Evolving Parameters for Xpilot Combat Agents

Gary B. Parker
Computer Science
Connecticut College
New London, CT 06320
parker@conncoll.edu

Matt Parker
Computer Science
Indiana University
Bloomington, IN, USA
matparker@cs.indiana.edu

Abstract—In this paper we present a new method for evolving autonomous agents that are competitive in the space combat game Xpilot. A genetic algorithm is used to evolve the parameters related to the sensitivity of the agent to input stimuli and the agent's level of reaction to these stimuli. The resultant controllers are comparable to the best hand programmed artificial Xpilot bots, are competitive with human players, and display interesting behaviors that resemble human strategies.

Keywords: Xpilot, Genetic Algorithm, Control, Autonomous Agent, Xpilot-AI

I. Introduction

Xpilot is a two dimensional interactive combat game set in the space environment. We developed a system, called Xpilot-AI, that allows researchers to create their own agents and test them in the Xpilot arena. In previous work (details in Section 3), we used the system to evolve controllers while avoiding the use of predefined functions. Although our evolving agents learned and were capable of reasonable combat, they were no match for human players or for our own hand programmed agent. With the further development of Xpilot-AI, a function set that includes advanced features such as an aiming function was made available at www.Xpilot-AI.org. In order to evolve an agent controller suitable to enter the CIG2007 Xpilot competition, we determined that our best chance was to use whatever features were available. The results are reported in this paper.

There has been significant use of evolutionary computation to learn game play although much of this has been to learn strategies for playing board games such as checkers [1] and go [2] or for beating an opponent playing the prisoner's dilemma problem. However evolutionary computation has also been used to learn controllers for interactive games. Funes and Pollack evolved controllers for light-cycles against human opponents in their online Java Tron applet [3], Yannakakis and Hallam evolved interesting ghost opponents for the game Pac-Man [4], and other researchers evolved controllers for opponents operating in real-time combat strategy games such as the commercial games Counter-Strike [5] and Quake3 [6] and in games developed by the authors such as Lagoon [7]. Stanley et al used a method for evolving artificial neural networks to control agents that could learn in real-time through a series of training exercises in the NERO video game [8].

II. Xpilot-AI

Xpilot-AI was developed as a means for researchers in artificial intelligence to test methods of the generating autonomous controllers. It uses the interactive Internet game Xpilot as a training/testing environment.

Xpilot is an open-source 2-dimensional multiplayer space combat game. Fig. 1 shows a typical game in play. The large area on the right is called the screen; it gives the players a detailed look at what's happening near their own ship. The two triangles are ships engaged in combat. In this case, the one at the top is the ship being controlled by the operator running the client producing the graphic interface. The other triangle, with the label Sel_41913 is the opponent or enemy ship. The boxes on the screen area are walls. The numbers 1 2 3 show the location for 3 bases, one each for ships from teams 1, 2, and 3. The light dots behind Sel are the result of Sel activating its thruster. The brighter dot behind Sel is a bullet shot by the operators' ship, which just missed as Sel accelerated to avoid it. The top left of the graphic shows the entire map, with the ships shown as dots. This is referred to as the radar. Below it is a list of the players with their scores. Also listed are the teams and their scores. The player controls a ship using the keyboard or mouse and must destroy enemy ships while avoiding being killed by the enemy or a wall collision. There are different maps that vary in size and objectives, with team play, capture the flag, or free-for-all combat. There are often ship upgrades made available that enhance a ship's abilities with added weapons like lasers or triple fire, or with ship upgrades such as cloaking or increased fuel. At this point only simple maps, with only bullets provided as offensive weapons, have been used for Xpilot-AI. Xpilot uses a client/server approach to support multiple players. It has realistic physics and solid networking code.

