Computer Science Faculty Publications                    Computer Science Department

6-2011

# Fitness Biasing for Evolving an Xpilot Combat Agent

Gary Parker
*Connecticut College*, parker@conncoll.edu

Phil Fritzsche

# Fitness Biasing for Evolving an Xpilot Combat Agent

# Fitness Biasing for Evolving an Xpilot Combat Agent

Gary Parker and Phil Fritzsche

Computer Science

Connecticut College

New London, CT, USA

parker@conncoll.edu, pfritzsche@gmail.com

*Abstract*—**In this paper we present an application of Fitness Biasing, a type of Punctuated Anytime Learning, for learning autonomous agents in the space combat game Xpilot. Fitness Biasing was originally developed as a means of linking the model to the actual robot in evolutionary robotics. We use fitness biasing with a standard genetic algorithm to learn control programs for a video game agent in real-time. Xpilot-AI, an Xpilot add-on designed for testing learning systems, is used to evolve the controller in the background while periodic checks in normal game play are used to compensate for errors produced by running the system at a high frame rate. The resultant learned controllers are comparable to our best hand-coded Xpilot-AI bots, display complex behavior that resemble human strategies, and are capable of adapting to a changing enemy in real-time.**

*Keywords-Xpilot; Xpilot-AI; Anytime Learning; Punctuated Anytime Learning; Fitness Biasing; Genetic Algorithm; Real-time Learning; Video Game Learning; On-line Learning*

## I. INTRODUCTION

The objective of this research is to develop a robust method for real-time learning in the Xpilot-AI game environment. An important issue faced by artificially intelligent systems is that they are limited in effectiveness without a method for being adaptable. They can often perform well in specific situations, but do not have the ability to cope when an unanticipated element is introduced. Though the systems may be intelligent, their intelligence is only applicable to these specific situations. This issue makes the use of many such systems impractical in the real, or even virtual worlds, simply because of the unpredictable nature of any complex environment. It is impossible to predict everything that will happen in real life, so writing a program that can only work in specific situations can easily be rendered ineffective by a simple change that was not predicted.

Of particular interest in the gaming world is the ability to produce human-like agents for opponents. One characteristic of humans is their ability to learn and adapt to the manner of play of others in the game. This is a large part of what makes video games enjoyable. Creating artificially intelligent systems that are capable of doing this in real-time is an important aspect, as well as one of the most difficult aspects, in game development [1].

To conduct research in methods for addressing this issue, Xpilot-AI, a robust testing environment for autonomous agent learning, was selected. Video games in general and Xpilot in specific provide a rigorous testing environment for artificial intelligence research. Its complex nature has the capability to test agent control programs in different scenarios made possible by the variable parameters of the game.

In order to test methods for providing agent learning in real-time in an environment, it is of benefit to first show that a static learning system can produce a controller that behaves intelligently. This has been done in the Xpilot environment using genetic algorithms (GAs) to evolve parameters for an expert system [2]. In this system, an expert system-controlled agent was modified to work with a GA. The algorithm evolved optimal parameters for the rules of the expert system. Other methods have also been used to demonstrate learning in Xpilot. For example, a genetic algorithm was used to evolve the weights for a neural network-controlled agent [3] and a cyclic genetic algorithm was used to directly evolve a control program for an agent [4]. These learning systems, while useful, learned control behaviors before the agent was active and had no means of adapting to changes in capabilities or the enemy's behavior.

Though these methods used a genetic algorithm to learn an effective controller, the final products are lacking the ability to adapt to changes in enemy behavior in real-time. In previous research, dynamic programming-based reinforcement learning techniques, such as Q-learning, were used to produce an Xpilot agent capable of real-time learning, but the controllers learned were for a very simple environment. The Q-learning method implemented requires an accurate model to be successful. Since the Xpilot combat environment is very complicated, this method was applied only to a single agent in a simple environment with no opponents [5]. Though it was a successful implementation of real-time learning, it was determined to not be as scalable as desired. As the complexity of the environment increases, such a system's ability to cope rapidly deteriorates. One possible solution for this issue is described by Lucas [6]. DynaQ, a form of Q-learning, updates the agent's model of the environment as it explores. Doing so could allow an agent to more adeptly adapt to the large and constantly changing Xpilot environment. In another attempt to demonstrate real-time learning in Xpilot, evolutionary strategies were used to learn agent controllers [7]. Though capable of real-time learning, their reliance on mutations of a chromosome to evolve led to slower learning and a lack of effectiveness.

