Reinforcement Learning in First Person Shooter Games

Michelle McPartland and Marcus Gallagher, *Member, IEEE*

***Abstract*—Reinforcement learning (RL) is a popular machine learning technique that has many successes in learning how to play classic style games. Applying RL to first person shooter (FPS) 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 pathfinding 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 pretrained 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.**

***Index Terms*—Artificial intelligence (AI), computer games, reinforcement learning (RL).**

I. Introduction

OVER the past two decades, substantial research has been performed on reinforcement learning (RL) for the robotics and multiagent systems (MASs) fields. In addition, many researchers have successfully used RL to teach a computer how to play classic games such as *Backgammon* [1] and *Hearts* [2]. However, there has been comparatively little research in the application of RL to modern computer games. First person shooter (FPS) games have some features in common with robotics and MAS, such as agents equipped to sense and act in their environment, and complex continuous movement spaces. Therefore, investigating the effects of RL in an FPS environment is a promising area of research.

Modern computer games, and particularly FPSs, are increasingly becoming more complex. Although games have complex environments, the artificial intelligence (AI) 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 pathfinding, 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 [3], and goal-based systems [4]. These techniques are typically associated with a number of problems including predictable behaviors [5], time-consuming fine tuning of parameters [6], 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. RL 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.

One of the main disadvantages of RL is the problem of scaling with increased complexity of the problem space. This complexity issue can in part be resolved by breaking the problem space down into smaller tasks, then finding ways to combine them to produce the overall behavior set. Similar work of breaking down complex problem spaces has been performed in using evolutionary algorithms in an FPS game environment by Bakkes *et al.* [7]. In this paper, we present a preliminary investigation and application of RL to produce 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 is concerned with FPS combat. The Sarsa(λ) algorithm is used as the underlying RL algorithm to learn the bot controllers.

The secondary aim of this paper is to investigate different methods of combining RL controllers to create generalized approaches to build autonomous bot behaviors. In particular, the bot AI produced in this research is appropriate to multiplayer FPS games such as deathmatch scenarios. However, it would be possible to translate the behaviors into a single player enemy bot. Previously trained single-task controllers will be combined to produce bots with a multipurpose behavior set. Three different types of RL will be compared in terms of the differences in performance statistics and behaviors produced. The first method combines previously trained combat and navigation controllers using hierarchical RL. The second method uses simple rules to determine when to use the combat or navigation controller. The third method uses RL to relearn the tasks of

Manuscript received August 06, 2009; revised December 19, 2009; accepted November 29, 2010. Date of publication December 20, 2010; date of current version March 16, 2011. This work was supported in part by the School of Information Technology and Electrical Engineering, University of Queensland and the Australian Research Council Center for Complex Systems.

The authors are with the School of Information Technology and Electrical Engineering, University of Queensland, Brisbane, Qld. 4072, Australia (e-mail: michelle@itee.uq.edu.au; marcusg@itee.uq.edu.au).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCIAIG.2010.2100395

combat and navigation simultaneously. These three methods are compared to a random controller and a state-machine-controlled bot. The groundwork of this paper is presented in the authors' previously published papers [8], [9].

II. Background

RL is a popular machine learning technique that allows an agent to learn through experience. An RL agent performs an action$a$ in the environment, which is in state s, at time t. The RL algorithm outputs an action that is something the agent can do in the environment. The environment then returns a reward r, based on how well that action performed based on a user defined reward function. The set of actions the agent can perform in the environment is termed the action space. A state is an instantiated numerical representation of the environment. For example, the state may contain information about the environment such as the location of cover positions or ammo stores. The set of all possible states is referred to as the state space. A state–action pair is the mapping of how well an action performs in a state and is stored by the policy. The policy evolves over time and provides the path an agent should take to reach the maximum reward for the task. The two most common types of policy representations are the tabular and generalization approaches. The tabular approach uses a lookup table to store numerical values of state–action pairs, while the generalization approach uses a function approximator to model and generalize the state-to-action mapping. However, in general, generalization approaches are not as suited to complex problems spaces such as game AI, as they require more time for both tuning their parameters and training the function approximator. Consequently, this paper will investigate how well the tabular approach can work in the complex environment of FPS games.

A recognized problem with the tabular approach is the issue of scalability in complex continuous domains [13], [14]. As the state and action space of the problem increases, the size of the policy lookup table exponentially increases. The literature suggests numerous ways to address this problem, such as approximating value functions [15] and abstracting sensor inputs [11], [13]. From this previous work, the use of tabular approaches coupled with data abstraction was shown to be a particularly successful strategy in enhancing learning in complex continuous problem spaces [11], [16]. Therefore, we argue that RL can successfully be used in complex environments such as FPS games if practical information is abstracted. Since bot AI can be reduced into separate parts, such as navigation and combat, it is reasonable to investigate this approach. Combat can be even further reduced into ranged and melee attacks if more complex behavior is required. RL can then learn smaller tasks, and planning systems can be used to form the complete AI.

At each time step, the RL algorithm chooses an action depending on the current action selection method, a process known as the dilemma of exploration versus exploitation. This dilemma is the tradeoff between the RL algorithm exploiting the knowledge currently in the policy and exploring the state space for better actions to perform. A number of methods have been proposed to deal with this issue such as greedy selection, ε-greedy selection, and softmax [10]. The literature shows that the ε-greedy method is popular and successful in many different scenarios [11], [12]. The parameter ε is used to control the ratio between exploration and exploitation by a random chance system. Once an action is chosen, the environment calculates a reward indicating how well the agent performed based on a reward function. The agent's internal policy is then updated according to an update (learning) function. Several RL algorithms have been developed over the years including temporal difference (TD), Q-learning, and Sarsa. The Sarsa algorithm, similar to Q-learning, has successfully been applied to MAS using computer game environments [13], [14].

TABLE I
Pseudocode for the Sarsa(λ) Algorithm

```
1:  Initialize $Q(s,a) = 0$, set $e(s,a) = 0$ for all $s$, $a$
2:  Repeat for each training game
3:    Repeat for each update step $t$ in the game
4:      Set $s'$ to the current state
5:      Select an action $a'$
6:      Take action $a'$, observe reward $r$
7:      $\delta \leftarrow r + \gamma Q(s',a') - Q(s,a)$
8:      $e(s,a) \leftarrow 1$
9:      For all $s$, $a$:
10:     $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$
11:       $e(s,a) \leftarrow \gamma\lambda e(s,a)$
12:     $s \leftarrow s', a \leftarrow a'$
13:   Until end game condition is true
```

This paper details the use of a tabular Sarsa algorithm owing to the success of this strategy in similar research and problem spaces [17], [18]. The Sarsa algorithm with eligibility traces [10], termed Sarsa(λ), is used for updating the policy values. Eligibility traces are a method to speed up learning by allowing past actions to benefit from the current reward, and also allowing for sequences of actions to be learned. Sequences of actions are particularly important in combat behavior where the bot AI should be able to perform a combination of actions in order to defeat an enemy. Table I lists the steps of the Sarsa(λ) algorithm. $Q(s,a)$ represents the policy and $e(s,a)$ is the eligibility trace variable for a given state–action pair. Line 7 calculates the TD error using the current reward, decay parameter (γ), and the current and next state–action pair values. Line 8 sets the eligibility trace (λ) to one, to flag that state–action pair as just being visited. The update function is listed in line 10. The previously calculated TD error is used with the learning rate (α) and eligibility trace (λ) to update each state–action pair in the policy. Line 11 reduces the eligibility trace of previously visited state–action pairs. This reduction affects state–action pairs that have just been visited by allowing them to receive more of the current reward than states further in the past. The reduction is determined by the decay (γ) and trace (λ) parameters.

Hierarchical RL has been widely used in agent-based systems [19], [20] and robotics [21] to help handle the complexity in these domains. Hierarchical RL is the process of splitting the problem space up into smaller tasks, and then using a hierarchy to decide what task to perform in different situations.

III. Related Work

The application of RL toward modern computer games remains poorly explored in current literature, despite preliminary findings displaying promising results. Manslow 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 [11]. Actions are then considered as a continuous function of the state–action pair. In a fighting simulation game, Graepel *et al.* applied the Sarsa algorithm to teach the nonplayer characters (NPCs) to play against the hard-coded AI [17]. The results indicated that not only a near-optimal strategy was learned but also interesting behaviors of the game agents were displayed by the game agents.

An interesting development in RL and games is seen in Merrick and Maher's research [16]. 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 work where states and actions are dynamically added to the corresponding space when certain conditions are met [22]. 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 [12] and robotics domains [15], there is very little applied research in FPS games. One particular example is the application of RL to learn winning policies in a team FPS game by Vasta *et al.* [23]. The problem model was directing a team player to move to certain strategic locations in a domination team game. Each of the players' actions was hard coded; only the domination areas on the map, where the team players could go, were learned. A set of three locations were used in the experiments, which reduced the state space considerably. The size 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.