Xpilot has a predetermined networking protocol, which made it a good platform for developing Xpilot-AI. The Xpilot server and client are synchronized frame by frame. For each frame, the server sends packets to the client with information about what to display. The client receives this information, parses it, and displays it graphically on the screen. The client sends back to the server information about mouse movement and what keys the player is pressing on the keyboard. Xpilot-AI parses the information received by the client and stores it in easily accessible variables. In addition, we have added modules with structures containing

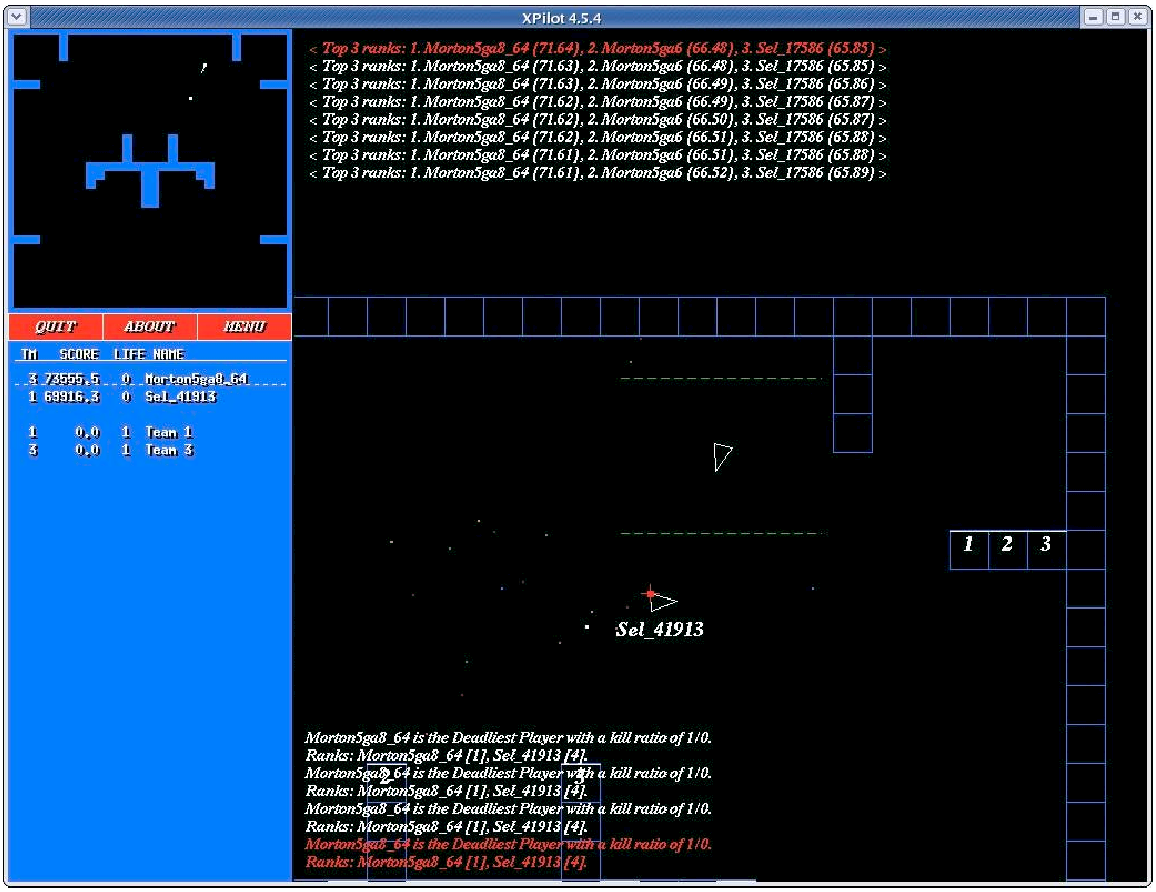

Fig. 1. Xpilot game in progress. Evolved agent Morton has just attempted to shoot the enemy ship Sel, but the shot went behind its target.

useful information about the player's ship, the enemy ships, the bullets, the radar, and the map. The structures contain information like X and Y coordinates, velocities, accelerations, distances, etc. The Xpilot-AI modules also include commonly used functions for aiming, calculating distances, finding walls, etc.

