broad range of techniques, they all fulfill the same three basic requirements.

Model of Game AI:

In this book there are a number of techniques. It would be easy to get lost, so it's important to understand how the bits fit together. They used a consistent structure to understand the AI used in a game. This isn't the only possible model, and it isn't the only model that would benefit from the techniques in this book. But to make discussions clearer, they will think of each technique as fitting into a general structure for making intelligent game characters. The following figure illustrates this model. It splits the AI task into three sections: movement, decision making, and strategy. The first two sections contain algorithms that work on a character-by character basis, and the last section operates on a whole team or side. Around these three AI elements is a whole set of additional infrastructure.



Not all game applications require all levels of AI. Board games like Chess or Risk require only the strategy level; the characters in the game (if they can even be called that) don't make their own decisions and don't need to worry about how to move.

On the other hand, there is no strategy at all in very many games. Characters in a platform game, such as *Jak and Daxter* [Naughty Dog, Inc., 2001], or the first *Oddworld* [Oddworld Inhabitants, Inc., 1997] game are purely reactive, making their own simple decisions and acting on them. There is no coordination that makes sure the enemy characters do the best job of thwarting the player.

Movement:

Movement refers to algorithms that turn decisions into some kind of motion. When an enemy character without a gun needs to attack the player in *Super Mario Sunshine* [Nintendo Entertainment,

Analysis and Development, 2002], it first heads directly for the player. When it is close enough, it can actually do the attacking. The decision to attack is carried out by a set of movement algorithms that home in on the player's location. Only then can the attack animation be played and the player's health be depleted. Movement algorithms can be more complex than simply homing in. A character may need to avoid obstacles on the way or even work their way through a series of rooms. A guard in some levels of *Splinter Cell* [Ubisoft Montreal Studios, 2002] will respond to the appearance of the player by raising an alarm. This may require navigating to the nearest wall-mounted alarm point, which can be a long distance away and may involve complex navigation around obstacles or through corridors.

Decision Making:

Decision making involves a character working out what to do next. Typically, each character has a range of different behaviors that they could choose to perform: attacking, standing still, hiding, exploring, patroling, and so on. The decision making system needs to work out which of these behaviors is the most appropriate at each moment of the game. The chosen behavior can then be executed using movement AI and animation technology.

At its simplest, a character may have very simple rules for selecting a behavior. The farm animals in various levels of the Zelda games will stand still unless the player gets too close, whereupon they will move away a small distance.

At the other extreme, enemies in *Half-Life 2* [Valve, 2004] display complex decision making, where they will try a number of different strategies to reach the player: chaining together intermediate actions such as throwing grenades and laying down suppression fire in order to achieve their goals.

Some decisions may require movement AI to carry them out. A melee (hand-to-hand) attack action will require the character to get close to its victim. Others are handled purely by animation or simply by updating the state of the game directly without any kind of visual feedback (when a country AI in *SidMeier's Civilization III* [Firaxis Games, 2001] elects to research a new technology, for example, it simply happens with no visual feedback).

Strategy:

You can go a long way with movement AI and decision making AI, and most action-based three dimensional (3D) games use only these two elements. But to coordinate a whole team, some strategic AI is required.

In the context of this book, strategy refers to an overall approach used by a group of characters. In this category are AI algorithms that don't control just one character, but influence the behavior of a whole set of characters. Each character in the group may (and usually will) have their own decision making and movement algorithms, but overall their decision making will be influenced by a group strategy.

In the original *Half-Life* [Valve, 1998], enemies worked as a team to surround and eliminate the player. One would often rush past the player to take up a flanking position. This has been followed in more recent games such as *Tom Clancy's Ghost Recon* [Red Storm Entertainment, Inc., 2001] with increasing sophistication in the kinds of strategic actions that a team of enemies can carry out.