IV. Experimental Setup

A small, purpose-built FPS game was used as the testbed for all experiments described in this paper (see Fig. 1). Investigatory work was performed in a purpose built FPS to keep control over processor cycles to perform a multitude of experiments while in the investigatory phase. A purpose built game was used instead of a commercial game to minimize the overhead of processor cycles (allowing the experiments to run faster), to be able to turn rendering on and off (which can be difficult with commercial game engines), and to maintain control over the update loop. Keeping processor cycles as minimal as possible was very important to run multiple experiments for testing varying parameters and reward functions. As noted in Laird's FPS research [24], commercial game engines may have insidious problems that cannot be changed such as the bots having an advantage over the human player. For example, in *Quake 2* the bots had faster turning circles than the human-controlled bots [24]. In the game designed for our research, all bots have the same capabilities to ensure the comparisons are as correct and fair as possible.

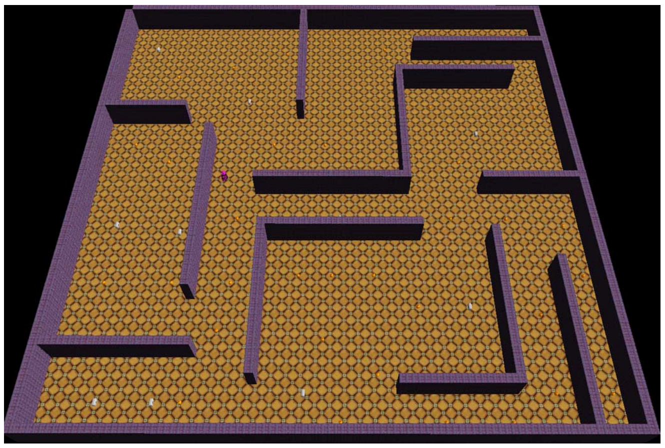
Fig. 1. The game environment.

The game testbed has the basic features of commercial FPS games such as walls, power up items, and bots. Bots are equipped with the ability to turn left, turn right, move forwards, move backwards, strafe left, strafe right, pick up items, and shoot. All bots in the game have exactly the same capabilities and parameters, e.g., turn speed (0.1), speed (0.2 m/update cycle), ammo clip (unlimited), weapon type (laser ray gun with five hit points per shot damage rating), and hit points (50). The laser guns have a cool down timer of 1 s, to avoid a constant stream of fire. Bots respawn ten update cycles after being killed, and they spawn in the same location each time with full health. Once collected items respawn after ten update cycles.

Three different level maps were designed for the experiments to test the AI in varying environments. The first map is the arena [Fig. 2(a)], which consists of four enclosing walls and no items to collect. The arena map is 100 m $\times$ 100 m. The second map is a maze environment [Fig. 2(b)], and was used to investigate how well the bots performed in a more complex environment where there was a lot of geometry to collide with and items to pickup. The third map [Fig. 2(c)] is the combat map and was a simplified version of the maze map. This map was used for combining combat, navigation, and item collection. The maze and combat maps are also 100 m $\times$ 100 m.

As mentioned above, the Sarsa(λ) RL algorithm was used for the experiments. This 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 [10]. A small learning rate was used in all experiments, and was linearly decreased during the training phase according to

$$d = \alpha i - \alpha e/n \tag{1}$$

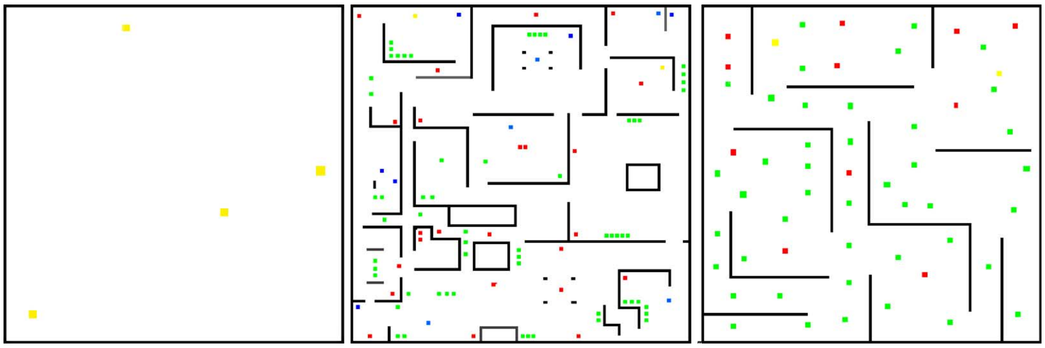

Fig. 2. The layout of (a) the arena map to the left, (b) the maze map in the middle, and (c) the combat map to the right. Yellow represents the bots spawn positions; red, green, and blue represent item spawn points.

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). These values were chosen from success in preliminary runs.

A ε-greedy exploration strategy was used with ε set to 0.2. In other words, random actions were 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 previous success in other RL applications [10], [16], [11].

Each bot was equipped with three sensors, one facing the current direction of the bot, one 20° to the left of the current facing and one 20° to the right. The sensors are able to record information in the environment relevant to the problem, for example, enemy position, geometry, and items. Near corresponds to 0–5 m, and far corresponds to 5–10 m.

V. Results

A. 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.

The reward function for the navigation task consisted of the three objectives: 1) minimize collisions; 2) maximize distance traveled; 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 over the training phase iteration was very high. If the reward values were higher, then the item collection reward would be negligible when it occurred.

The policy table consisted of 2187 state–action pairs. Of the six sensor inputs, three determined whether an obstacle is nonexistent (0), near (1), or far (2). The other three sensors determined whether an item is nonexistent (0), near (1), or far (2). The three output actions were move forward (0), turn left (1), and turn right (2). Each action uses the bot's aforementioned movement speed and turn speed to calculate the quantity of the move.

TABLE II
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

Table II 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 learned policy was replayed 20 times with random seeds ranging from 101 to 120, over 5000 iterations with a small learning rate of 0.1. Replays were performed as they provide a more realistic picture of how the learned policy performs. The following section displays and discusses the averaged results from the replays of the learned policies.

Fig. 3 displays a visual representation of the distribution of rewards during the learning phase of the bot. The positive scores indicate a reward, and the negative scores indicate a penalty. During the first 1200 iterations, the bot received a lot of penalties. After 1300 timesteps, the number of rewards starts increasing quickly. After the second half of the timesteps, the bot received mainly rewards indicating the bot has learned a good policy for this environment.

Trials 1, 4, and 7 learned the exact same policies during the learning phase indicating that the policy converged during the learning phases. Although there are a high number of collisions

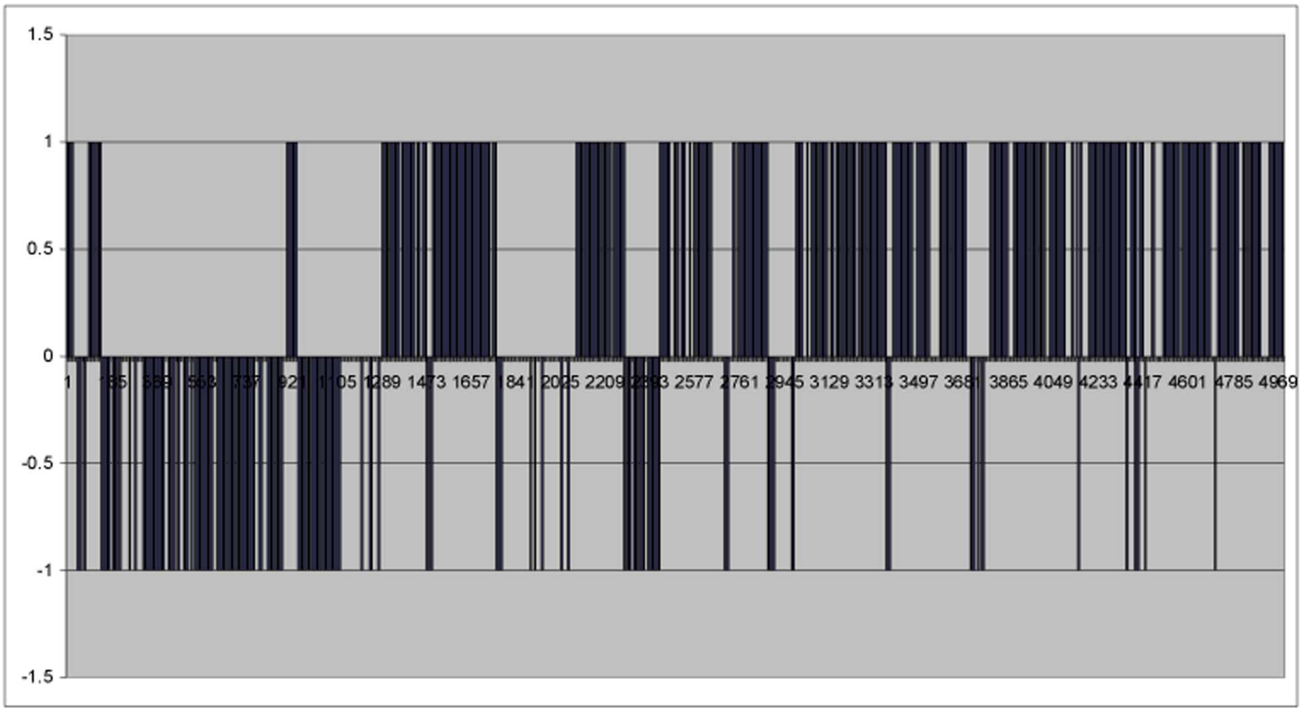

Fig. 3. Frequency of rewards for trial 6 during the 5000 timesteps of the learning phase.