Xpilot-AI controls the ship by simulating keyboard key presses and mouse movement. To turn the ship, it sends a packet to the server saying the mouse has moved a certain amount. To thrust, shoot, and do other controls it sends a packet saying a particular key has been pressed on the keyboard. Between the sending and receiving, an AI function can be inserted that takes as inputs environment information and outputs the control for every frame. Because all this is done in the client, any number of AI controlled clients may be connected to play against one another or against real people. Xpilot-AI programs in C and Scheme are available online at www.Xpilot-AI.org.

III. Previous Xpilot Research

Autonomous Xpilot agents learning to combat a single agent have been evolved using four learning methods, all of which attempted to make minimal use of predefined functions. In all of these works, the agent being evolved was engaged in combat with a single opponent. Both the learning agent and the opponent had three possible outputs: turn up to 15° left or right, thrust on or off, and shoot on or off. The learned control program was to take in a number of inputs and determined these three outputs between each frame of the game.

A. Evolving Weights for a Perceptron [9]

In this research a genetic algorithm was used to learn the connection weights for a fixed artificial neural network controlling the agent as it engages in combat with a single opponent. The AI ship controller was a single layer feedforward neural network with twenty-two inputs and three outputs. The inputs included information about the agent, its enemy, bullets, the walls, and one was set to 1 to act as a threshold. The chromosome was made up of 66 genes, each of which was made up of six bits. Each gene represented a weight and was converted to a number between -1 and 1 before use in the Perceptron controller.

B. Evolving the Priorities and Responses of Production Rules [10]

A set of 16 conditions important for reasonable play was developed. This included conditions dealing with the ship's position relative to walls, the enemy ship, and hostile fire. A

binary coding was developed to represent the possible responses that were to be learned by the GA. In addition, the GA learned the priorities of theses different rules to resolve conflicts if more than one fired at the same time. The Xpilot agent was represented by a chromosome of 16 genes. Each gene was represented as a segment of 14 bits, with each gene containing information for how the agent should behave in each of 16 different cases which serve as antecedents for a rule based system.

C. Generating the Controller Program using a Cyclic Genetic Algorithm [11]

The cyclic genetic algorithms (CGA) is a type of genetic algorithm in which the genes represent tasks to be performed rather than traits of a problem solution. In addition, a group of these tasks can be set to repeat, which allows for cyclic behavior. The CGA was initially developed to generate the cycles required for hexapod robot walking gaits, which required only single cycles. In further work, the CGA has been extended to give it the capability of generating multi-loop control programs, which is how it was used in this application to evolve Xpilot controllers. These controllers were represented by a chromosome of 64 genes, with each gene being 11 bits, so a chromosome was comprised of 704 bits. Each 11-bit gene either contained information for how the robot should act in a given frame, or an instruction to jump to another gene to look for control information.

D. Incrementally Evolving a Multi-Layer Neural Network [12]

In this research we used incremental evolution to learn the neural network (NN) controllers. In the first increment we used specific training environments to learn specific facets of control. These were then used in the second increment to evolve a two layer NN that used a separate NN to control its second layer connection weights. Fig. 2 shows a diagram of the entire incremental network. The enemy distance and bullet alert are inputs to a network whose outputs are the weights between the specialized networks' outputs and the three resultant outputs. Although the agent learned each specific facet, the combination was not as competitive as hoped, plus it was significant work to evolve.

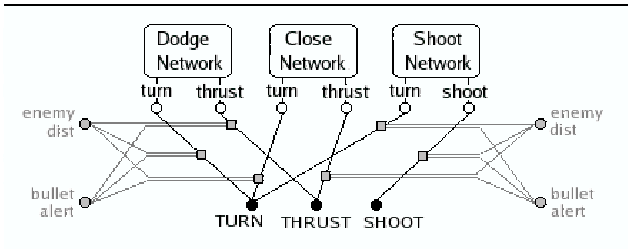

Fig 2. Incremental network evolved in previous work.