For other games, a number of different strategies have been used to attempt real-time learning. For the DEFCON computer game, researchers applied decision-tree learning and case-based reasoning combined with simulated annealing methods with the intention of creating human-like behavior [8]. Others applied evolutionary techniques to neural networks by starting

with simple networks then slowly adding nodes and connections while the game is running to make the agent learn increasingly complex behavior in real-time [9]. While both of these do learn in real-time, they also both rely on past knowledge and pre-defined courses of action. Changes are still made even after the game is over. Others have used a genetic algorithm approach, attempting to learn competitive human-like behavior in the video game Quake [10].

While useful, none of the above systems were able to fully solve the problem of creating a real-time learning system in the interactive game environment. Although genetic algorithms have been used successfully to generate controllers for interactive game environments, they have not been used in a real-time learning system since they require each individual of the population to be tested. In order to be able to adapt to an opponent's play in real-time, these tests would have to be done by playing the opponent. In addition, the genetic algorithm typically starts with a random population of solutions. This would make particularly poor play for someone playing the game. What is needed is to be able to do the learning on a model of the game with periodic checks to make sure the model represents the actual play.

In the study of learning in robotics, systems have been created to solve the issue of real-time learning with genetic algorithms. One such example is Anytime Learning (AL) developed by Grefenstette and Ramsey [11]. AL places a learning module in a robot and uses an observer module to learn from the robot's environment. The information gathered is used to influence the learning processes such that there is a link between the actual robot and environment to the simulation. As the learning system is on-board the robot, it has the potential to learn indefinitely. As long as the robot is running, it will continue to attempt to improve the controller.

Anytime Learning worked well for robots with the capability to carry the learning system on the robot. However, this is not always practical. For example, if the robot's environment is highly dangerous, it may be more practical to use several less expensive and expendable robots as opposed to one to complete the mission. Rather than having each robot carry an expensive on-board learning system, it would be more practical to have one off-board learning system used for all of the robots. One other issue with anytime learning is that the observer module had to recognize and categorize changes in the environment, a task that requires extensive computation.

Punctuated Anytime Learning (PAL), which was originally developed for evolving robot controllers [12], is a modification of AL that was developed to address these issues. In this paper, PAL is used to create a real-time learning system for control of agents in Xpilot-AI.

## II. XPILOT-AI

Xpilot is an open source multiplayer two-dimensional space combat game consisting of two main components: the server and the client (Figure 1).

The server controls global settings, such as the number of frames per second in a game and the map being used. It also keeps track of the players playing the game, their scores, and other information. The client is what the users control to play
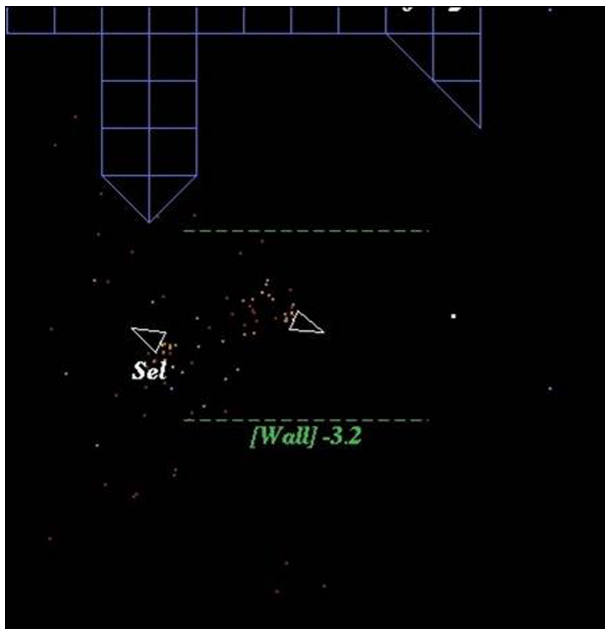


Figure 1. Two agents face off in Xpilot

the game. Specific keystrokes allow the users to control their ship by thrusting, turning, or shooting.

