Learning to be a Bot: Reinforcement Learning in Shooter Games

Michelle McPartland and Marcus Gallagher

School of Information Technology and Electrical Engineering
University of Queensland
St Lucia, Australia
{michelle,marcusg}@itee.uq.edu.au

Abstract

This paper demonstrates the applicability of reinforcement learning for first person shooter bot artificial intelligence. Reinforcement learning is a machine learning technique where an agent learns a problem through interaction with the environment. The Sarsa(λ) algorithm will be applied to a first person shooter bot controller to learn the tasks of (1) navigation and item collection, and (2) combat. The results will show the validity and diversity of reinforcement learning in a first person shooter environment.

Introduction

Over the past decade substantial research has been performed on reinforcement learning (RL) for the robotics and multi-agent systems (MAS) fields. In addition, many researchers have successfully used RL to teach a computer how to play classic strategy games such as backgammon (Tesauro 1995) and go (Silver, Sutton, and Muller 2007). However, there has been little research in the application of RL to modern computer games. First person shooter (FPS) games have common features to the fields of robotics and MAS, such as agents equipped to sense and act in their environment, and complex continuous movement spaces. Therefore, investigating the affects of RL in an FPS environment is an applicable and interesting area to research.

FPS bot artificial intelligence (AI) generally consists of pathfinding, picking up and using objects in the environment, and different styles of combat such as sniper, commando and aggressive. Bot AI in commercial games generally uses rule-based systems, state machines and scripting (Sanchez-Crespo Dalmau 2003). These techniques are typically associated with problems including predictable behaviors (Jones 2003), time consuming fine-tuning of parameters (Overholtzer 2004), and writing separate code for different creature types and personalities. RL is an interesting and promising algorithm to overcome or minimize such problems. For example, experiments with different parameters can be automated, and the same algorithm can be used to generate different personality types.

The aim of this paper is to investigate how well RL can be used to learn basic FPS bot behaviors. Due to the complexities of bot AI, we have split the learning problem into two tasks. The first task looks at navigation and item collection in a maze-type environment. The second task looks at FPS combat. The Sarsa(λ) algorithm will be used as the underlying RL algorithm to learn the bot controllers. Results will show the potential to create different personalities in FPS bots using the same underlying algorithm.

This paper is organized as follows. First, a brief overview of RL will be explained, followed by an outline of RL applied to computer games. The method section will outline the common algorithm used in both experiments. The next two sections describe the experimental setup, results and discussion of the navigation and combat experiments.

Background

RL is a popular machine learning technique which allows an agent to learn through experience. An RL agent performs an action a in the environment which is currently in state s, at time t. The environment returns a reward r indicating how well the agent performed based on a reward function. The agent's internal policy is then updated according to an update function. Several RL algorithms have been developed over the years including TD, Q-learning and Sarsa. The Sarsa algorithm, similar to Q-learning, has successfully been applied to MAS using computer game environments (Bradley and Hayes 2005; Nason and Laird 2005).

An important part of all RL algorithms is the policy. The policy is a mapping between states and actions, called state-action pairs, and provide the path the agent should take to reach the maximum reward for the task. The two most common types of policy representations are the tabular and generalization approach. The tabular approach uses a lookup table to store values indicating how well an action performs in a state, while the generalization approach uses a function approximator to generalize the state to action mapping.

A recognized problem with the tabular approach is the issue of scalability in complex continuous domains (Bradley and Hayes 2005; Lee, Oh, and Choi 1998). As the state and action space of the problem increases, the size of the policy lookup table exponentially increases. The literature shows numerous ways to address this problem, such as approximating value functions (Lee, Oh, and Choi 1998) and abstracting sensor inputs (Bradley and Hayes 2005; Manslow 2004). This paper uses the tabular approach with data abstraction for the sensor inputs, due to successes in the literature in similar complex continuous problem spaces (Manslow 2004; Merrick and Maher 2006).