All of these attempts were successful in that reasonable agents were developed from random, but they were not good enough to compete with the standard human player.

IV. EVOLVING PARAMETERS

For the researcher reported in this paper we started with an agent controller made available at the Xpilot-AI web site. This agent, which the developers call Morton, has a hand coded controller that takes into account walls, bullets, and the enemy ship, but is still only moderately competitive. Our initial work was to expand this program to make it take into account different levels of wall collision danger, bullet danger, and to be able to consider more than one enemy and bullet at a time. In addition, we included cases where it could recognize if there was a wall between it and the enemy and/or bullet. The program was a large case statement made up of 18 cases, which were ordered in what we considered to be of greatest importance. For example, if a bullet was close and had a high probability of hitting our agent ship, this took priority over a moderate chance of wall collision. The turn and thrust setting of the agent for that frame was determined by the applicable case statement. To control whether the ship was to shoot, a separate if statement was used since turn and thrust are always of importance and shoot is purely offensive.

The program was originally written with the developers' best guess for the many parameters such as angle of wall feelers and the turn to be made in case of an approaching wall. This resulted in a reasonable agent but we found it difficult to find the correct set of parameters to make it competitive. Our next step was to determine all of the parameters needed to be set and their appropriate ranges. We extended these beyond what we thought appropriate in case our preconceived notions of what they should be were hindering us. We identified 22 parameters and determined that we could set them all using a range of 0 to 15. Some were multiplied, some were divided into plus or minus possibilities, while still others were added to previous parameters to set their parameter.

The set of parameters that were learned are shown in Fig. 3. Some of these parameters have to do with general use functions, such as "same", which was used to define what should be considered equal in reference to distances from the ship. Some of the parameters are for specific situations. The ship uses feelers to determine the distance to walls at angles set relative to its track. There were different feelers for different situations. One set of feelers was to be used early in the case statement, which signifies that these were for higher priority situations. The second set was only used when none of the prior cases were true. When the initial program was written we used the same angle for both of these sets of feelers, then later determined that it might be advantageous to have different angles dependent on the priority. Some of the parameters were used to determine when a case would be considered true (such as those dealing with feelers), others were used to determine what the action would be when the case true (such as those dealing with an angle required for the ship to turn to or an angle required before the ship would thrust or shoot).

- span – The angle between the line to a target location and the edge of the nearest wall. Used to determine how much the wall is blocking the ship from a bullet.
- offsetinc – indicates the increments used to determine the turn direction needed for a ship to avoid a wall.
- samespread – the difference allowed between the distances returned by 2 wall-feelers which would result in considering them the same.
- wall-span1 – the angle off of the ship's track used to feel for the closest wall.
- wall-span2 – same as wall-span1, except used for a second set of feelers that are checked in a later case statement.
- vd-bullet-dist – determines the bullet alert required to consider the bullet very dangerous.
- d-bullet-dist – same as vd-bullet-dist, except located in a later case statement (value of this parameter is added to vd-bullet-dist to determine the distance).
- vd-dodge-bullet-angle – the angle the ship will turn away from a bullet considered very dangerous in order to dodge it.
- d-dodge-bullet-angle – the angle the ship will turn away from a bullet considered dangerous in order to dodge it.
- close-wall-speed – this speed, in conjunction with the distance of the closest wall, determines if the ship shoot take action to avoid it.
- medium-wall-speed – same as close-wall-speed, except used in a later case statement.
- c-angle-before-thrust – the angle of the ship's heading away from the closest wall before the ship will thrust.
- m-angle-before-thrust – same as c-angle-before-thrust, except used in a later case statement.
- wall-avoid-angle – how small the angle has to be between the ship's heading and its desired track to avoid a wall before it will thrust.
- screen-thrust-speed – if the ship's speed is lower than this and it is turning to attack an enemy on the screen, it will thrust.
- radar-nothrust-speed - if the ship's speed is lower than this and it is turning to attack an enemy on radar, it will thrust.
- ship-error-to-shoot – the maximum angular difference between the desired aim direction and the ship's heading before it will shoot at an enemy on the screen.
- radar-error-to-shoot – the maximum angular difference between the desired aim direction and the ship's heading before it will shoot at an enemy on radar.
- wall-turn-angleR – the angle between the ship's track and heading that the ship turns to avoid colliding with a wall when responding to a right feeler indicating a wall that is too close.
- wall-turn-angleL – the angle between the ship's track and heading that the ship turns to avoid colliding with a wall when responding to a left feeler indicating a wall that is too close.
- wall-turn-angleB – the angle between the ship's track and heading that the ship turns to avoid colliding with a wall when responding to an equal distance from both walls.
- shoot-dir-rand – the angular range that the ship will use to randomly affect its direction to aim.