Using a genetic algorithm to tune First person shooter bots [4]:

Commercial game developers are faced with the challenge of creating realistic, human-like artificial intelligence robot controllers (game AI) within tight hardware constraints. In terms of first-person shooters, rule-based expert system robot controllers which play the game are called 'bots'. Due to computational constraints, most bots are written in C/C++, taking the form of state-based machines. In order to save both computation and the programmer's time, the game AI uses many hard-coded parameters to complete the bot's logic. Authors of bots spend an enormous amount of time setting parameters. Parameters can be thought of as values which act as thresholds in the bot's rule-based logic.

The following figure shows an example of such parameters in terms of Counter Strike gameplay. Counter Strike is a popular first- person shooter game.

Counter Strike image with bots

By adding more parameterized rules, the bots become more realistic. Consequently, the development time increases since the programmer has more parameters to tune using trial-and-error. The tuning of these parameters becomes increasingly complicated even if the programmer is an expert in the game's strategy.

While there exists work in applying genetic algorithms to board games such as checkers and game theoretic problems like the iterated prisoner's dilemma, little work has been done within the scientific community in applying a genetic algorithm to a popular, 3D first-person shooter game like Counter Strike.

The paper proposes a technique which applies a genetic algorithm to the task of tuning these parameters. This will help game developers write game AI that is efficient, realistic, and easy to develop. The process of finding an acceptable set of parameters is referred to as tweaking the bot. In Counter Strike, this would be the time spent tuning the bot's weapon preference, initial aggressiveness, path preference, and style of gameplay. If a bot selected the same weapon round after round, then opponents would easily exploit the bot since no single weapon is perfect for every situation. Therefore, programmers give each weapon a preference of selection. This leads to a biased randomness in the bot's weapon selection behavior. The higher the preference for a weapon, the more likely it is to be purchased by the bot. Using relative preferences for weapon selection allowed for fast tuning of the bots, since a preference for a single weapon can be updated without having to update the preferences for other weapons. Weapon preferences are normalized to weapon selection probabilities during evaluation.

Choosing a correct set of parameters is not always a straightforward process and requires a great deal of trial-and-error testing. An acceptable parameter set can be thought of as the code to a safe, and in the case of Counter Strike bots, there are many combinations which will unlock the safe. In terms of search space, there are a number of acceptable parameter optima that will lead to good gameplay. Determining the correct set of parameters is a tedious and time-consuming one, since a slight change to one parameter will often have a negative impact on other parameters, i.e. they are non-linearly dependent. This is the motivation for applying a genetic algorithm to find a good set of these rule-based parameters.

Recently, the scientific community has taken an interest in using commercial 3D engines such as Quake, Unreal, and Half-Life as a testbed for advanced AI research. It is also believed that Counter Strike, a game which runs inside the Half-Life engine, to a be a good testbed for AI research. They use a Soar AI Engine, a rule-based expert system, to design bots that play Quake II, another popular first-person shooter game. The Soar engine is re-useable from game to game within a specific genre, requiring only changes to the engine calls and not to the AI logic inside of the Soar Engine. The authors have developed a framework inside the Unreal Tournament engine which allows them to study AI behavior within the virtual environment provided by the engine. They have also developed a non-violent game in

order to attract more researchers, who would otherwise be turned away by the extreme violence found in most first-person shooters.

The Pogamut Platform Project [5]:

Many research projects oriented on control mechanisms of virtual agents in videogames have emerged in recent years. However, this boost has not been accompanied with the emergence of toolkits supporting development of these projects, slowing down the progress in the field. Here, the paper presents Pogamut 3, an open source platform for rapid development of behavior for virtual agents embodied in a 3D environment of the Unreal Tournament 2004 videogame.

The development of control mechanisms for videogame agents (that is, their "artificial intelligence") is hard by itself. The developer's work is typically further complicated by the fact that many tedious technical issues, out of the main scope of their work, have to be solved. For instance, they have to integrate their agent within a particular 3D environment, develop at least simple debugging tools or write a code for low-level movement of their agents. This list may continue for pages and developers address these issues over and over, in most cases a waste of energy and time. For last four years, they have been developing the Pogamut toolkit, which provides general solutions for many of these issues, allowing developers to focus on their main goals. A new major version, Pogamut 3, has been released. Importantly, Pogamut 3 is designed not only for advanced researchers, but also for newcomers. Pogamut 3 can help them to build their first virtual agents, a feature which makes it applicable as a toolkit for training in university and high-school courses. The purpose of this paper is to review the main features of Pogamut 3, its architecture, extensions, and work in progress.