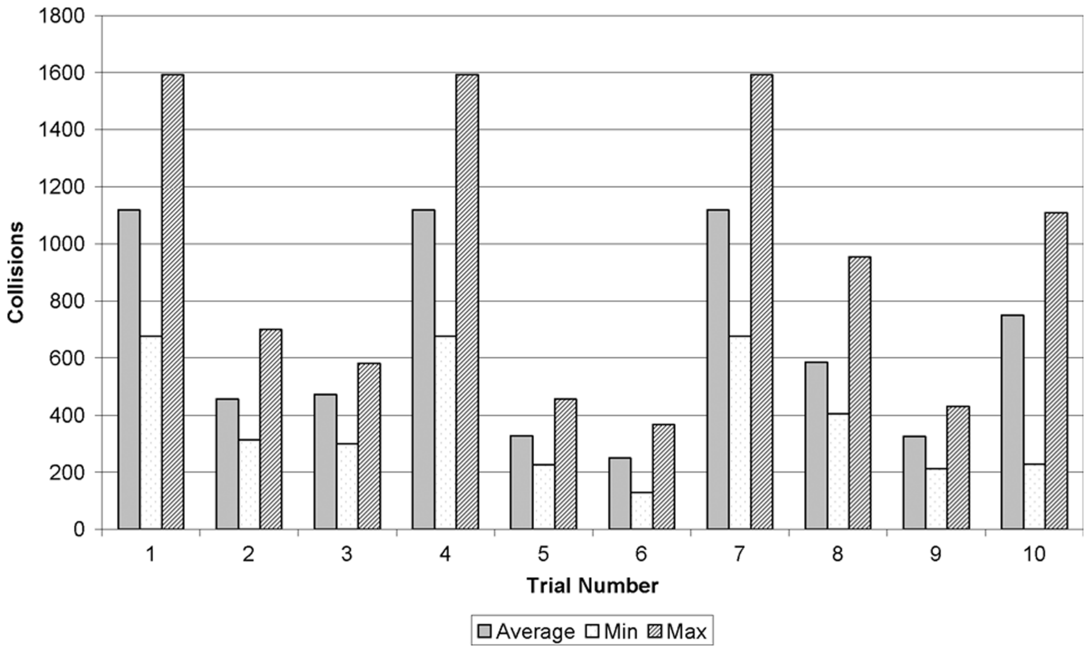

Fig. 4. Graph of number of collisions.

(Fig. 4), many of these were partial collisions where the bot was traveling at an angle against a wall. The high number of collisions is seen due to items being placed near walls. As the item collection reward (1.0) was much higher than the collision penalty (-0.00002), the bot would learn to go for the item and accept the lower penalty. Although this would not be an ideal controller for a commercial game, it does show how tuning the reward values can control the output behavior of the bot. By simply increasing the collision penalty, a bot which is better at avoiding obstacles than collecting items is produced. Trial 6 produced the best policy in minimizing collisions. However, this policy was one of the worst in item collection as seen in Fig. 5.

Trials 1, 4, 5, and 7 performed the best in item collection, also performing the best in a single run with a maximum of 23. All trials except 3 performed better than the random controller showing that the majority of trials have learned good navigation policies. The better trials almost doubled the average items collected of the random trial.

The best performing in the distance objective are the three identical trials 1, 4, and 7. These no planning trials performed the best in items collected and distance traveled, however ex-

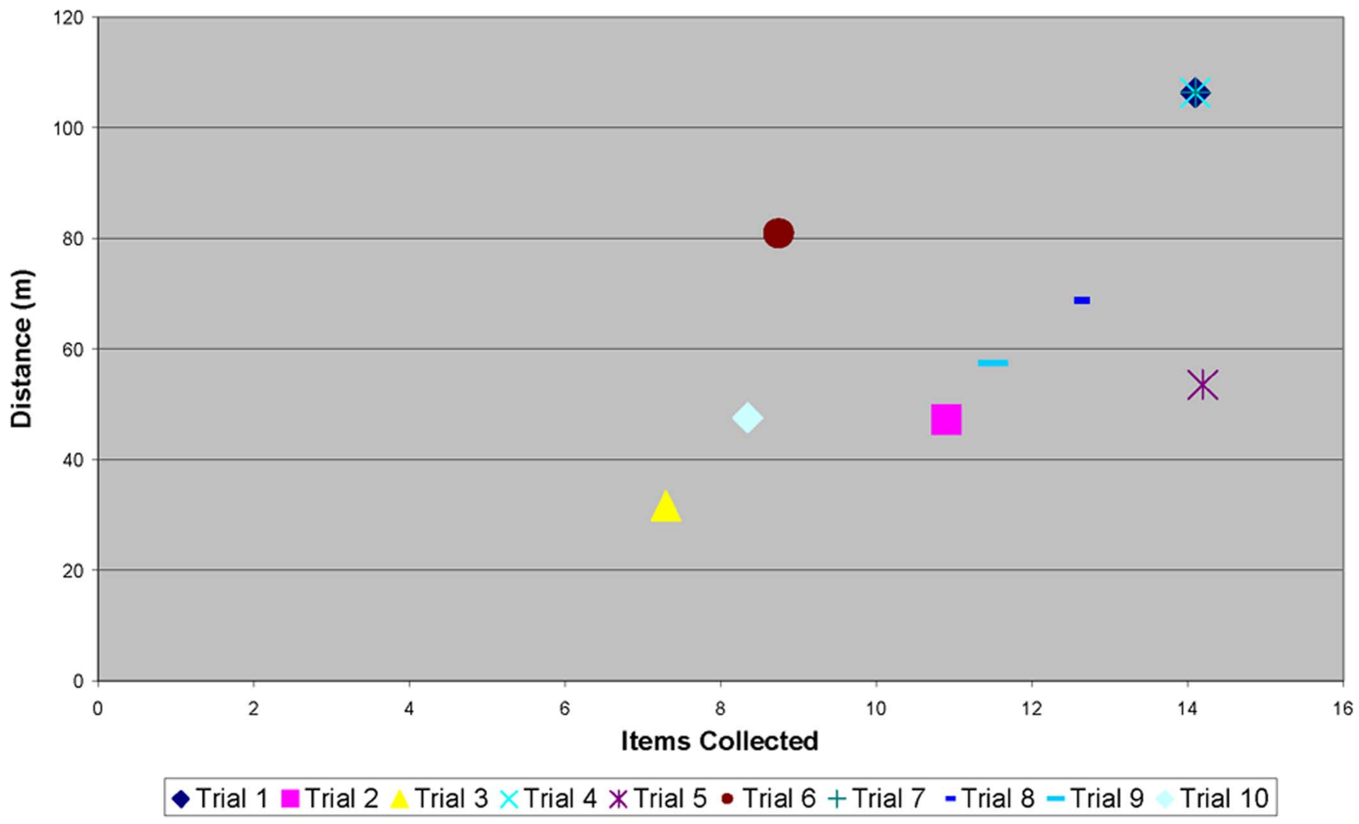

Fig. 5. Pareto front for items collected versus distance traveled.

hibited the most collisions. The random trial traveled the second least distance with trial 3 again performing the worst in the distance objective.

Fig. 5 maps the distance traveled versus items collected using a Pareto front visualization approach. The Pareto front approach was used to map two conflicting objectives to see which solutions dominated in both objectives. The points to the up and rightmost of the plot show the best solutions. Distance and items collected were mapped as these two objectives are interrelated. As many of the items were placed near walls, the RL algorithm had to sacrifice the distance reward to pick up these items. According to the figure, trials 1, 4, and 7 have performed the best in these two objectives combined. These trials have no backup or trace parameters, indicating that no planning is needed to learn the navigation task. This result was expected as navigation is generally based on altering your path based on items in sight to collect, or obstacles to avoid. Trial 5 performed the best in average items collected, however it did not perform as well in distance traveled.

Fig. 6 clearly shows how well the RL bots have learned to navigate the environment. The example from trial 1 (116) shows the bot that traveled the farthest distance in all the tests. Trial 8 (109) performed the best in the item collection objective, and shows a smooth that travels around the bottom half of the map. The majority of bots were able to move around with smooth paths, and were able to pick up items in corners and navigate out of tight corners. Although a high number of collisions occurred with most bots, the paths show clear movement throughout the replay. As mentioned previously, the high collision values are due to the partial collisions, where the bot clips the wall while

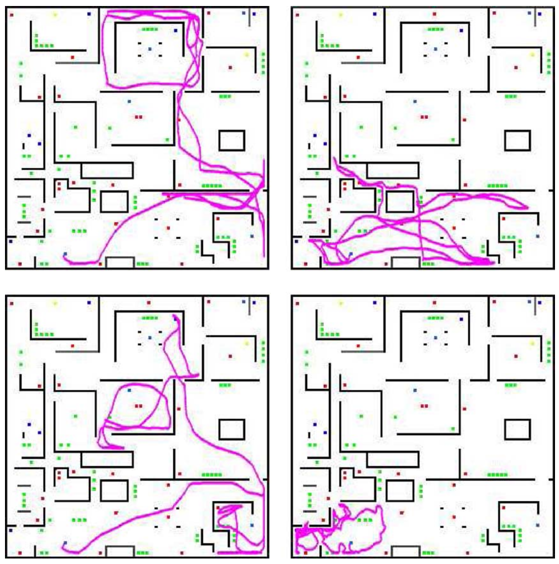

Fig. 6. Recorded paths of trial 1 (116) (top left), trial 8 (109) (top right), trial 1 (106), and trial 10 (bottom).

turning or moving nearby. Improvements to this setup would look at the number of partial and complete collisions. The display path for the random bot path shows only a small area was traversed. The RL bot's paths are smoother and less robotic looking when compared to the random bots.