Fig. 3. Set of parameters that were evolved by the genetic algorithm.

V. Genetic Algorithm

A standard GA with a population of 128 chromosomes was used to learn the parameters, which we plugged into the agent while it competed in the Xpilot arena (Fig. 3). The chromosome used was 88 bits with 22 genes (parameters), each made up of 4 bits (allowing raw values from 0 to 15).

Selection was determined by putting the agent in the arena with a standard opponent. The opponent used, named Sel, is a hand-coded agent that up until this point has been our most successful combatant, which is competitive with a standard human player. It is available to compete against on the Xpilot-AI server, but its code is not available. The example code for an agent, Morton, which is given on the web page, is what we used to create our learning agent. Each chromosome of the genetic algorithm was used to control an agent that competed with Sel for one round, which ran until the death of the learning agent. If during the round, Sel died, the competitors went back to their starting stations and competed again. The learning agent gained one point of fitness for each frame that it survived and 1000 points for each time it killed Sel. After all 128 individuals were tested in the arena and assigned a fitness, the GA perform stochastic selection. Individuals selected were crossed over with a 100% chance of crossover and were mutated with a 1 in 300 probability of flipping each bit.

VI. Results

Five test runs using five randomly generated populations were run for 240 generations of training. At each generation the best and average fitness of the population was saved. The results are shown in figures 4 and 5. Fig. 4 shows the average fitness over time (generations) as the individuals evolved. Fig. 5 shows the fitness of the best individual of the population at each generation. The graphs show all these data points with different symbols used for each of the populations. In addition, fifth polynomial least squares trend lines are used to help visualize the growth of fitness. The fitnesses were highly variable since the starting locations for the ship and enemy were changed before each generation. However, generalizations about the learning progress can be made. There was significant improvement as the evolution took place. In the case of the average fitnesses, they started out at below 200 and rose to above an average of 700. In the case of the best individual at each generation, the fitness started out at approximately 1500 and rose to be over 5000.

These empirical results were backed up by observations of the agents as they engaged in combat. One could observe significant improvements as the agents evolved. Most started out with a slow spinning in place motion for a slight drift until the enemy approach, at which time they would take some action in response, which would often result in a wall collision. Others turned left and then right as they darted in a somewhat random manner on the map. They would usually shoot, but not necessarily in the direction of the enemy. While watching several individuals in the population positive characteristics for many were observed. Some had a tendency to be good at avoiding wall collisions, others reacted well to approaching bullets, and still others tended to shoot in the general direction of the enemy. These positive attributes are probably what the GA built on to