Eligibility traces are a method to speed up the learning process by increasing the memory of the agent (Sutton and Barto 1998). A trace history of each state-action pair is recorded and is represented as $e(s,a)$. A received reward is propagated back to the most recently recorded state-action pairs. The eligibility trace factor (λ) and decay factor (γ) is used to update the traces as seen in equation 1.

$$e(s,a) \leftarrow \gamma\lambda e(s,a) \quad (1)$$

The Sarsa(λ) algorithm updates each state-action pair $Q(s,a)$ in the policy according to equation 2.

$$Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a) \quad (2)$$

Where α is the learning rate and δ is defined in equation 3.

$$\delta \leftarrow r + \gamma Q(s',a') - Q(s,a) \quad (3)$$

Where r is the received reward, γ is used to ensure convergence of the policy by discounting future rewards, and $Q(s',a')$ is the value of the next state-action pair.

The application of RL toward modern computer games remains poorly explored in current literature, despite preliminary findings displaying promising results. Manslow (2004) applied an RL algorithm to a racing car game and dealt with the complexity of the environment by assessing the state at discrete points in time. Actions are then considered as a continuous function of the state-action pair. In a fighting simulation game Graepel, Herbrich, and Gold (2004) applied the Sarsa algorithm to teach the non-player characters (NPCs) to play against the hard-coded AI. Results found a near optimal strategy and interesting behaviors of the game agents were observed.

An interesting development in RL and games is seen in Merrick and Maher's (2006) research. A motivated RL algorithm is used to control NPCs in a role-playing game. The algorithm is based on Q-learning and uses an ε-greedy exploration function. They use a cognitive model of curiosity and interest similar to Blumberg et al.'s (2002) work where states and actions are dynamically added to the corresponding space when certain conditions are met. Results showed that the agent was able to adapt to a dynamic environment. The method used in these approaches is not necessarily suited to FPS bot controllers, as they do not need to adapt to new types of objects in the environment. In FPS games, object types and how to interact with them are usually defined before the game starts.

While RL has been extensively used in the MAS (Tan and Xiao 2005) and robotics domains (Lee, Oh, and Choi 1998), there is very little applied research in FPS games. Previous work provides an overview of applying RL to FPSs and preliminary results (McPartland, 2008). Vasta, Lee-Urban, and Munoz-Avilla (2007) have applied RL to learn winning policies in a team FPS game. The problem model was directing a team player to move to certain strategic locations in a domination team game. Each player's actions were hard-coded, only the domination areas on the map, where the team players could go, were learnt. A set of three locations were used in the experiments which reduced the state space considerably. The complexity of the state space was reduced to 27 combinations enabling the algorithm to develop a winning policy that produced team coordination similar to human teams.

Method

A purpose-built 3D FPS game environment was used for both experiments described in this paper. The game world was an indoor building type environment, equipped with walls, items, and spawn points. Bots in the game were able to move around the environment, sense their surroundings, pick up items, and shoot at enemies.

The RL algorithm used for the experiments was the tabular Sarsa algorithm with eligibility traces (Sarsa(λ)) (Sutton and Barto 1998). The tabular Sarsa(λ) algorithm was chosen as it learns the action-selection mechanism within the problem (i.e., mapping states to actions in the policy table). On the other hand state value RL algorithms (e.g., TD-lambda) are able to learn the state transition function, but need an extrinsic action-selection mechanism to be used for control. Therefore state to action mapping algorithms, such as tabular Sarsa(λ), are more suitable than state value algorithms for FPS bot AI.

When a state-action pair occurred, the eligibility trace was set to 1.0, instead of incrementing the current trace by 1.0, as the former case encourages faster learning times (Sutton and Barto 1998).

A small learning rate was used in all experiments, and was linearly decreased during the training phase according to equation 4.

$$d = \alpha i - \alpha e / n \quad (4)$$

Where d is the discount rate applied at each iteration, αi is the initial learning rate (0.2), αe is the target end learning rate (0.05), and n is the total number of iterations for the training phase (5000).

An ε-greedy exploration strategy was used with ε set to 0.2, in other words random actions where chosen two out of ten times, otherwise the best action in the current policy was chosen. If the policy consisted of equal highest valued actions, then one was selected at random. This strategy was chosen due to its success in other RL problems (Manslow 2004; Merrick and Maher 2006; Tan and Xiao 2005).