The Sarsa(λ) algorithm was successfully used to learn a controller for the task of navigation and item collection. Many of the experiments found good policies for navigating the environment. Although the learned controllers for navigation are not usable for a commercial game, mainly due to the high number of collisions, the experiments prove that good policies were able to be learned in this environment. Some of the better runs showed sensible navigation strategies. The results show that the eligibility trace was needed to be kept small as the best solutions required little to no planning. In other words, the navigation task only needed one-step backup to find a good solution. There was an issue with bots replayed getting stuck in corners of the map. This problem is due to the discrete output values of the tabular approach, i.e., turn left corresponded to turn 5° to the left and the turn right output action corresponded to turn 5° to the right. During the replay it was possible that the state space could get caught alternating between two states where the outputs were turn left and then turn right. To reduce this problem a small degree of learning was enabled during the replay (alpha and epsilon). This small amount of learning in the replays allowed the bot to continue learning, and therefore reduced the problem; however, it did not solve it completely. Further work will look at using interpolated tables for linear instead of discrete output actions (e.g., instead of turn left corresponding to a fixed amount, it will instead correspond to a value between a minimum and maximum amount). Despite this problem, the RL algorithm was able to learn a good navigation strategy significantly better than a random strategy, however not as good as using A* with item finding rules.

B. *FPS Combat Task*

The aim of the combat task was to investigate how well a bot could learn to fight when trained against a state-machine-controlled opponent. The enemy AI, in this task, is controlled by a state machine. The state machine bot 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.

The policy table consisted of 189 state–action pairs. Three sensor inputs with three possibilities determine whether an enemy is nonexistent (0), near (1), or far (2) away from the bot. The output actions of the combat task were: move forward (0), turn left (1), turn right (2), move backwards (3), strafe left (4), strafe right (5), shoot (6), and halt (7). Each action uses the bot's aforementioned movement speed and turn speed to calculate the quantity of the move.

The same parameters were used in the combat task as the navigation task (see Table II). The tenth bot was the state machine bot. The state machine bot from the trial 6 experiments was averaged and used to determine how well the RL bots performed. The number of iterations during the training phase was 5000, and then the learned policy was replayed 20 times with random seeds ranging from 101 to 120, over 5000 iterations.

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.

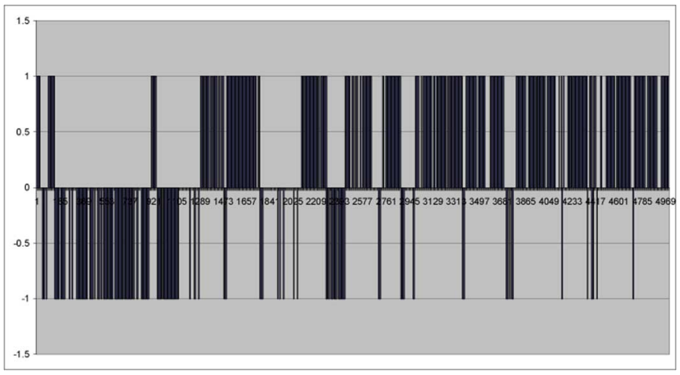

Fig. 7. Frequency of rewards for combat task trial 6 during the 5000 timesteps of the learning phase.

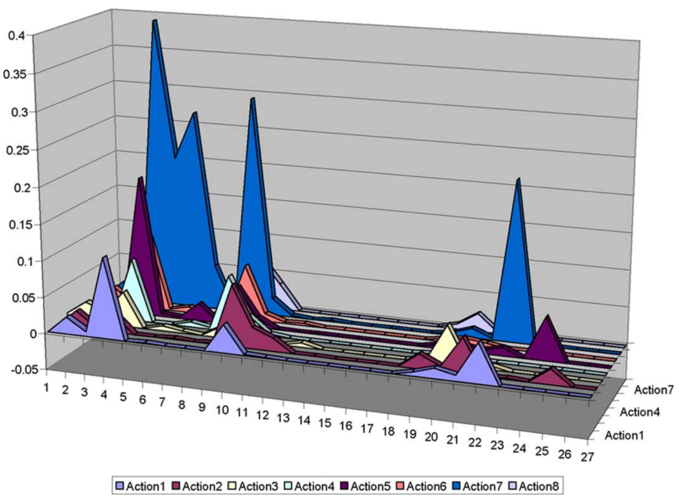

Fig. 8. Policy visualization from trial 6 of the combat task.

Fig. 7 displays a graphical representation of the rewards versus penalty distribution of trial 6 of the combat task. Rewards in the combat task were infrequent, as were accurate shots and kills did not happen often. The frequency of rewards increased in the last 2000 timesteps, indicating that the learning phase could have been extended. Although the combat policy could have been improved with a higher learning phase, the replay shows that a good policy was learned during this number of iterations. Penalties are heavy throughout the learning phase, but are less frequent towards the end 2000 timesteps indicating that the policy has evolved.

Fig. 8 displays a visual representation of the learned policy in trial 6, the best performing bot in the combat task. The policy landscape shows that a good number of the state–action pairs have been activated with action 7 (shoot) dominating the landscape in the first half of the states. Many of the state–action pairs in the middle of the state space had near zero values as they only experienced guide rewards.

Fig. 9 displays the shooting accuracies of the bots in the combat task. Trial 5 performed the best with an average of 86%, and maximum of 95% accuracy. The best single run was trial 8 with 100%. Unfortunately trial 8 only shot 11 times during the replay and only managed one kill. Similar to the navigation task, trials 1, 4, and 7 learned the same policies and therefore have

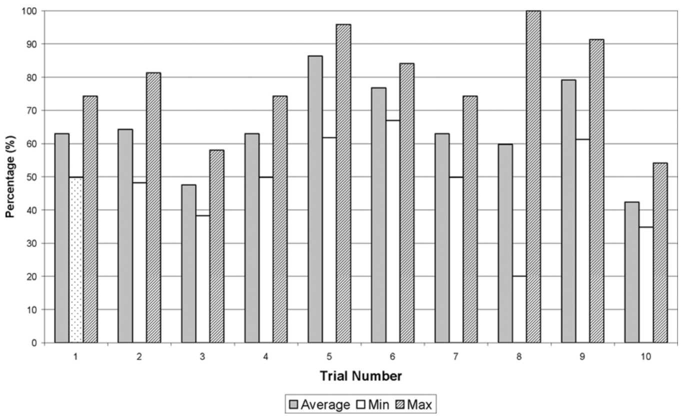

Fig. 9. Graph of shooting accuracies in the combat task.

identical results. The worst scoring trial was trial 3 with an average of 48%. The high average shooting accuracies indicate that all controllers have learned how to shoot.

Fig. 10 displays the mapping of deaths versus kills using a Pareto front visualization. Since the death objective was supposed to be minimized, the death count minimum scale on the graph goes from greatest to smallest. The best scoring trials are those who have the greatest number of kills and least number of deaths. The Pareto front is therefore the top rightmost solutions. Fig. 10 indicates that the best performing experiment in maximizing kills and minimizing deaths is clearly trial 6 with an average of 12.6 kills and 5.7 deaths. The high number of average kills and half the amount of deaths strongly indicates the bot has learned a very competitive combat strategy. Trial 6 used medium discount and high trace parameters ($\gamma = 0.4 \lambda = 0.8$). The next best performing trials were 1, ($\gamma = 0.0 \lambda = 0.0$), $4 (\gamma = 0.4 \lambda = 0.0)$, and 7 ($\gamma = 0.8 \lambda = 0.0$). The results indicate that either no planning or a high amount of planning is needed to learn a good combat strategy.

The worst performing experiments were trial 8 ($\gamma = 0.8 \lambda = 0.4$), and 2 ($\gamma = 0.0 \lambda = 0.4$) with the least amount of average kills (0.15) and a lot of deaths (9.5). The parameters of the worst performing experiments had medium trace values.

Trial 5 learned a combination of aggressive and defensive type of combat strategy, generally hanging back when the enemy was far away, only starting to shoot when it came in close range. In close range fighting the bot would face the enemy straight on and shoot as rapidly as allowed. The bot would dodge the state machine bots attacks by circle strafing around it. The trial 5 bot scored in the midrange of both the kills and death objectives.