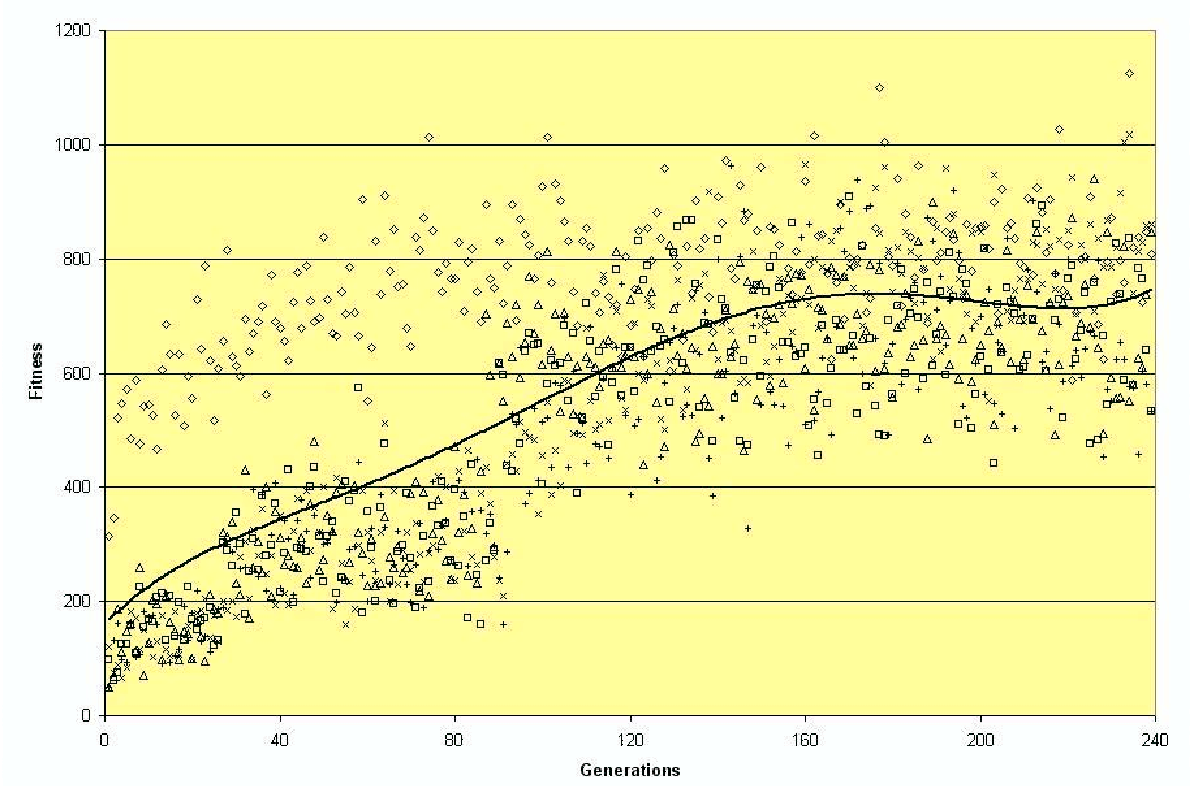

Fig. 4. Average of the fitnesses of all individuals in the population at each generation during training. Five tests are shown. A fifth order polynomial least squares trend line helps to show the learning curve.