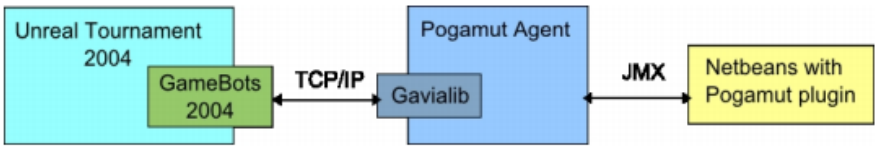
Features of Pogamut 3:

The agent development cycle can be conceptualized into five stages: (1) inventing an agent, (2) implementing the agent, (3) debugging the implemented agent, (4) tuning the agent's parameters, and (5) validating the agent through series of experiments. Individual features of Pogamut 3 were purposely designed to provide the support during the last four stages.

The features of Pogamut 3 include:

1) a binding to the virtual world of the Unreal Tournament 2004 videogame (UT2004),
2) an integrated development environment (IDE) with a support for debugging,
3) a library with sensory-motor primitives, path-finding algorithms, and a support for shooting behavior and weapon handling,
4) a connection to the POSH, which is a reactive planner for controlling behavior of agents, and a visual editor for POSH plans,
5) a support for running experiments, including distributed experiments running on a GRID.

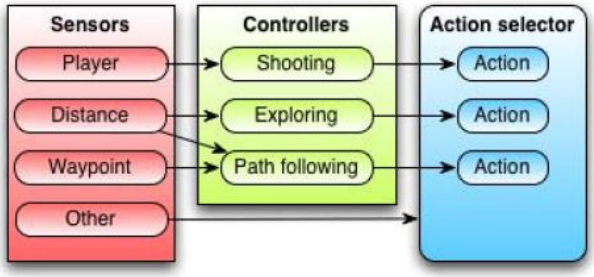
Technically, the Pogamut 3 toolkit integrates five main components: UT2004, GameBots2004, the GaviaLib library, the Pogamut agent, and the IDE. These components implement the features mentioned above, enabling a user to create, debug, and evaluate his or her agents conveniently. In general, the components of the Pogamut 3 toolkit can be used per partes. For instance, one can use GameBots2004 independently as well as a part of the GaviaLib library or the IDE.



Pogamut Architecture

Hierarchical Controller Learning in a First-Person Shooter [6]:

This paper describes the architecture of a hierarchical learning-based controller for bots in the First-Person Shooter (FPS) game Unreal Tournament 2004. The controller is inspired by the subsumption architecture commonly used in behavior based robotics. A behavior selector decides which of three sub-controllers gets to control the bot at each time step. Each controller is implemented as a recurrent neural network, and trained with artificial evolution to perform respectively combat, exploration and path following. The behavior selector is trained with a multi-objective evolutionary algorithm to achieve an effective balancing of the lower-level behaviors. The paper argues that FPS games provide good environments for studying the learning of complex behaviors, and that the methods proposed here can help developing interesting opponents for games.



Hierarchical Structure

First-person shooter games are three-dimensional combat simulation games which are viewed from a first-person perspective, and where the player is tasked with surviving in an adversarial environment through winning firefights with other agents. In many FPS games the player takes control of an armed soldier on a battlefield, which might simulate aspects of historical battles (e.g. the Call of Duty series) or science fiction scenarios (e.g. the Halo series), fighting against enemies such as other soldiers or monsters. Playing an FPS well requires mastering a number of related but distinct skills. To begin with, there are the lower level perceptual and motor skills, such as quickly identifying that something moved on a part of the screen, identifying what it was (friend or foe?) and reacting appropriately (e.g. aiming at the moving object, or backing away). Intermediate level skills require simple planning and include deciding in what order to attack enemies when several are present in a room, selecting appropriate weapons for the current battle, and finding and moving to good positions from where the player can take aim at enemies without needing to worry about being attacked from behind. Higher level "cognitive" skills are concerned with creating a complex representation of the environment and include mapping the area the player is moving in, keeping track of the positions of health-packs, ammunition supplies and enemies, and planning where to explore and what resources to gather before particular battles.