The game itself contains relatively realistic physics. Going too fast into a wall will cause the agent to explode, while running into a wall at a very slow speed while cause the agent to both bounce off the wall as well as decrease their speed slightly. As it is set in space, gliding without thrusting causes the agent to move with constant velocity.

Xpilot-AI is an add-on to the Xpilot game that allows users to write scripts to control the Xpilot agents. These scripts can be used to learn intelligent behavior and as a result, Xpilot-AI has become a powerful testing ground for researchers of artificial and computational intelligence. Agents controlled by scripts can play along with other scripts, humans, or server-controlled robots on any standard Xpilot server.

This provides an interesting opportunity to test autonomous agents by running them against both other computer-learned agents as well as human players. Seeing the agent in action in both environments can not only increase our understanding of what behavior is being learned but also make sure that it is able to compete against all types of opponents.

## III. FITNESS BIASING

Fitness Biasing is one of two methods of Punctuated Anytime Learning, which is a type of Anytime Learning. Anytime Learning as a system, described for evolutionary robotics by Grefenstette and Ramsey [11], is split up into two major components: the execution system and the learning system (Figure 2). The execution system handles everything essential to running the agent in its environment. It runs the decision maker, which is responsible for deciding how the agent should react in a given situation. The knowledge base acts as a current strategy for the decision maker as it instructs the agent to execute actions in its environment. Also in the execution system is a monitor, which is responsible for
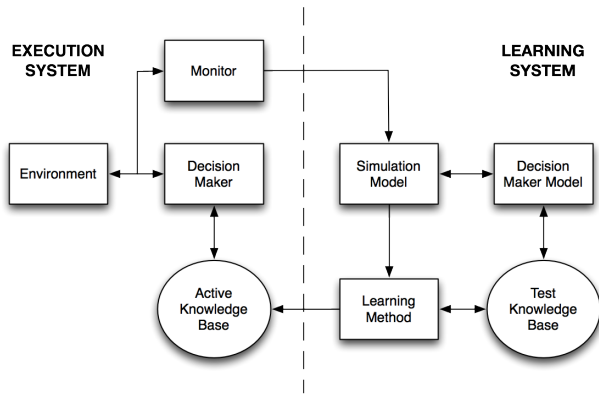
Figure 2. Anytime Learning System (replicated from Grefenstette and Ramsey [10])

gathering information about the agent's environment. It identifies and remembers changes in the environment, then sends this information to the learning system.

The learning system uses this information to alter the simulation model to more accurately represent the actual environment. A decision maker also works in simulation to facilitate the learning process. It relies on a test knowledge base, separate from the active one in the execution system. Based on the information gathered from the simulation, the learning method works to increase the robustness of the system and sends information to the active knowledge base whenever something new and improved is learned. One potential issue with this system is the monitor. Accurately and dynamically determining changes in the environment to update the simulation is a difficult task. Another issue with AL is that it requires the learning system to be onboard the robot. This is not always possible when it is desirable to have multiple less expensive robots perform the task.

PAL, an extension of AL, was developed to address these issues [12]. It requires the monitor to observe only the performance of the system and not attempt to observe what changes caused the differences. Rather than requiring the monitor to observe the entire environment and remember any changes, it is instead required only to track the performance of the agent. This simpler task leaves less room for error. In addition, the learning system is designed to be off-board the agent. Disconnecting the two systems makes it easier to integrate additional agents to a single learning system and as a result increase the robustness of the system by allowing the same learning system to continue on even if one agent fails. There are two types of PAL: Fitness Biasing and co-evolution of model parameters.