Navigation Task

The aim of the navigation task was to investigate how well a bot could learn to traverse a maze-type environment while picking up items of interest.

Experimental Setup

The test environment was at a scale of 50m x 50m. Figure 1 shows the layout of the navigation map. There were 54 item spawn points, each with a respawn time of 10 update cycles. All bots in the experiments moved at a speed of 0.2 meters/update cycle. The human-sized RL bot was equipped with six sensors, which were split into two groups of three. One sensor was directly in front of the bot, one 20 degrees to the left, and one 20 degrees to the right. The first three sensors were used to determine if there were any obstacles in view of the bot. The value returned by the obstacle sensors is either 0, 1 or 2, where 0 is no obstacle, 1 means there is an obstacle close to the bot (within four meters), and 2 means there is an obstacle far away from the bot (within ten meters). The second set of sensors was used to indicate where items were in relation to the bot. The item sensors were spaced in the same formation as the obstacle sensors, and also use the same abstraction to indicate if items are close or far away. The RL bot was equipped with the following actions: move forward; turn left; and, turn right. The policy table for the navigation task totaled 2187 entries.

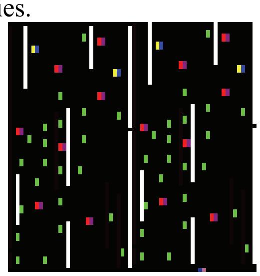

Figure 1. Navigation map. Green represents health items, red represents ammo, and yellow represents bot spawn points.

The reward function for the navigation task consisted of the three objectives: (1) minimize collisions; (2) maximize distance travelled; and, (3) maximize number of items collected. These objectives were guided through rewards and penalties given to the bot during the training phase. A small penalty (-0.000002) was given when the bot collided with environment geometry. A small reward (0.000002) was given when the bot moved, and a large reward (1.0) was given when the bot collected an item. Small values were chosen for the first two objectives as the occurrence of them in the training phase was very high. If the reward values were higher, then the item collection reward would be negligible when it occurred.

Table 1 lists the trial number, discount factor, and eligibility trace parameters for the navigation experiment. A range of values were chosen to determine the overall effect they had on the task.

The navigation bot was trained over 5000 iterations. Following the training phase, the learnt policy was replayed over 5000 iterations with learning disabled. Replays were performed as they provide a more realistic picture of how the learnt policy performs. The following section displays and discusses the results from the replay of the learnt policy.

Table 1. Trial Parameters

Trial number	Parameters
1	$\gamma = 0.0$ $\lambda = 0.0$
2	$\gamma = 0.0$ $\lambda = 0.4$
3	$\gamma = 0.0$ $\lambda = 0.8$
4	$\gamma = 0.4$ $\lambda = 0.0$
5	$\gamma = 0.4$ $\lambda = 0.4$
6	$\gamma = 0.4$ $\lambda = 0.8$
7	$\gamma = 0.8$ $\lambda = 0.0$
8	$\gamma = 0.8$ $\lambda = 0.4$
9	$\gamma = 0.8$ $\lambda = 0.8$
10	Random

Results and Discussion

Figure 2 shows the number of collisions that occurred with the environment geometry or walls. Trials 2, 5, 8 and 9 did not collide with any objects, while the random trial (10) collided many times (800). Trials 2, 5 and 8 have their eligibility trace value in common ($\lambda = 0.4$), which suggests that the eligibility trace is very susceptible to finding good policies in this problem. Trials 1, 4 and 7 had no eligibility trace ($\lambda = 0$), and they collided the most with objects. The collisions in trial 1 (387) were almost double that of trial 4 (197), while trial 7 collided significantly more again (642). The results show that when planning is used (high eligibility traces), no collisions occurred, but when one-step backup is used ($\lambda = 0$) more collisions occurred.

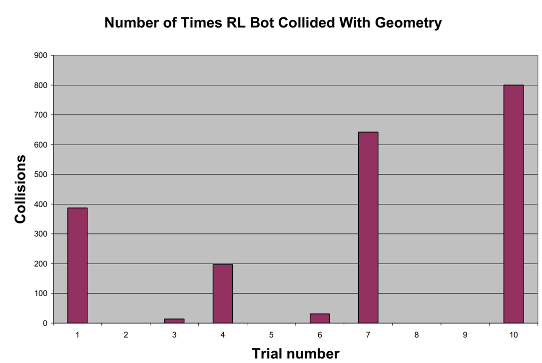

Figure 2. Graph of collisions with obstacles.