Tools for Computational Intelligence [7]:

Several computational analytic tools have matured in the last 10 to 15 years that facilitate solving problems that were previously difficult or impossible to solve. These new analytical tools, known collectively as *computational intelligence tools*, include artificial neural networks, fuzzy systems, and evolutionary computation. They have recently been combined among themselves as well as with more traditional approaches, such as statistical analysis, to solve extremely challenging problems. Diagnostic systems, for example, are being developed that include Bayesian, neural network, and rule-based diagnostic modules, evolutionary algorithm-based explanation facilities, and expert system shells. All of these components work together in a "seamless" way that is transparent to the user, and they deliver results that significantly exceed what is available with any single approach.

At a system prototype level, computational intelligence tools are capable of yielding results in a relatively short time. For instance, the implementation of a conventional expert system often takes one to three years and requires the active participation of a "knowledge engineer" to build the knowledge and rule bases. In contrast, computational intelligence (CI) system solutions can often be prototyped in a few weeks to a few months and are implemented using available engineering and computational resources. Indeed, computational intelligence tools are capable of being applied in many instances by "domain experts" rather than solely by "computer gurus." This means that biomedical engineers, for example, can solve problems in biomedical engineering without relying on outside computer science expertise such as is required to build knowledge bases for classical expert systems. Furthermore, innovative ways to combine CI tools are cropping up every day. For example, tools have been developed that incorporate knowledge elements with neural networks, fuzzy logic, and evolutionary computing theory. Such tools are quickly able to solve classification and clustering problems that would otherwise be extremely time consuming using other techniques.

The concepts, paradigms, algorithms, and implementation of computational intelligence and its constituent methodologies-evolutionary computation, neural networks, and fuzzy logic—are the focus of this book. In addition, they emphasize practical applications throughout; that is, how to apply the concepts, paradigms, algorithms, and implementations discussed to practical problems in engineering and computer science. This emphasis culminates in the real-world case studies in a final chapter available on the Web. Computational intelligence is closely related to the field called "soft computing." There is, in fact, a significant overlap.

Soft computing is not a single methodology. Rather, it is a collection of computing methodologies which collectively provide a foundation for the conception, design and deployment of intelligent systems. The principal members of soft computing are fuzzy logic (FL), neuro-computing (NC), genetic-computing (GC), and probabilistic-computing (PC). In contrast to traditional hard computing, soft computing is tolerant of imprecision, uncertainty and partial truth. The guiding principle of soft computing is: "Exploit the tolerance for imprecision, uncertainty and partial truth to achieve tractability, robustness, low solution cost and better rapport with reality."

The author believes that soft computing is serving as the foundation for the emerging field of computational intelligence. They believe that soft computing is a large subset of computational intelligence. They heartily agree with the author when he says, "Hybrid intelligent systems are definitely the wave of the future" (Zadeh 1994). The extensive rewrite and reorganization of that material reflect the authors change in perception of computational intelligence that has occurred over the years. That change is reflected in an increased emphasis on evolutionary computation as providing a foundation for Computational Intelligence. It also features significant recent developments in particle swarm optimization and other evolutionary computation tools.

Evolutionary Programs for Finite State Machines [8]:

This paper focuses on experiments with evolutionary programming. In particular, self-adaptive parameters that provide information on the generation of offspring represented as finite state machines are incorporated into evolving solutions and are simultaneously subjected to mutation and selection. Such operations have been applied to real-valued function optimization problems, but can be extended to problems in discrete combinatorial optimization. The paper begins with background on the use of self-adaptation in evolutionary computation. The results of experiments comparing the efficiency of two self-adaptive methods on finite state machines for time series prediction are described. Finally, potential avenues for further investigation are discussed.