The statistics show trial 6 performed the best in kills and deaths, and replay backed this by showing a near-perfect aggressive combat strategy. The trial 6 bot outplayed the state machine bot many times, in both far and close distance combat. The bot followed and shot at any bot in site, chasing down the nonreactive bots, and never giving up on the state machine bot either.

The bot in trial 9 tended to stop in combat and shoot haphazardly. Due to the frequent stopping, the enemy bots would frequently move out of sensor range of the RL bot and escape safely. Only when the bot was in close quarters would the bot become aware of the incoming shots and would strafe to the left.

TABLE III
REWARD FUNCTION FOR THE HIERARCHICAL RL TASK

Reward	Value
Item collected	1.0
Enemy kill	1.0
Accurate shot	1.0
Distance traveled (0.4m/step)	-0.0000002

The trial 9 bot learned the worst performing combat strategy out of all the trials. However, the bot was still able to learn to kill the enemy bots but at a lower ability as compared to trials 5 and 6 bots.

In summary, the results in this section suggest that the Sarsa(λ) algorithm can be used to learn a combat controller for FPS bots. The findings indicated that a number of bots learned successful behaviors and the best scoring bots were able to learn a more successful combat strategy than the state machine bot. Observation showed that different styles of combat could be produced, each of which varied depending on the parameters.

C. General Purpose Task

1) Setup: Each experiment included one RL-type bot, one state-machine-controlled bot (termed StateMachine), and two nonreacting navigation bots. The StateMachine bot was programmed to navigate the environment, pick up items in its near vicinity, and follow and kill an enemy in sight. The StateMachine bot was coded with an aggressive type behavior as it would not back down from a fight, and would pursue the enemy to the end. The two nonreacting navigation bots were included to give the RL bots practice targets to learn combat. These bots navigated the environment at all times. Two different enemy types were also included to compel the RL bot to learn a generalized combat strategy.

Three different RL setups were used in these experiments: HierarchicalRL, RuleBasedRL, and RL. The HierarchicalRL setup uses hierarchical RL to combine a combat and navigation controller that was trained in the previous section. The HierarchicalRL controller learns when to use the combat or navigation controller by receiving the rewards listed in Table III. Collision and death penalties were not included as preliminary runs showed better performance without these penalties. A range of parameters were experimented with, and results were varying. In general, medium discount factors and eligibility traces were the best performing.

The state and action spaces for the HierarchicalRL controller were the combined set of the combat and navigation controller spaces as follows:

$$S = \left\{ \begin{array}{l} L\mathrm{Obj}_1, F\mathrm{Obj}_2, R\mathrm{Obj}_3, L\mathrm{Enemy}_4, \\ F\mathrm{Enemy}_5, R\mathrm{Enemy}_6, L\mathrm{Item}_7, \\ F\mathrm{Item}_8, R\mathrm{Item}_9 \end{array} \right\}, \qquad s_i \in \{0, 1, 2\}$$

$$A = \{\mathrm{NavCntr}_1, \mathrm{ComCntr}_2\}$$

where prefixes L, F, and R stand for left, front, and right, respectively, and correspond to sensors relative to the bot's position and orientation. The total number of state–action pairs in the policy table was 39 366.

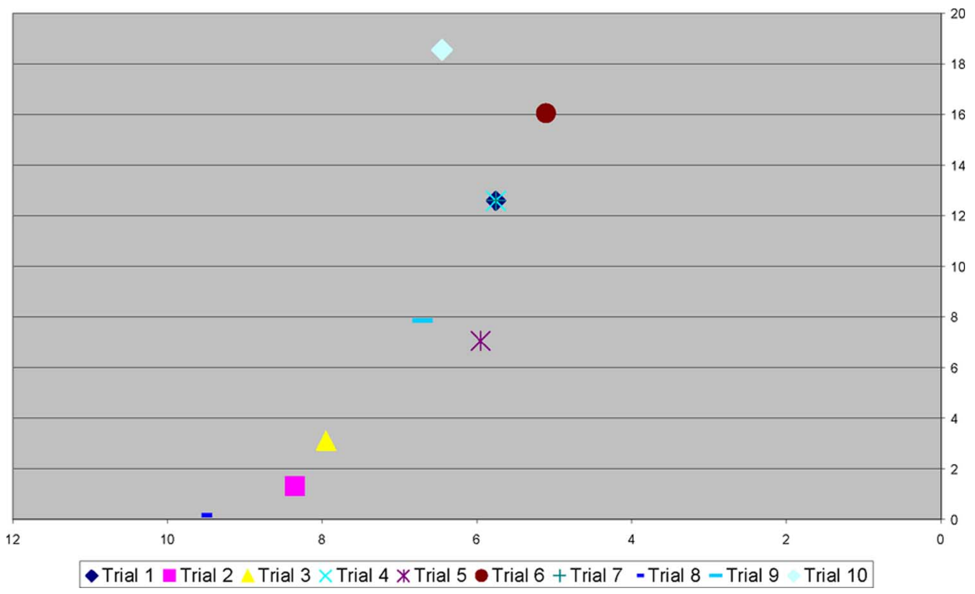

Fig. 10. Pareto front for deaths versus kills.

TABLE IV
REWARD FUNCTION FOR THE RL TASK

Reward	Value
Item collected	1.0
Enemy kill	1.0
Accurate shot	1.0
Collision	-0. 000002

The RuleBasedRL controller uses the same previously trained navigation and combat controllers as the HierarchicalRL controller. The RuleBasedRL controller uses rules to determine when to use the combat or navigation controller. When an enemy is in sight the combat controller is used, otherwise the navigation controller is used.

The RL controller learns the entire task of combat and navigation from scratch, using a flat RL structure. The state space is identical to the HierarchicalRL state space previously listed. The action space consists of all possible actions the bot can take as follows.

The number of state–action pairs for the RL task is 137 781, more than three times larger than the HierarchicalRL table. Table IV lists the rewards and corresponding values for the RL bot. A reward was given to the RL bot when it picked up an item, killed an enemy bot, and when it accurately shot an enemy. A small penalty was given when the RL bot collided with an obstacle.

All three RL bots were trained for 5000 iterations in both the arena and maze environments, where an iteration was a single step of the game. The learned policies were replayed for 5000 iterations, with learning disabled ($\alpha = 0$), 20 times. The three RL bots were then compared against the StateMachine bot from the RuleBasedRL experiments, as well as against a random bot (termed Random). The Random bot randomly selects actions from the previously defined action set in the RL controller setup.

Initial experiments were performed with a wide range of parameters to find the best solutions. The RL parameters used for the HierarchicalRL controller were $\alpha = 0.2, \varepsilon = 0.2, \gamma = 0.35$, and $\lambda = 0.65$, and the RL controller were $\alpha = 0.2, \varepsilon = 0.1, \gamma = 0.9$, and $\lambda = 0.2$. The best scoring policy was then replayed 20 times with different random seeds and the average, minimum, and maximum results for each bot were recorded. The recorded results included the number of deaths, the number of enemy kills, shooting accuracy, the number of collisions with world geometry, distance traveled, and number of items collected (maze map only).

2) Statistical Results and Discussion: Table V displays the results from the arena map experiment. The average number of deaths is similar for all five bots, while the number of kills varied greatly. The greatest variance in deaths is seen in the RL bot with a maximum of 13 deaths and minimum of 0 in one trial. The variance is due to the instability of the learned policy, where the bot was more likely to ignore enemies and slide through the environment. The similarity of performance compared to the Random bot also demonstrates the variability of the RL bot. The HierarchicalRL and RuleBasedRL bots were more stable in terms of death count, showing similar results to the StateMachine bot.

The StateMachine bot clearly showed its skill in combat with a very high score for kills in the arena map. The RuleBasedRL bot followed with half as many average kills, but with a high score of 12 kills in one run. The HierarchicalRL bot was close behind the RuleBasedRL bot, with an average of five kills and high score of nine kills. While the StateMachine bot recorded a high number of kills, we have observed that the kills were mainly from the two nonreacting bots. Since the number of deaths for the StateMachine bot was similar to the RuleBasedRL bot, we know that the RuleBasedRL bot killed the StateMachine bot as many times as it was killed. Similar results were seen against the HierarchicalRL bot. The similar death count therefore implies that the RuleBasedRL and HierarchicalRL bot were formidable opponents to the StateMachine bot when head-to-head, but the underlying combat controller was less effective

TABLE V
STATISTICAL RESULTS FOR ARENA MAP EXPERIMENT

Bot	Value	Deaths	Shots	Hits	Accuracy (%)	Kills	Collisions	Distance (m)
HierarchicalRL	Average	3.6	70.2	56.9		5.0	520.6	112.3
	Minimum	1.0	37.0	25.0	61.0	2.0	8.0	109.7
	Maximum	5.0	108.0	87.0	91.9	9.0	1430.0	114.0
RuleBasedRL	Average	3.6	110.1	84.8	76.4	7.4	27.7	111.3
	Minimum	2.0	68.0	40.0	58.8	3.0	0.0	101.9
	Maximum	5.0	154.0	137.0	89.0	12.0	510.0	114.9
RL	Average	4.2	155.0	27.0	21.4	1.9	2147.8	91.6
	Minimum	0.0	56.0	2.0	0.7	0.0	59.0	25.3
	Maximum	13.0	299.0	54.0	38.6	6.0	4604.0	144.1
Random	Average	3.9	292.0	4.5	1.5	0.1	127.0	142.5
	Minimum	0.0	281.0	1.0	0.3	0.0	0.0	140.3
	Maximum	9.0	306.0	12.0	4.0	1.0	244.0	146.6
StateMachine	Average	2.8	272.1	136.9	50.4	14.1	26.0	143.9
	Minimum	2.0	221.0	96.0	37.4	10.0	0.0	130.2
	Maximum	5.0	345.0	189.0	60.2	19.0	131.0	163.6

against the nonreactive navigation bots. The RL bot managed a maximum of six kills, indicating that some form of combat behavior was learned. This result was surprising considering the high number of deaths similar to the Random bot.