In this research, Fitness Biasing is used to help sync the simulation with the agent's environment (Figure 3). The learning system periodically contacts the agent for testing. At an appropriate time, every chromosome in the learning system's population is tested on the agent. The fitnesses of the chromosomes tested on the actual agent are recorded and compared to the fitnesses the chromosomes received in simulation. For each fitness, a bias is calculated by dividing the fitness received while testing on the agent by the fitness

received in simulation. These biases are then sent back to the learning system. From that point on through the next punctuated generation, when calculating a new score for a particular control program, it is first multiplied by its assigned bias to skew it towards what would be more expected on the actual agent. The biases are also altered during the learning process. While generating a new population every generation, each time two chromosomes are selected and recombined their biases are averaged together, and this averaged bias is assigned as the bias for their offspring.

## IV. FITNESS BIASING APPLIED TO XPILOT-AI

When creating an Xpilot server, it is possible to change the number of frames per second (FPS). An increased number of FPS causes the game to run considerably faster. As a result, more information must be sent between the server and the clients connected to it. Ideally, when attempting to run a learning algorithm it is run as fast as possible. However, due to limitations in the design of Xpilot-AI, running it at a too high FPS causes the game to behave abnormally, reducing the learning capabilities. This is a problem for real-time learning. Typical game play is at 16 FPS. If a GA was run at this speed against the opponent, it would not complete enough games to have a noticeable learning effect. A background GA can be running, but unless it is at increased speeds, the real-time learning will again be nominal. What is needed is to run a background GA at high speeds, using Fitness Biasing to periodically help in the simulated learning process. This is what was done in this research. In order to determine the best FPS for the learning system, a standard genetic algorithm was used to do learning at different speeds.

This agent was controlled by an expert system and the genetic algorithm learned optimal values for the parameters shown in Figure 4. To calculate the fitness, each agent was allowed to engage for two minutes in simulation. Its opponent was a robust hand-coded expert agent named Sel. During these
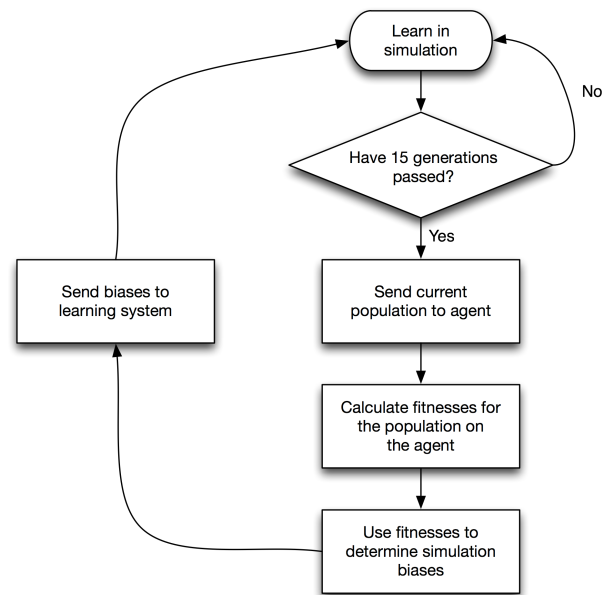


Figure 3. Fitness Biasing

• span – the angle between the line from the agent's nose to a target location and the edge of the nearest wall. Used to determine if the agent is blocked from a bullet by a wall.
• offset_inc – indicates the increments used to determine the optimal direction to turn to avoid crashing into a wall
• same_spread – the difference allowed between the distances returned by two wall feelers which would result in considering them equal.
• wall_span1 – the angle off the ship's track used to feel for the closest wall.
• wall_span2 – the angle off the ship's track used to feel for the second closest wall.
• vd_bullet_dist – determines the bullet alert value required to consider the bullet very dangerous.
• d_bullet_dist – determines the bullet alert value required to consider the bullet dangerous.
• 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 – the speed of the ship in relation to the distance to the closest wall. Used to determine if the ship should take action to avoid the wall.
• medium_wall_speed – similar to close_wall_speed, but for walls that are farther from the agent.
• 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 – similar to c_angle_before_thrust, but used in a rule with lower priority
• 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_no_thrust_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.

Figure 4. A list of the parameters from the control program that the genetic algorithm learned for the Xpilot combat agent

two minutes, the agent would receive one point of fitness for every frame it was alive, plus 1000 points of fitness for every time it killed its opponent, and lose 20 seconds off its total time available for each death it experienced.