Figure 3 shows how far in meters the bot travelled. The data shows that trials 2, 3, 5, 6, 8, and 9 (the trials that had little to no collisions) did not travel a significant distance. In fact, on observation of the replay, some of the bots became stuck upon colliding with the first wall they encountered, and entered a flip-flop state (e.g., repeatedly turning left then right). In contrast, trials 2 and 9 did not move a single step as there was no move forward action for the initial starting state the bot was in. On the other hand, trials 1, 4 and 7 all performed well in this objective, with all trials at least doubling the distance achieved in the random trial.

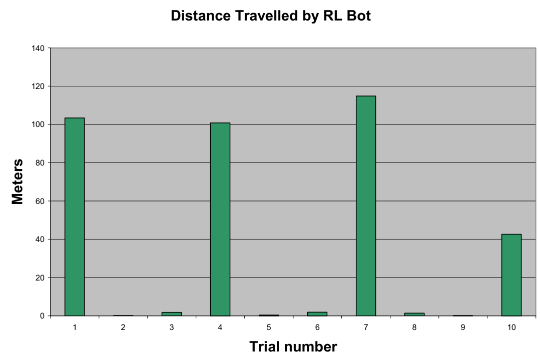

Figure 3. Graph of distance travelled by the bot.

Figure 4 shows the number of items collected in the replay. Here we see the complete picture of trials 3 and 6. Each bot did not travel far, however they were able to pick up a reasonable number of items. Observation showed that these bots were 'camping' (i.e., not moving from) an item's spawn point, and were successful in two of the three objectives in the reward scheme. Trials 2, 5 and 8 were successful in the collision objective, as they minimized collisions to zero (i.e., the optimal solution). However, these trials performed badly in the other two objectives. All three trials had a common eligibility trace of 0.4, which implies that trials with this parameter experienced many collisions in the training phase, therefore learning to minimize collisions at the cost of shorter distance travelled. Similarly, trial 9 performed well in the collision objective, but badly in the other two. The failure of trial 9 may be attributed to the high trace factor and eligibility trace. For the navigation task, the data indicates that no eligibility trace leads to the most successful outcomes, with trace factor varying the success only slightly. Trials 1, 4 and 7 performed well in all three objectives. The policy learnt that allowing some collisions resulted in a bot that moved further through the environment and that was able to pick up more items. The success of trials with the eligibility trace set to zero indicates that the navigation task did not require much planning (only one-step backup was needed).

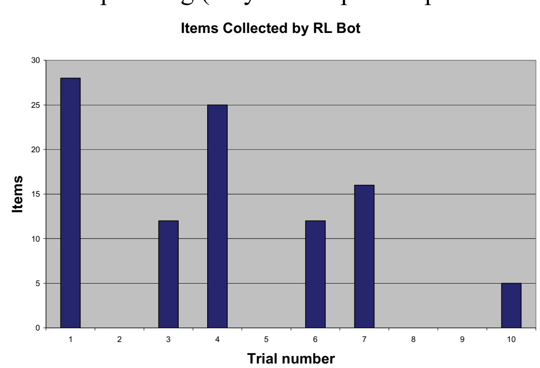

Figure 4. Graph of items collected by the bot.

Figure 5 shows the navigation paths of the trained bots in four of the trials. The figures clearly show how well the bot in trials 1, 4 and 7 performed in the navigation task. The random path showed that the bot never became immobilized, but stayed in the same small area for all 5000 iterations.