While the StateMachine outperformed the other bots in kills, the HierarchicalRL and RuleBasedRL bots significantly outperformed the StateMachine bot in shooting accuracy. The HierarchicalRL controller learned the highest accuracy with an average of 81% and maximum of 91.9%. This accuracy was greater than the RuleBasedRL bot's average accuracy of 76.4%, and maximum of 89%. The HierarchicalRL controller had a higher shooting accuracy as it learned not to use the combat controller when the bot would shoot and miss. This situation occurred when there were enemy bots far and to the right of the bot's current position. However, because the HierarchicalRL bot did not initiate combat in this state, the enemy would go out of sight range or have the advantage in following the fleeing bot, which is the main reason the HierarchicalRL bot killed fewer enemies than the RuleBasedRL. This behavior can be likened to an expert FPS player's type of strategy, where only near guaranteed shots are taken; otherwise a safe position is sought. The StateMachine bot had a much lower accuracy than the HierarchicalRL and RuleBasedRL bots. Although the StateMachine's accuracy was quite low, the bot shot many times more than the higher accuracy bots, therefore having the advantage in combat kills. Due to the large differences in shooting accuracies, it is expected that the HierarchicalRL and RuleBasedRL bots would perform better than the StateMachine bot in scenarios where ammo was limited, a common situation in most commercial FPS games. The RL bot exhibited a very low accuracy rating. The RL bot learned to shoot often, sometimes even shooting when no enemy was in sight. However, the RL bot's accuracy was still higher than the Random bot. This difference shows that the RL bot was able to learn a combat strategy that consisted of shooting as often as possible, generally when enemies were in sight, in the hope of taking down the enemy. This type of combat behavior is similar to beginner FPS players as they tend to constantly shoot haphazardly in times of danger.

The StateMachine and RuleBasedRL bots have similar, very low average collision counts (around 26 and 27.7, respectively). The HierarchicalRL bot has a much higher average count of 520.6. The RL bot had the highest number of collisions in this experiment, with an average of 2147.8, and maximum of 4604.

The StateMachine and Random bots traveled similar distances during the experiments. This result was expected from the similar number of collisions, as number of collisions is correlated with distance traveled. The exception to this rule is when the bots slide across a wall, which will increase the collision count, but the bot will still move a small distance (although less than a full step forward). The RL bot was the only trial where distance was not consistent, again showing the instability of this controller. The HierarchicalRL and RuleBasedRL controllers have consistent distances in average, minimum, and maximum indicating that the recorded collisions were sliding wall collisions as opposed to head-on collisions where the bots were stuck. The type of collisions was confirmed through observation of the replays.

The maze map environment was significantly more complex than the arena map, and therefore less contact between bots occurred than previously. The RL parameters used for the HierarchicalRL controller were $\alpha = 0.2, \varepsilon = 0.1, \gamma = 0.5$, and $\lambda = 0.2$, and the RL controller parameters were $\alpha = 0.2, \varepsilon = 0.1, \gamma = 0.4$ and $\lambda = 0.3$. Table VI displays the results for the maze map experiment with the additional statistic of item collection.

The average number of deaths is similar across all bots in the maze map experiment. The HierarchicalRL bot had the highest average death count, very closely followed by the RL bot. The RuleBasedRL bot scored similarly to the StateMachine, while the Random bot had the least number of average deaths. Overall, the death statistics were similar for all five bots, indicating that interaction occurred evenly between the bots during the replay games.

A good performance is seen from the StateMachine bot in enemy kills, with an average of 9.8 kills per trial, and a maximum of 15. The HierarchicalRL and RuleBasedRL bots performed similarly in average kills. The RuleBasedRL bot also scored slightly higher in maximum kills (8) than the HierarchicalRL bot (7). The gap between the RuleBasedRL and HierarchicalRL bot in average kills was greatly decreased in the maze

TABLE VI
STATISTICAL RESULTS FOR MAZE MAP EXPERIMENT

Bot	Value	Deaths	Shots	Hits	Accuracy (%)	Kills	Collisions	Distance (m)
HierarchicalRL	Average	3.1	59.5	50.6	84.4	3.9	419.9	91.6
	Minimum	0.0	16.0	12.0	65.8	1.0	115.0	58.3
	Maximum	6.0	103.0	80.0	94.7	7.0	675.0	106.9
RuleBasedRL	Average	2.3	71.2	56.0	81.4	4.4	74.0	107.0
	Minimum	0.0	9.0	7.0	43.6	0.0	13.0	100.8
	Maximum	5.0	165.0	90.0	97.0	8.0	131.0	109.0
RL	Average	3.0	46.2	17.8	38.7	1.1	2011.6	110.3
	Minimum	0.0	6.0	0.0	0.0	0.0	125.0	20.3
	Maximum	7.0	81.0	38.0	76.9	3.0	4465.0	156.4
Random	Average	1.3	292.9	4.1	1.4	0.1	348.8	139.5
	Minimum	0.0	283.0	1.0	0.3	0.0	81.0	130.5
	Maximum	8.0	303.0	10.0	3.5	1.0	971.0	144.0
StateMachine	Average	1.8	187.6	102.4	54.9	9.8	763.2	121.5
	Minimum	0.0	121.0	57.0	40.9	5.0	218.0	82.2
	Maximum	5.0	289.0	151.0	67.9	15.0	2765.0	149.6

map compared to the arena map. The RL bot was unable to learn a good combat strategy in the maze map, with an average of 1.1 kills per trial, and maximum of three kills. As in the arena map, both the kill and death scores of the RL bot are similar to that of the Random bot, indicating the RL bot had difficulty learning a good combat strategy.

Similar to the arena map, the HierarchicalRL and RuleBasedRL bots significantly outperformed the StateMachine bot in shooting accuracy. The HierarchicalRL bot had an average accuracy of 84.4%, and maximum of 94.7%. The RuleBasedRL bot performed slightly worse than the HierarchicalRL, with an average of 81.4%, but higher maximum of 97%. As expected, the StateMachine bot had a similar accuracy as in the arena map. Again the StateMachine bot shot many more times than the HierarchicalRL and RuleBasedRL bot, explaining the higher kill rate. The Random bot shot even more times than the StateMachine, but achieved a maximum accuracy of only 3.5%. The RL bot achieved an average of 38.7%, and maximum accuracy of 76.9%, showing that it was able to learn some part of combat as these scores were significantly higher than the Random bot.

The collision count in the maze map displays a different trend to the arena map. The results show that the RuleBasedRL bot had the lowest overall collision count. The HierarchicalRL bot had significantly more collisions than the RuleBasedRL bot, while the StateMachine bot performed poorly in collisions in the maze map compared to the arena map. The worst performing bot in collisions was the RL bot, with an extremely high average, although the minimum score was quite low in comparison.

Despite the high number of collisions, the StateMachine bot managed to travel a good distance through the maze with an average of 121.5 m, and maximum of 149.6 m. The HierarchicalRL and RuleBasedRL bots performed similarly with averages of 91.6 and 107 m, respectively. However, the RL bot traveled the greatest distance out of all the bots in a single trial with maximum of 156.4 m. The highest scoring in average distance traveled was the Random bot (139.5 m), as random actions allowed the bot to move around in a small space, frequently choosing to move backwards and forwards.

Although the RL controller had difficulty learning combat in the maze area, it did surprisingly well in the item collection task. The RL bot performed the best in average and maximum items collected. The RL bot's success in item collection, distance traveled, and to a lesser extent accuracy, indicates the necessity to split different tasks of the AI into subcontrollers. The RuleBasedRL also performed well in the item collection task. The HierarchicalRL bot performed very badly in the item collection task, which was surprising considering the good result of the RuleBasedRL bot. Even the Random bot performed slightly better than the HierarchicalRL bot. The StateMachine was able to collect an average of 7.2 items per run, with maximum of 11. Considering the rules forcing the StateMachine bot to collect items nearby, the RL bot was able to learn a very good item collection strategy.

3) Observational Results and Discussion: This section examines the paths traveled by the bots during the experiments. The paths are displayed for the best scoring bots from the maximum rows in Tables V and VI. The number of runs displayed in the path figures varies depending on how many times the bot died during the 5000 replay iterations.