This genetic algorithm agent ran on servers set to varying frames per second, starting at 16 FPS and up through 128 FPS. Five trials were run at each speed with the fitnesses at each 5 generations averaged. The results of these trials [Figure 5] were used to confirm that the performance of agent learning drastically decreases as the FPS of a server increases. An agent running at 16 FPS performed the best, followed by 32 FPS and
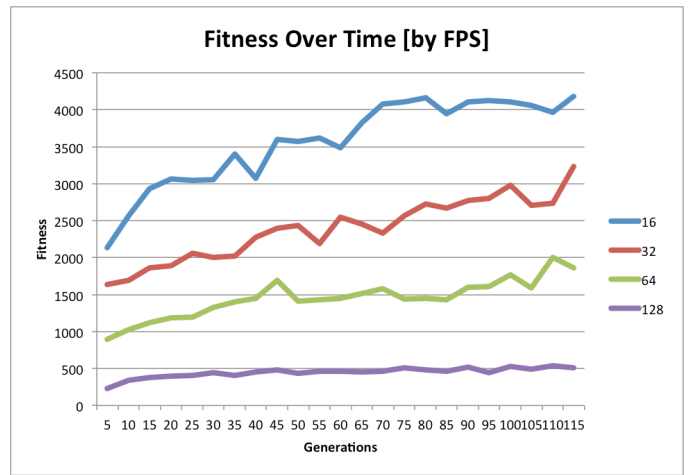


Figure 5.   Data from the genetic algorithm-controlled agent running at different FPS

so on. Agents running at 128 FPS showed slow but steady improvement over the course of 115 generations.

As the increase in FPS from server to server effectively increases the amount of game play the agent gets during a certain period of time, we scaled the data to get the above graph to normalize every agent to the 16 FPS agent. Since each was run for two minutes despite the FPS, we divided the fitnesses of the agent run at 32 FPS by two, the fitnesses of the 64 FPS agent by 4, and so on. This division occurred because the fitness evaluations for the varying FPS all lasted for two minutes. An agent running at 32 FPS, for example, was effectively allowed be tested twice as long given that twice the number of frames passed in the same amount of time.

To confirm these results, we took the best agent for each of these genetic algorithms at the 50th and 100th generations and tested them in games at 16 FPS to attain their fitnesses over five trials [Figure 6]. Their fitnesses were consistent with the results of the genetic algorithm trials shown in Figure 5.

In another test to ensure that the results were consistent, we ran an agent at 128 FPS with one-eighth the time for each fitness calculation [Figure 7]. The fitness matches that of the
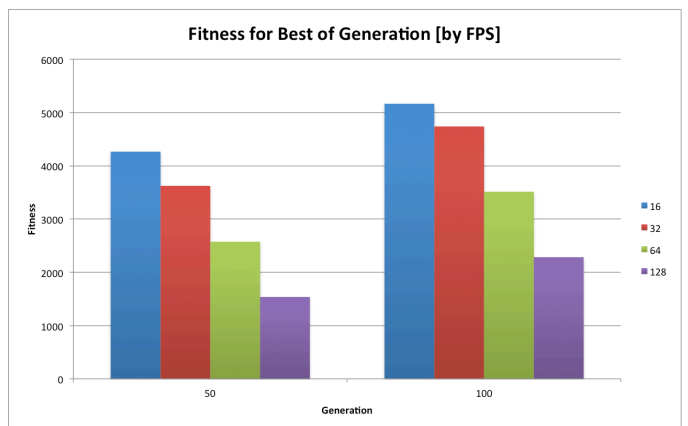