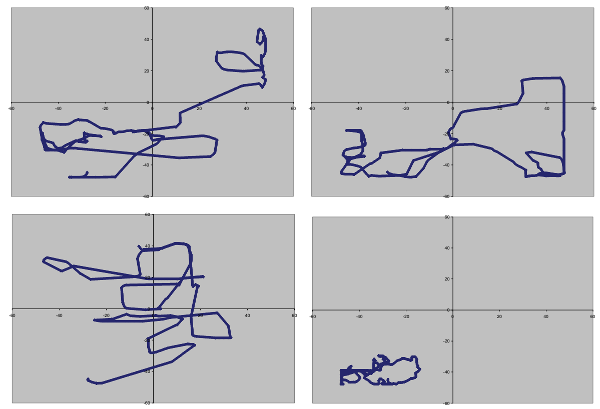

Figure 5. Recorded paths of trial 1 (top left), trial 4 (top right), trial 7 (bottom left) and trial 10 (bottom right)

Overall, trial 4 ($\gamma = 0.4$ $\lambda = 0.0$) learnt the best policy for all three objectives. Collisions were low (197), distance travelled was high (101m), and a good number of items were collected (25). The medium discount factor indicates that the previous state-action pair needs to be rewarded at 0.4 times the reward from the successful or not successful next state-action pair. Whereas the high discount factor in trial 7 ($\gamma = 0.8$, $\lambda = 0.0$), had high collisions (642), high distance (115m) and medium item collections (16). The high discount factor was good at learning the travel objective, but was not as effective in the collision and item collection objectives. The low discount factor in trial 1 ($\gamma = 0.1$, $\lambda = 0.0$) learnt a policy with double the collisions than trial 4 (387), but had a slightly higher travel distance (103m) and item collections (28). Therefore, the discount factor mostly impacted on the collision and items collected objectives, as the results show the distance travelled was similar in trials 1, 4 and 7. On observation of trial 4 it was noted that the bot was able to traverse the maze-type environment, and was able to enter enclosed rooms, pick up items, and then exit the rooms.

The Sarsa(λ) algorithm was successfully used to learn a controller for the task of navigation and item collection. The results show that the eligibility trace needed to be kept small as the best solutions did not require much planning. In other words, the task only needed one-step backup to find a good solution. The discount factor had less effect on the objectives than the eligibility trace, but was useful in fine-tuning good policies.

Combat Task

The aim of the combat task experiment was to investigate how well a bot could learn to fight when trained against a state machine controlled opponent. This task will also investigate whether different styles of combat can be learnt from the same algorithm. The enemy AI, in this task, is controlled by a state machine. The state machine bot has a shooting accuracy of approximately 60%, and is programmed to shoot anytime an enemy is in sight and its weapon is ready to fire. The state machine bot follows enemies in sight until the conflict is resolved.

Experimental Setup

The environment used in the combat task was an enclosed arena style environment. This map style was chosen to remove navigation from the problem, and therefore allowing the algorithm to concentrate on combat alone. The state space is defined as follows.

$$S = \{(s_1, s_2, s_3)\},\ s_i \in \{0,1,2\}$$

Where s_1, s_2, s_3 correspond to the bot's three sensors, left, front and right respectively. The sensors differ to those in the navigation task due to the need to have all the enemies' relative positions in the state space. The combat task sensors determined the relative distance and direction of enemy bots from the RL bot. The RL bot was able to perform the following actions: start move forward (a_1); start move backward (a_2); strafe left (a_3); strafe right (a_4); halt (a_5); turn left (a_6); turn right (a_7); and, shoot (a_8). Therefore the action space is defined as follows.

$$A = \{a_1, a_2, \ldots, a_8\}$$

The policy table for the combat task had 216 entries.

A large reward (1.0) was given to accurately shooting or killing an enemy, as these events did not occur very often in the training phase. A very small penalty (-0. 0000002) was given when the bot shot and missed the target, as shooting and missing occurred many times in the training phase. A large penalty (-1.0) was given when the bot was killed, and a small penalty (-0.000002) was given when the bot was wounded.

In this experiment the decay factor and eligibility trace parameters were kept high (see Table 2), due to the need for planning in combat and initial experiments showing good results for higher values. The number of iterations during the training phase was 5000, and then the learnt policy was replayed over 5000 iterations.