Fig. 11 displays the route taken by the HierarchicalRL bot. The HierarchicalRL bot's first run mostly used the navigation controller, even when multiple enemies were in view. The combat controller was occasionally chosen, and a few shots were fired; however, the bot ended up navigating away from the fight with the enemy chasing and killing the HierarchicalRL bot quickly. On the next run (i.e., spawning of the bot) the HierarchicalRL bot took down three enemies in quick succession before being killed by the StateMachine enemy. The HierarchicalRL bot killed an enemy soon into the third run, then proceeded to navigate the walls until an enemy came into sight. The HierarchicalRL bot tracked the enemy from behind, but the enemy reacted and killed the HierarchicalRL bot first. The last two runs were similar, with the HierarchicalRL bot killing a few enemies, and navigating when no bots were in sight.

Although the RuleBasedRL bot used the same combat and navigation controllers as the HierarchicalRL bot, a different type of behavior was observed (see Fig. 12). While the Hierarchi-

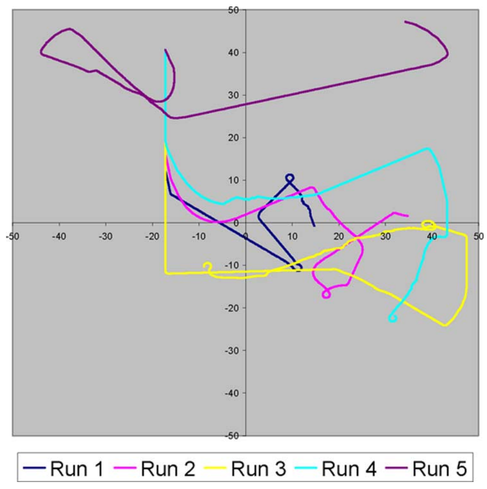

Fig. 11. HierarchicalRL bot paths from replay of the arena map experiment with random seed 102.

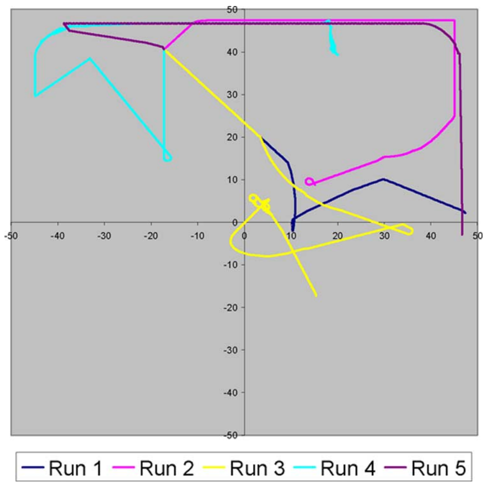

Fig. 13. RL bot paths from the replay of the arena map experiment with random seed 101.

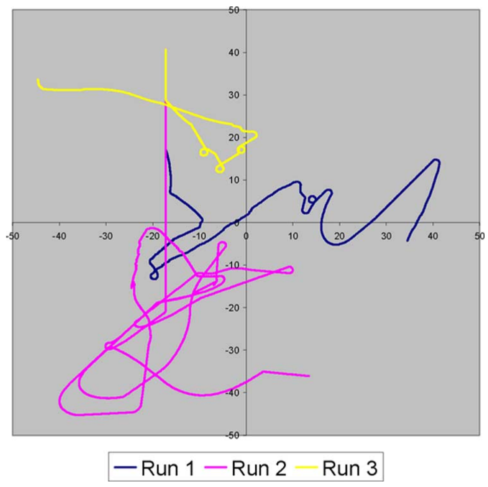

Fig. 12. RuleBasedRL bot paths from the replay of the arena map experiment with random seed 120.

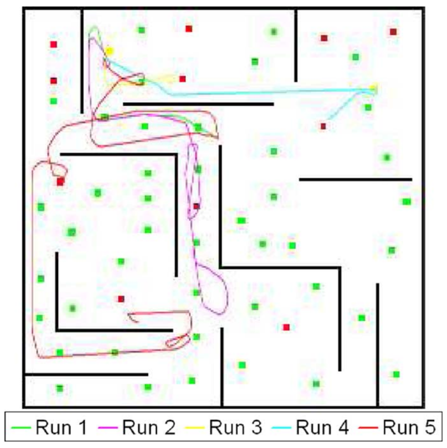

Fig. 14. HierarchicalRL bot paths from replay of the maze map experiment with random seed 112.

calRL bot tended to shy away from multiple enemies, the RuleBasedRL bot was very aggressive and would fight any in sight. Although it was expected that the RuleBasedRL bot would be an aggressive fighter due to the controlling rules, it is interesting to note that the same underlying controllers can be used to produce varying behaviors by using different high level controllers. The RuleBasedRL bot had three runs in total, the first being the shortest lived, with only a few kills before dying. In the second run, the RuleBasedRL bot went on a killing spree with a total of eight kills. During this run, the RuleBasedRL bot tracked and killed the enemy while it was distracted shooting a nonresponsive bot. In the third run, the bot killed a few enemies before the total number of iterations was completed.

The RL bot displayed very different behavior from the previous two, although some similarities were seen during combat (see Fig. 13). The RL bot began by moving diagonally through the arena before encountering multiple enemies. The RL bot headed towards an enemy while shooting, and then did a loop and began traveling backwards. The bot continued on the backwards path until colliding with the eastern wall, where it became stuck and was shot down by the enemy. On the second run, the RL bot navigated the northern wall by strafing left, until encountering the corner where it turned and started moving forward along the eastern wall. An enemy came in sight a quarter of the way down the eastern wall, and the RL bot reacted by following and shooting. The RL bot was then killed by the enemy. In the third run, the RL bot killed the StateMachine bot and a nonreacting bot, and took down a third bot to 10% health before re-encountering the StateMachine bot and running away. At one point in the fourth run, the RL bot had trouble navigating the walls, deciding to shoot the wall and travel back and forth before continuing on. During the last run, the RL bot killed the enemy and then navigated the northern and eastern walls until the iterations were completed.

The HierarchicalRL bot in the maze map experiments started off using a combination of the navigation and combat controllers to navigate (see Fig. 14). The bot stayed close to the walls, sometimes sliding along, which caused a high collision count even though the bot did not become immovable on the walls. The first enemy encountered was the StateMachine bot, which proceeded in a close head-to-head fight, but the StateMachine bot won with five hit points to spare. In the next three runs, the HierarchicalRL

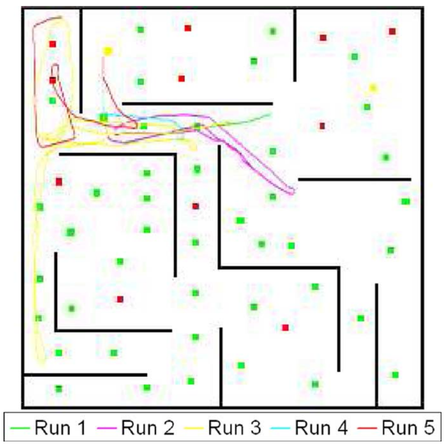

Fig. 15. RuleBasedRL bot paths from the replay of the maze map experiment with random seed 120.

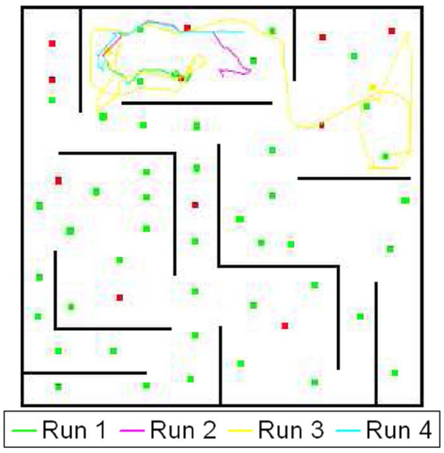

Fig. 16. RL bot paths from the replay of the maze map experiment with random seed 101.

bot mainly navigated the area, killed the StateMachine bot once per run, and then was later killed. In the last run the HierarchicalRL bot killed the StateMachine bot, and two nonresponsive bots. Between fights, the HRL bot navigated the environment. At one point at the end of the run, the HierarchicalRL bot was heading towards an item using the navigation controller, but when it came close to the item, the combat controller was used that caused the bot to turn away.

The RuleBasedRL bot started off similarly to the HierarchicalRL, with a short run and a close but unsuccessful head-to-head fight with the StateMachine bot (see Fig. 15). The second run was more successful, with three enemy kills in a short time, one of which was the StateMachine bot. The third run was the longest run of all five, with the RuleBasedRL bot navigating a fair portion of the maze, picking up items, and killing three enemies. In contrast, the fourth run did not last long, with the StateMachine killing the RuleBasedRL bot soon after it respawned. During the last run, the RuleBasedRL bot killed the StateMachine and a nonresponsive bot, before navigating until the iterations were completed.