Figure 6.   Fitnesses of the best agents from the 50th and 100th generations of the GAs
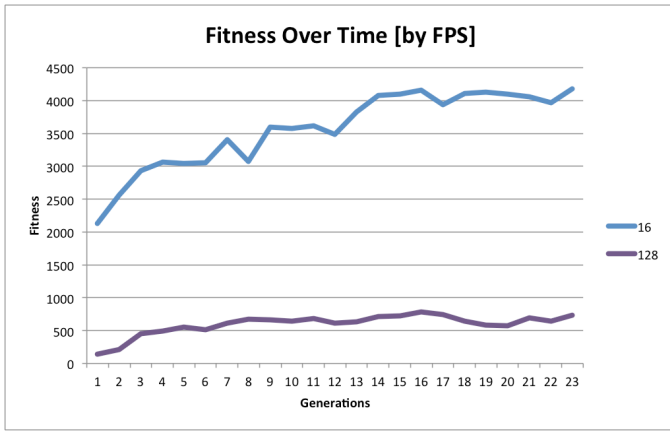
Figure 7.   Fitnesses over time of 16 FPS agent and a 128 FPS agent with 1/8 the time for fitness calculations



Figure 8.   Fitness biased agent compared to a standard GA run at both 16 FPS and at 128 FPS

fitness plot we received by dividing as described for Figure 5.

Since 128 FPS was determined to be the lowest FPS that caused issues for learning while still showing consistent aspects of continued learning, it was selected for testing the application of Fitness Biasing to Xpilot-AI. In the past, tests have been run primarily at 64 FPS as that was the fastest Xpilot could be run at without losing too much in the way of performance. Here, an attempt is made to learn even while running at faster speeds.

To run PAL, the previously created GA-agent was modified into two forms: one to act as the simulation and one to act as the agent. The simulation would run just as the normal GA did with two key additions: while calculating fitnesses, it would bias them as previously described. In addition, every 15 generations, it would connect to the agent and calculate new biases. While doing this, the GA would halt itself and discontinue learning until the new biases were received.

The actual agent would run separately but simultaneously, though it would not learn although it had the ability to calculate fitness. It had a communication system that ran indefinitely, waiting for the learning system to signal it with a new population to test. Upon receiving a new population, it would run each chromosome of the population on the agent, record their fitnesses, calculate their biases, and then send the new information back to the simulation. At this point it would continue to play using the best of the chromosomes for its controller.

Upon receiving results from the agent, the simulation would overwrite its own collection of biases with the new information and continue learning until another 15 generations had passed. This process is repeated until stopped by the researcher, or some predetermined stopping point has been reached (i.e. stop after a certain number of generations).

## V.   RESULTS

Five tests runs using five randomly generated populations were run for 115 generations each. Based on this data, it is clear that the fitness biasing produced favorable results. The system tested the current population on the agent to generate new biases every 15 generations. Every 5 generations, the average fitness of the population was recorded. Figure 8 shows
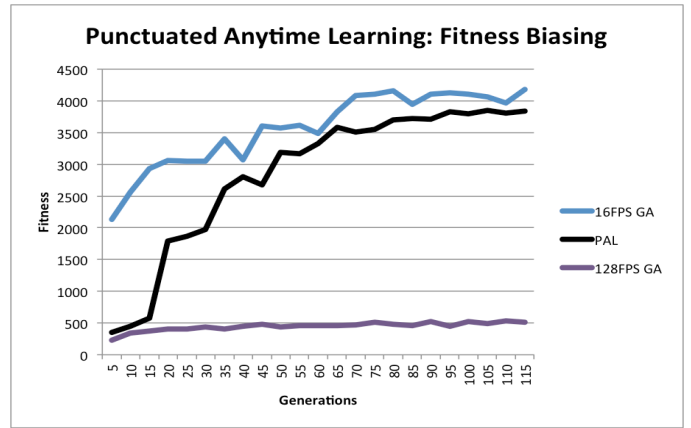
the average fitness over time as the population is evolving. Based on this data, it can be observed that there was great improvement in the fitness of the agent throughout the learning period. In the beginning, the fitness averaged at approximately 350 and rose to an average of approximately 3840 by the end of testing.

Of additional importance is a comparison of controller quality related to the amount of time spent playing on the agent. In the genetic algorithm controlled agents, the entire game is played live against competitors in order to learn in real time. After five generations, its fitness averaged at about 2100. In the fitness-biased agent, though it had learned over the course of 35 generations in our testing, only two of these generations were spent on the agent playing against Sel. After these two generations, the fitness biasing system's average fitness is already higher than that of the fifth generation genetic algorithm-controlled agent and is approximately equal to the tenth generation. This is an important factor for real-time learning because it is learning faster per games played against its opponent.