Evolutionary programming was first thought as an alternative method for generating artificial intelligence. Experiments were offered in which finite state machines were used to predict time series with respect to an arbitrary payoff function. Mutations were imposed on the evolving machines such that each of the possible modes of variation were given equal probability. The current study investigates the use of self-adaptive methods of evolutionary programming on finite state machines. Each machine

incorporates a coding for its structure and an additional set of parameters that determine in part how it will distribute new trials. Two methods for accomplishing this self-adaptation are implemented and tested on two simple prediction problems. The results appear to favor the use of such self-adaptive methods.

In these simulations, a population of abstracted chromosomes are modified via operations performed by crossover, inversion and simple point mutation. An external selection criterion (objective function) is used to determine which chromosomes to maintain as parents for successive generations. These procedures have come to be termed *genetic algorithms*. Alternatively, Fogel (1962, 1964), offered methods for simulating evolution as a phenotypic process, that is, a process emphasizing the behavioral link between parents and offspring, rather than their genetic link. These simulations also maintain a population of abstracted organisms either as individuals or species but emphasis is placed on the use of mutation operations that generate a continuous range of behavioral diversity yet maintain a strong correlation between the behavior of the parent and its offspring. These methods are known as *evolution strategies* and *evolutionary programming*, respectively.

Inexpensive techniques for various games [9]:

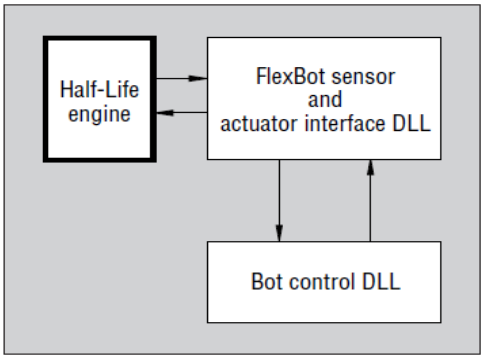
Modern computer games are highly sophisticated in their simulation of an artificial game world. They present the player with beautifully rendered environments, often accompanied by a complex and consistent world physics model. Increasingly, these games also use intelligent characters that can help or hinder the player and interact with the world and each other. Developers are using artificial intelligence to develop more engaging games, and the complexity of this game AI has become an increasingly important selling point. However, developing these intelligent systems is a complicated endeavor. Modern computer games impose tight computational constraints: graphics and world physics simulations consume most of a CPU's cycles, leaving only a small fraction available for the AI subsystem to control game agents. Some genres (such as first-person shooters) feature highly dynamic environments that update many times per second, requiring the AI system to be highly responsive to unexpected events. Thus the AI engine must continuously manage a potentially large number of agents while using only a small fraction of processor time.

The approach concentrates on using inexpensive techniques to develop computer game characters. They used two such techniques to develop agents for deathmatch games in the first-person shooter genre. The first system called *Groo*, engages in intelligent tactical behavior using a fairly simple and static behavior network. The second system called *tt14m*, uses simple text processing to attempt engagement in the social aspects of the game Counter-Strike. The techniques used in both systems are computationally inexpensive and easy to implement, which addresses the constraints found in most game development.

The *Groo* project attempts to create an efficient *bot*, that plays a first-person-shooter death match game in a tactically intelligent manner. The objective was to model an average death match player's basic tactical skills. Researchers on the Quake Bot project performed some related work in this area, with focus on improving the bot's abilities by incorporating predictive capabilities and learning. The project places heavier emphasis on efficiency of simple mechanisms. As mentioned earlier, the typical modern game leaves little time for AI processing, so game AI must be efficient. Their hypothesis is that a reactive system based on behavior-based techniques can model an average first-person-shooter player's skills while still being efficient enough for game developers.

First, existing AI techniques that implement complex cognition are computationally expensive relative to available CPU time. Traditional symbolic reasoning systems let you manipulate arbitrarily sophisticated representations but require highly serial computations operating on a large database of logical assertions. Updating and then reasoning over a symbolic knowledge base for a complex and highly dynamic environment in the small remaining time slice is a difficult challenge.

Second, a recent case study seems to suggest that for bots in a first-person shooter game, decision time and aim are key to the perception of skill. The former quality is directly tied to our focus on efficient AI processing; the latter is not a function of higher level cognition.