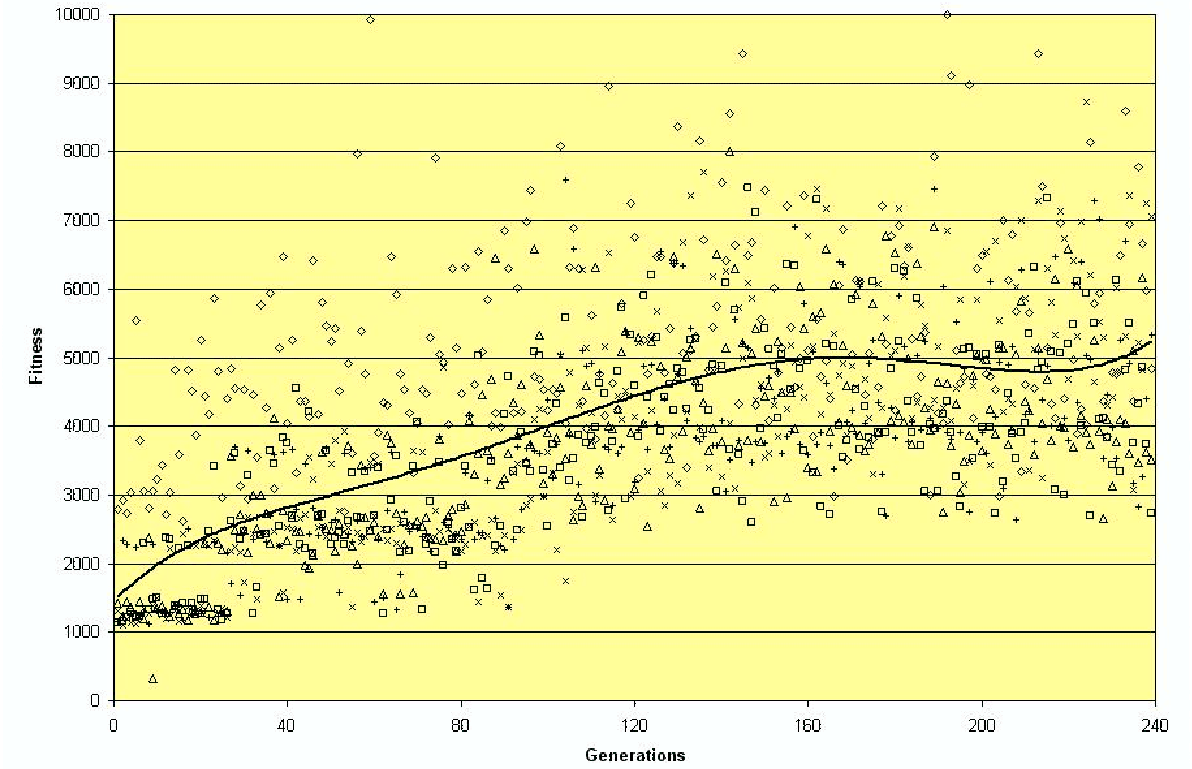

Fig. 5. The fitness of the best individual in the population at each generation during training. Five tests are shown. A fifth order polynomial least squares trend line helps to show the learning curve.

make competent combat agents. As the generations progressed, the individuals started to get better at each of the positive attributes and, in some cases, combine useful skills.

By the end of training, all 5 tests produced excellent autonomous controllers that gave their agents skills comparable to that of Sel. In some aspects they were better and in others not as good. Sel is equipped with very good aiming and chasing capabilities. It is programmed to maneuver around a wall and quickly/accurately attack its enemy. The evolved agents were not as aggressive (due to the fitness function) and had to learn their parameters for accurate aiming. Sel is also pre-programmed with good angles for dodging bullets and avoiding wall collisions; the evolved agents needed to learn these. Nevertheless, these agents learned to compete with Sel bot. This could be observed as the training progressed by looking at the Xpilot scores of each of the combatants. By the end of training the scores for Sel and the training agent are consistently within 1% of each other. Sometimes Sel is ahead and sometimes the evolving agents. This shows that the population of agents, or average performance of the population's individuals, is equal to that of Sel.

In addition to competing with Sel, the trained agents are very competitive with human players. Tests were done with the fitness evaluations being done on the Xpilot-AI server. Human players could join the game and compete against both Sel and the individuals evolving in the population. This makes for interesting competition since both Sel and the agents (added to the publicly accessible game after several generations of training) are very good and far superior to the Xpilot game supplied bots. It makes for particularly interesting combat because Sel is very consistent in its behaviors, but the bot representing the evolving agents varies slightly each time the game restarts (which happens after all but one have died) because all individuals of the population are being tested using it. We have had several human players join the game and in most cases, they are not as good as our evolved bots.

Trained individuals in the later generations start to develop some very interesting behaviors. Although they are controlled only by a set of simple cases, the behaviors they exhibit make them appear to have complicated controllers equipped with combat strategies. Due to the simplicity of the controller, we consider these to be emergent behaviors. One such behavior is the agent's apparent use of walls to shield it from the enemy. As an enemy approaches, an agent using this strategy will move in a direction to place the wall in between it and the enemy. It will then make quick moves out to the edge of the wall, just clearing it enough to take a shot and the then return for cover. This strategy is not only effective but gives the agent a human controlled appearance.