The above empirical data can also be confirmed by observation. At the start of the learning process, the agent regularly made clear mistakes in combat. For example, some control programs would not turn sharply enough to avoid a bullet or a wall, not thrust to avoid a bullet, or aim incorrectly at its opponent when firing. However, towards the end, these mistakes were corrected in the majority of the learned control programs. The agent would adeptly dodge enemy bullets while responding quicker and firing back with greater accuracy. This increase in optimal behavior can be attributed to the learning that took place as a result of the Fitness Biasing system. By the end of training, all five trials produced individuals with intelligent behavior that were able to consistently beat Sel both in fitness and by the scoring system Xpilot uses to measure player kills and deaths.

## VI.   CONCLUSIONS AND FUTURE WORK

Fitness Biasing when applied to Xpilot-AI agents can be used to evolve exceptional controllers that are highly competitive when facing opponents. The agents show the ability to learn effective combat behavior that appears complex

to an observer. At this point, this research has only been tested on the agent's ability to defeat computer-controlled agents. In future work, we will continue to build on the complexity of the system and test it against human opponents to measure its effectiveness against adaptive players as well as its capability to learn and change based on its enemy's varying behaviors. In addition, we will test the system with co-evolution of model parameters, an alternative method of Punctuated Anytime Learning.

## REFERENCES

[1] G. Yannakakis and J. Hallam, "Evolving opponents for interesting interactive computer games," Proceedings of the 8th International Conference on the Simulation of Adaptive Behavior (SAB 2004), pp. 499 – 508, 2004.

[2] G. Parker and M. Parker, "Evolving parameters for Xpilot combat agents," Proceedings of the 2007 IEEE Symposium on Computational Intelligence in Games (CIG 2007), Honolulu, HI, April 2007.

[3] G. Parker and M. Parker, "The evolution of multi-layer neural networks for the control of Xpilot agents," Proceedings of the 2007 IEEE Symposium on Computational Intelligence in Games (CIG 2007), Honolulu, HI, April 2007.

[4] G. Parker and M. Parker, "Using a queue genetic algorithm to evolve Xpilot control strategies on a distributed system," Proceedings of the 2006 IEEE Congress on Evolutionary Computation (CEC 2006), Vancouver, BC, Canada, July 2006.

[5] M. Allen, K. Dirmaier, and G.Parker, "Real-time AI in Xpilot using reinforcement learning," Proceedings of the 2010 World Automation Congress International Symposium on Intelligent Automation and Control (ISIAC 2010), Kobe, Japan, September 2010.

[6] S. M. Lucas, "Estimating Learning Rates in Evolution and TDL: Results on a Simple Grid-World Problem," Proceedings of the 2010 IEEE Congress on Computational Intelligence in Games (CIG 2010), Copenhagen, Denmark, August 2010.

[7] G. Parker and M. Probst, "Using evolutionary strategies for the real-time learning of controllers for autonomous agents in Xpilot-AI," Proceedings of the 2010 IEEE Congress on Evolutionary Computation (CEC 2010), Barcelona, Spain, July 2010.

[8] R. Baumgarten, S. Colton, and M. Morris, "Combining AI methods for learning bots in a real-time strategy game," International Journal of Computer Games Technology, vol. 2009.

[9] K. Stanley, B. Bryant, I. Karpov, and R. Miikkulainen, "Real-time evolution of neural networks in the NERO video game," AAAI-06, pp. 1671-1674, Boston, MA, 2006.

[10] S. Priesterjahn, O. Kramer, A. Weimer, and A. Goebels, "Evolution of human-competitive agents in modern computer games," Proceedings of the 2006 IEEE Congress on Evolutionary Computation (CEC 2006), Vancouver, BC, Canada, July 2006.

[11] J. Grefenstette and C. Ramsey, "An approach to anytime learning," Proceedings of the Ninth International Conference on Machine Learning, pp. 189-195, 1992.

[12] G. Parker, "Punctuated Anytime Learning for Hexapod Gait Generation," Proceedings of the 2002 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2002), EPFL, Switzerland, October 2002.