Flexbot Architecture Design

Reinforcement Learning (RL) in shooter games [10]:

Modern computer games, and particularly FPS's, are increasingly becoming more complex. Although games have complex environments, the artificial intelligence algorithms do not need complete knowledge to act intelligently. This paper will show that a small subset of the environment can be known by the bot and interesting behaviors can still be produced. FPS bot AI generally consists of path finding, picking up, and using objects in the environment, and different styles of combat such as sniper, commando, and aggressive. Bot AI in current commercial games generally uses rule-based systems, state machines, scripting, and goal-based systems. These techniques are typically associated with a number of problems including predictable behaviors, time-consuming fine tuning of parameters, and the need to write separate code for different creature types and personalities. Although goal-based systems are able to reuse code for base behaviors, they can become complicated when many different bot types are used, and can be hard to debug. Reinforcement Learning is an interesting and promising class of algorithms to overcome or minimize such problems due to its flexibility and ability to continue learning online. Predictability can be overcome by allowing a small learning rate during the game, allowing the AIs to slowly adapt to their surroundings. Parameter tuning can be minimized by automating experiments to produce different types of bots.

Reinforcement learning (RL) is a popular machine learning technique that has many successes in learning how to play classic style games. Applying Reinforcement Learning to first person shooter games is an interesting area of research as it has the potential to create diverse behaviors without the need to implicitly code them. This paper investigates the tabular Sarsa(λ) RL algorithm applied to a purpose built FPS game. The first part of the research investigates using RL to learn bot controllers for the tasks of navigation, item collection, and combat individually. Results showed that the RL algorithm was able to learn a satisfactory strategy for navigation control, but not to the quality of the industry standard path finding algorithm. The combat controller performed well against a rule-based bot, indicating promising preliminary results for using RL in FPS games. The second part of the research used pre-trained RL controllers and then combined them by a number of different methods to create a more generalized bot artificial intelligence (AI). The experimental results indicated that RL can be used in a generalized way to control a combination of tasks in FPS bots such as navigation, item collection, and combat.

Conclusion:

In this paper, we have discussed various methods and algorithms that are being used in the field of First Person Shooter games. Many properties have been looked into and also the various problems that

the authors faced during their experiments. However, the combination of the various evolutionary algorithms along with the proposed finite state machine mechanisms have proven to be useful in generating AI bots for First Person Shooter games.

References:

[1] P. Lily ”Doom to Dunia: A Visual History of 3D Game Engines.”, Retrieved 8 12, 2009, Maximum PC

[2] C. Overholtzer and S. Levy, “Adding Smart Opponents To A Firstperson Shooter Video Game Through Evolutionary Design” aaai.org, 2005.

[3] I. Millington. “Artificial Intelligence for games.”, Maugen Kaufman, 2006.

[4] N. Cole, S. J. Louis, and C. Miles, “Using a genetic algorithm to tune First person shooter bots.”, International Congress on Evolutionary Computation, pp. 139-145, 2004.

[5] Pogamut Platform Project, accessed 14 March 2011

[6] N. V. Hoorn, J. Togelius “Hierarchical Controller Learning in a First-Person Shooter.”, In IEEE Symposium on Computational Intelligence and Games (CIG), 2009.

[7] R. C. Eberhart. “Computational Intelligence PC Tools.”, Academic Press, London, 1st ed. ed. Boston, MA: Academic press professional, 1996.

[8] L. J Fogel, P. J. Angeline, and D. B. Fogel, “An Evolutionary Programming Approach to Self-Adaptation on Finite State Machine.”, Massachusetts: MIT Press, 1995, Page(s):355-365.

[9] A.Khoo and R.Zubek. “Applying inexpensive techniques to computer games.”, In *IEEE lntelligent Systems*, pages 2-7, 2002.

[10] M. McPartland and M.Gallagher, “Learning to be a Bot: Reinforcement Learning in Shooter Games,” presented at Artificial Intelligence in Interactive Digital Entertainment (AIIDE), Stanford, USA, 2008.