VII. Conclusions and Future Work

Learning controllers for Xpilot agents using the evolution of decision and action parameters can produce excellent controllers that are competitive with human players. The agents produced are not only competitive, but learn emergent behaviors that make them appear to use complex combat strategies. In addition, by using a publicly accessible Xpilot-AI server to continue evolution after several generations of initial training, human players can compete against seemingly non-deterministic bots that will adapt (given sufficient time) to their style of play.

In future work, we will continue to build the complexities of the controller to allow it to take on additional tasks such as base capture. In addition, experiments will be done with alternate fitness functions to produce different behaviors. The agents evolving in this paper were fairly passive; reducing the reward for kills by how long it takes to attain them should produce more aggressive resultant agents. The learning method presented in this paper, which produces competitive combat agents, greatly opens up the possibilities for future expansions of the use of evolved agents in the Xpilot environment.

References

[1] Fogel, D. Blondie24: Playing at the Edge of AI, Morgan Kaufmann Publishers, Inc., San Francisco, CA., 2002.

[2] Konidaris, G., Shell, D., and Oren, N. "Evolving Neural Networks for the Capture Game," Proceedings of the SAICSIT Postgraduate Symposium, Port Elizabeth, South Africa, September 2002.

[3] Funes, P. and Pollack, J. "Measuring Progress in Coevolutionary Competition," From Animals to Animats 6: Proceedings of the Sixth International Conference on Simulation of Adaptive Behavior. 2000.

[4] Yannakakis, G. and Hallam, J. "Evolving Opponents for Interesting Interactive Computer Games," Proceedings of the 8th International Conference on the Simulation of Adaptive Behavior (SAB'04); From Animals to Animats 8, 2004, pp 499-508.

[5] Cole, N., Louis, S., and Miles, C. "Using a Genetic Algorithm to Tune First-Person Shooter Bots," Proceedings of the International Congress on Evolutionary Computation 2004 (CEC'04), Portland, Oregon, 2004, pp 139–145.

[6] Priesterjahn, S., Kramer, O., Weimer, A., and Goebels, A. (2006). "Evolution of Human-Competitive Agents in Modern Computer Games." Proceedings of the 2006 IEEE Congress on Evolutionary Computation (ECE 2006), Vancouver, BC, Canada, July 2006.

[7] Miles, C. and Louis, S. (2006). "Towards the Co-Evolution of Influence Map Tree Based Strategy Games Players." Proceedings of the 2006 IEEE Symposium on Computational Intelligence and Games (CIG 2006).

[8] Stanley, K., Bryant, B., and Miikkulainen, R. (2005). "Evolving Neural Network Agents in the NERO Video Game." Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games (CIG 2005).

[9] Parker, G., Parker, M., and Johnson, S. (2005). "Evolving Autonomous Agent Control in the Xpilot Environment," Proceedings of the 2005 IEEE Congress on Evolutionary Computation (CEC 2005), Edinburgh, UK., September 2005.

[10] Parker, G., Doherty, T., and Parker, M. (2005). "Evolution and Prioritization of Survival Strategies for a Simulated Robot in Xpilot," Proceedings of the 2005 IEEE Congress on Evolutionary Computation (CEC 2005), Edinburgh, UK., September 2005.

[11] Parker, G., Doherty, T., and Parker, M. (2006). "Generation of Unconstrained Looping Programs for Control of Xpilot Agents," Proceedings of the 2006 IEEE Congress on Evolutionary Computation (CEC 2006), Vancouver, BC, Canada, July 2006.

[12] Parker G. and Parker M. (2006). "The Incremental Evolution of Attack Agents in Xpilot," Proceedings of the 2006 IEEE Congress on Evolutionary Computation (CEC 2006), Vancouver, BC, Canada, July 2006.