Fig. 16 displays the routes the RL bot took in the maze map experiment. As seen from the statistical data, the RL bot performed well in navigation and item collection in the maze environment. The RL bot navigated well in the first run, but showed the lack of combat skills when the enemy StateMachine bot came into view. The RL bot shot at the wall instead of the enemy until it was inevitably killed. On the next run, the RL bot had more success in combat, taking the StateMachine bot down to 20% health before being killed. During all four runs, the RL bot showed an interesting navigation and item collection strategy. The bot generally favored strafing left and right to navigate close walls, and also used this zigzagging movement to collect items in the area. During the third run, the RL bot successfully killed the StateMachine bot twice, showing that in certain state setups it was able to learn a very good combat strategy. Both fights were head-to-head and the RL bot targeted and headed straight towards the enemy shooting accurately. The RL bot killed a nonresponsive bot at the beginning of the fourth run, traveling backwards while shooting at the enemy.

A distinct advantage of using RL over state machines and rule-based systems is evident in the results. The state machine bot behaves the same under similar conditions; while the RL bots are also deterministic they are able to react differently in similar conditions. Because of how the state space is defined, the RL bots can react to multiple enemies (up to three) in its field of view, while the StateMachine bot only reacts to one enemy. For state-machine-controlled bots to react to multiple enemies, this behavior needs to be implicitly coded into the rules and may be difficult to tune.

VI. Conclusion

This paper has shown that RL can successfully be applied to a simplified FPS game. Results indicated that the tabular Sarsa(λ) algorithm can be used to learn the FPS bot behaviors of navigation and combat. The navigation controller was able to learn a decent strategy for path planning, although not to the ability of the standard A* algorithm. The combat controller learned a strategy that beat the state-machine-controlled opponent, showing the strength of RL algorithms.

In the general purpose task experiments, both the arena and maze maps showed similar trends in combat and navigation statistics, indicating the robustness of the RL controllers. The hierarchical RL and rule-based RL controllers performed significantly better than the flat RL controller in combat and navigation skills. The hierarchical RL bot performed best in the shooting accuracy objective, outperforming all other bots in the experiment. The rule-based RL bot performed slightly better in the other objectives than the hierarchical RL bot, however the observational results showed that the hierarchical RL bot had a wider range of behaviors in combat scenarios. For example, in some situations with multiple enemies, the hierarchical RL bot would flee the scene whereas the rule-based RL bot tended to stay with the fight.

The flat RL controller was able to learn some basic combat and good navigation behaviors, however not to the degree of the hierarchical RL or rule-based RL. On the other hand, the flat RL controller was able to excel in the item collection task, outperforming all bots including showing a slightly better performance than the state machine bot, which was explicitly coded to pick up items in its path.

The hierarchical RL and rule-based RL controllers displayed characteristics of advanced FPS players, by initiating fights when they were confident of a win. In comparison, the RL controller displayed characteristics of beginner players, such as panicky shooting in all directions. Both the hierarchical RL and rule-based RL methods show potential for creating diverse behavior sets for FPS bots. Although the rule-based RL bot showed slightly better statistical scores, the hierarchical RL bot has potential for creating more diverse and less predictable behaviors. This is due to the fact that the action set of the hierarchical RL bot is not defined by hard-coded rules as those underpinning the behaviors of the rule-based RL bot.

Although a simple hierarchical structure is used in the experiments, this can easily be extended to include other features of bot AI such as operating vehicles or using portals. Future work will increase the complexity of the bots behaviors by increasing the state and action spaces. Interpolated tables will be used for smoother state transitions, and the bots will then be tested in a commercial FPS game.

REFERENCES

[1] G. Tesauro, “Temporal difference learning and TD-gammon,” *Commun. ACM*, vol. 38, no. 3, pp. 58–68, 1994.

[2] N. Sturtevant and A. White, “Feature construction for reinforcement learning in hearts,” in *Proc. 5th Int. Conf. Comput. Games*, 2006, pp. 122–134.

[3] D. S.-C. Dalmau, *Core Techniques and Algorithms in Game Programming*. Indianapolis, IN: New Riders, 2003.

[4] D. Isla, “Handling the complexity in Halo 2,” in *Proc. Game Developers Conf.*, 2005.

[5] J. Jones, “Benefits of genetic algorithms in simulations for game designers,” Master’s thesis, Schl. Inf., Univ. Buffalo, Buffalo, NY, 2003.

[6] C. A. Overholtzer and S. Levy, “Evolving AI opponents in a first-person-shooter video game,” in *Proc. 20th Nat. Conf. Artif. Intell.*, Pittsburgh, PA, 2005, pp. 1620–1621.

[7] S. Bakkes, P. Spronck, and E. Postma, “TEAM: The team-oriented evolutionary adaptability mechanism,” in *Proc. Entertain. Comput.*, 2004, pp. 296–307.

[8] M. McPartland and M. Gallagher, “Learning to be a bot: Reinforcement learning in shooter games,” presented at the Artif. Intell. Interactive Digit. Entertain., Stanford, CA, 2008.

[9] M. McPartland and M. Gallagher, “Creating a multi-purpose first person shooter bot with reinforcement learning,” in *IEEE Symp. Comput. Intell. Games*, 2008, pp. 143–150.

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

[11] J. Manslow, “Using reinforcement learning to solve AI control problems,” in *AI Game Programming Wisdom 2*, S. Rabin, Ed. Hingham, MA: Charles River Media, 2004.

[12] A. H. Tan and D. Xiao, “Self-organizing cognitive agents and reinforcement learning in multi-agent environment,” in *Proc. Int. Conf. Intell. Agent Technol.*, Compiegne, France, 2005, pp. 351–357.

[13] J. Bradley and G. Hayes, “Group utility functions: Learning equilibria between groups of agents in computer games by modifying the reinforcement signal,” in *Proc. Congr. Evol. Comput.*, 2005, vol. 2, pp. 1914–1921.

[14] S. Nason and J. E. Laird, “Soar-RL: Integrating reinforcement learning with soar,” *Cogn. Syst. Res.*, vol. 6, no. 1, pp. 51–59, 2005.

[15] J. H. Lee, S. Y. Oh, and D. H. Choi, “TD based reinforcement learning using neural networks in control problems with continuous action space,” in *Proc. IEEE World Congr. Comput. Intell.*, Anchorage, AK, 1998, vol. 3, pp. 2028–2033.

[16] K. Merrick and M. L. Maher, “Motivated reinforcement learning for non-player characters in persistent computer game worlds,” in *Proc. ACM SIGCHI Int. Conf. Adv. Comput. Entertain. Technol.*, Los Angeles, CA, 2006.

[17] T. Graepel, R. Herbrich, and J. Gold, “Learning to fight,” in *Proc. Int. Conf. Comput. Games, Artif. Intell. Design Educ.*, 2004, pp. 193–200.

[18] K. Merrick, “Modeling motivation for adaptive nonplayer characters in dynamic computer game worlds,” *ACM Comput. Entertain.*, vol. 5, no. 4, 2008, DOI: 10.1145/1324198.1324203.

[19] S. Cohen, O. Maimon, and E. Khmlenitsky, “Reinforcement learning with hierarchical decision making,” in *Proc. 6th Int. Conf. Intell. Syst. Design Appl.*, 2006, vol. 3, pp. 177–182.

[20] T. Watanabe and Y. Takahashi, “Hierarchical reinforcement learning using a modular Fuzzy model for multi-agent problem,” in *Proc. IEEE Int. Conf. Syst. Man Cybern.*, Montreal, QC, Canada, 2007, pp. 1681–1686.

[21] M. Huber, “A hybrid architecture for hierarchical reinforcement learning,” in *Proc. IEEE Int. Conf. Robot. Autom.*, San Francisco, CA, 2000, vol. 4, pp. 3290–3295.

[22] B. Blumberg, M. Downie, Y. A. Ivanov, M. Berlin, M. P. Johnson, and B. Tomlinson, “Integrated learning for interactive synthetic characters,” *ACM Trans. Graph.*, vol. 21, no. 3, pp. 417–426, 2002.

[23] M. Vasta, S. Lee-Urban, and H. Munoz-Avila, “RETALIATE: Learning winning policies in first-person shooter games,” in *Proc. 17th Innovative Appl. Artif. Intell. Conf.*, 2007, pp. 1801–1806.

[24] J. Laird, “It knows what you’re going to do: Adding anticipation to a Quakebot,” in *Proc. 5th Int. Conf. Autonom. Agent*, 2001, pp. 385–392.

Michelle McPartland received the B.S. degree in information technology (honors) from the University of Canberra, Canberra, A.C.T., Australia, where she specialized in genetic algorithms and neural networks. She is currently working towards the Ph.D. degree in computer science at the University of Queensland, Brisbane, Qld., Australia.

She is concurrently working for 2K Marin as an AI Programmer. Her research is focused on using machine learning techniques for first person shooters AIs to make them more believable and fun to play against.

Marcus Gallagher (M’99) received the B.Comp.Sc. and Grad.Dip.Sc. degrees from the University of New England, Armidale, N.S.W., Australia, in 1994 and 1995, respectively, and the Ph.D. degree in computer science from the University of Queensland, Brisbane, Qld., Australia, in 2000.

He is a Senior Lecturer in the Complex and Intelligent Systems Research group at the School of Information Technology and Electrical Engineering, University of Queensland. His main research interests are metaheuristic optimization and machine learning algorithms, in particular techniques based on statistical modeling. He is also interested in biologically inspired algorithms, methodology for empirical evaluation of algorithms, and the visualization of high-dimensional data.