Table 2. Experimental parameters

Trial number	Parameters
1	$\gamma = 0.4$ $\lambda = 0.4$
2	$\gamma = 0.4$ $\lambda = 0.8$
3	$\gamma = 0.8$ $\lambda = 0.4$
4	$\gamma = 0.8$ $\lambda = 0.8$

Results and Discussion

Data was collected from the replay and collated into graphs. Figure 6 shows the accuracy of the bot, or the percentage of shots that successfully hit the enemy. Figure 7 displays the number of times the bot died during the replay. Figure 8 displays the number of times the bot killed the enemy.

The bot trained in trial 1 learnt a commando style of combat. The bot would take one shot at the enemy and then turn tail and run away. Unfortunately for the bot, the enemy AI was easily able to track it down and kill it. This strategy saw a high number of deaths (19) and only one kill.

The second and forth trial bots learnt similar strategies. The strategy was very similar to the enemy AI's, as they all favored close combat and turning to keep the enemy to their front. Trial 4 performed slightly better than trial 2, with a 3% higher accuracy (31%) and one more kill (6). In this experiment the bots had to learn to maximize kills while simultaneously trying to minimize deaths. The strategies that learnt to balance the two rewards saw the bots using the environment to their advantage (i.e., by continually moving through the environment). This strategy increased the amount of time spent in combat, which minimized their death count and increased their kill count.

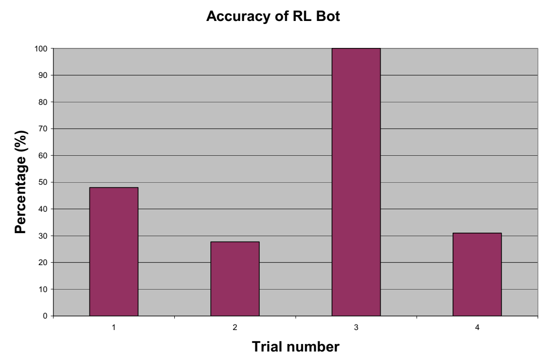

Figure 6. Graph of the bot's accuracy in combat.

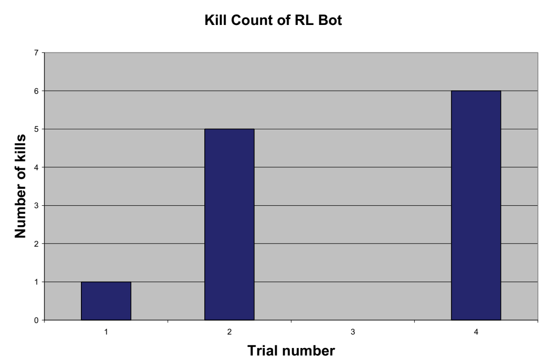

Figure 7. Graph of the number of enemies the bot killed.

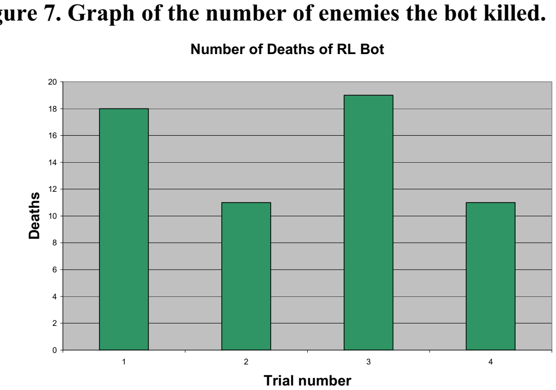

Figure 8. Graph of the number of times the bot died.

The bot in trial 3 did not perform well in the combat objectives. While the policy learnt a reasonable movement style by tracking the enemy, the bot only fired a few shots during the entire replay. The shots were fired with perfect accuracy, but the enemy was not injured enough to die.

The two bots from trial 2 and 4 learnt an aggressive combat strategy similar to the enemy AI. The strategy proved competitive even though the enemy achieved a higher kill rate. One bot learnt a commando (or cowardly) style of fight, where it shot from afar and then ran away. While this strategy was not effective in the arena environment used, it may perform better in other scenarios such as a maze-type environment. The two best scoring bots learnt to imitate the behavior of the enemy AI. One bot learnt suitable movements in combat, however did not learn to shoot enough to kill the enemy.

This section has shown that the Sarsa(λ) algorithm can be used to learn a combat controller for FPS bots. Results indicated that two bots learnt successful behaviors and proved competitive opponents against the state machine controlled bot. It was also observed that different styles of combat could be produced, each of which varied depending on the eligibility trace. In other words, the amount of memory or planning the bot does will ultimately affect the combat strategy it learns.

Conclusion

This paper has shown that RL provides a promising direction for bots in FPS games. A number of advantages for using RL over rule-based systems exist such as minimal code needed for the underlying algorithm and decrease in the time spent tuning parameters. Results have shown that different bot personality types can be produced by changing the parameter associated with planning. Results indicate that the Sarsa(λ) algorithm can successfully be applied to learn the FPS bot behaviors of navigation and combat.

Further work will investigate different environmental setups and multiple runs with changing random seeds. An extension to this work will investigate combining the two controllers, using a hierarchical method, to create a more complete AI for FPS bots. The combined controllers will be investigated in different environment types, such as indoor buildings and sparsely populated open areas.

Acknowledgments

We would like to acknowledge support for this paper from the ARC Centre for Complex Systems.

References

Blumberg, B., Downie, M., Ivanov, Y.A., Berlin, M., Johnson, M.P., and Tomlinson, B. 2002. Integrated Learning for Interactive Synthetic Characters. *ACM Transactions on Graphics* 21(3): 417-426.

Bradley, J., and Hayes, G. 2005. Group Utility Functions: Learning Equilibria Between Groups of Agents in Computer Games By Modifying the Reinforcement Signal. *Congress on Evolutionary Computation*.

Graepel, T., Herbrich, R., and Gold, J. 2004. Learning to Fight. In *Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education*.

Jones, J. 2003. Benefits of Genetic Algorithms in Simulations for Game Designers. Thesis, School of Informatics, University of Buffalo, Buffalo, USA.

Lee, J.H., Oh, S.Y., and Choi, D.H. 1998. TD Based Reinforcement Learning Using Neural Networks in Control Problems with Continuous Action Space. *IEEE World Congress on Computational Intelligence*. Anchorage, USA.

Manslow, J. 2004. Using Reinforcement Learning to Solve AI Control Problems, in *AI Game Programming Wisdom 2*, S. Rabin, (Editor). Hingham, USA: Charles River Media.

McPartland, M. 2008. A Practical Guide to Reinforcement Learning in Shooter Games, in *AI Game Programming Wisdom 4*, S. Rabin, (Editor). Boston, USA: Charles River Media.

Merrick, K., and Maher, M.L. 2006. Motivated Reinforcement Learning for Non-Player Characters in Persistent Computer Game Worlds. In *ACM SIGCHI International Conference on Advances in Computer Entertainment Technology*. Los Angeles, USA.

Nason, S., and Laird, J.E. 2005. Soar-RL: Integrating Reinforcement Learning with Soar. *Cognitive Systems Research* 6(1): 51-59.

Overholtzer, C.A. 2004. Evolving AI Opponents in a First-Person-Shooter Video Game, Thesis, Computer Science Department, Washington and Lee University: Lexington, VA.

Sanchez-Crespo Dalmau, D. 2003. *Core Techniques and Algorithms in Game Programming*. Indianapolis, Indiana: New Riders.

Silver, D., Sutton, R., and Muller, M. 2007. Reinforcement Learning of Local Shape in the Game of Go. In *International Conference on Artificial Intelligence*. Hyderabad, India.

Sutton, R.S., and Barto, A.G. 1998. *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press.

Tan, A.H., and Xiao, D. 2005. Self-Organizing Cognitive Agents and Reinforcement Learning in Multi-Agent Environment. In *International Conference on Intelligent Agent Technology*. Compiegne, France.

Tesauro, G. 1995. Temporal Difference Learning and TD-Gammon. *Communications of the ACM* 38(3): 58-68.

Vasta, M., Lee-Urban, S., and Munoz-Avila, H. 2007. RETALIATE: Learning Winning Policies in First-Person Shooter Games. In *Seventeenth Innovative Applications of Artificial Intelligence Conference* (IAAI-07). AAAI Press.