FPS Bot Artificial Intelligence with Q-Learning

Vladislav Gordiyevsky and Kyle Joaquim
Department of Computer Science, University of Massachusetts Lowell
Lowell, MA 01854

***Abstract*—Innovation has stagnated in artificial intelligence implementations of first-person shooter bots in the video games industry. We set out to observe whether reinforcement learning could allow bots to learn complex combat strategies and adapt to their enemies' behaviors. In a general approach, a simple combat environment and a shooter bot with basic functionality were created as a testbed; using this testbed, q-learning was implemented to allow for updating of the bot's policy for choosing high-level combat strategies. Multiple tests were run with different numbers of iterations of a combat scenario in which the bot with the q-learning implementation faced off against a simple reaction-based agent. The learning bot updated its policy to make strategic decisions and increase its chances of winning, proving its ability to adapt to the behaviors of its opponents. The minor success of this particular test case indicates that the implementation of reinforcement learning abilities in first-person shooter bots is an option worthy of further exploration.**

Keywords—artificial intelligence, q-learning

I. Introduction

Adaptive bots – bots which change their behaviors to best suit the situation – are not common in first-person shooter video games, despite the wide range of player skill levels. The most likely reason for this stagnation in artificial intelligence development in modern games is because of the unpredictability of learning in complex and dynamic environments, and since video games are commercial products, they are guided by a set of rules that tends to favor reliable customer satisfaction rather than experimentation. Thus, commercial video game development has tended to favor "rule-based systems, state machines, scripting, and goal-based systems" [2], which tends to lead to predictable behaviors, fine-tuning of parameters, and a necessity to write separate code for different behavior types [2][3]. Predictable behaviors can lead to players quickly learning and exploiting the behavior of their computer-controlled opponents, which in turn can lead to general boredom in single-player games. Thus, the possibility of creating agents that can adapt and change behaviors based on their environments in a commercial environment is an enticing concept for consumers.

Although commercial game development has stuck to reliable and tested methods, learning research in video game environments has seen a surge in recent years [2]. However, current research tends to employ purpose-built testbeds [2], use previously released game engines with little flexibility for future use or development [1] and employ action spaces with low-level functions [1][2][3]. This is a logical approach as a controlled and fully known environment can lead to discoveries in algorithm implementation and modification. The goal of our work was to explore the possibility of integrating reinforcement learning artificial intelligence via q-learning in a modular development environment, implementing an action space with higher-level functions in order to achieve "consistent and controlled unpredictability" in our implementation (in terms of bot behavior), and creating a foundation for future research.

II. Literature Review

One approach to employing reinforcement learning in a video game environment to combat predictability utilizes a technique called dynamic scripting, implemented by Policarpo, D & Urbano, Paulo & Loureiro, T [3]. In this approach, a series of rules are created outlining actions to be taken in the case of certain conditions being met. The agent then selects a subset of these rules – a script – to follow based on rule weights that are updated after each learning episode. All rules within a script are given a reward based on the measured success of the script. A statically coded agent was used as the opponent for the learning episodes. Within 100 matches, the agent was able to find the optimal policy or script for defeating its opponent, demonstrating that an agent could easily learn the optimal policy to face off against a given opponent simply by utilizing the same conditional rules already implemented in first-person shooter bots.

Another approach to implementing learning in first-person shooter games taken by Michelle McPartland and Marcus Gallagher employs a tabular Sarsa reinforcement learning algorithm, which allows an agent to speed up learning and even learn sequences of actions by using eligibility traces [2]. This bot was trained with low-level actions in navigation, item collection, and combat, using sensors to update its state after each action. The bot was able to outperform a statically programmed state machine bot within just 6 trials; however, the training of low-level actions did not lead the bot to account for all nuances of the environment, nor display higher-level rational behavior such as running away when low on health or hiding in cover, which would be favorable in modern video game environments.

Another implementation of reinforcement learning used deep neural networks and q-learning to train a bot for the video game *DOOM* [1]. This project employed vision-based learning techniques, using pixel data from the game as input. Using these methods, they were able to successfully train a bot to navigate environments and fight by making rational decisions. However, the bot's action space was limited to

turning, moving, and shooting, and did not focus on adapting strategies to different opponents nor environments, but rather the ones already present in the original DOOM game which released in 1993.

All three of these projects took different approaches to implementing reinforcement learning in a first-person shooter; however, no single one of them investigated using higher-level actions to learn strategies rather than how to play the game. The agents utilized in their work employed action spaces consisting of actions such as moving, turning, and shooting. The ViZDoom project [1] and the project utilizing the Sarsa RL algorithm [2] were successful in creating learning agents that observably improved; however, the basis of these agents' functions on low-level actions meant the aim of their work was to investigate whether an agent can learn how to play the game rather than adapt new strategies.

III. Methodology

Our experiment and implementation consisted of creating our own testbed within the Unreal Engine, a popular and powerful modern games engine. The testbed consisted of two agents, one reaction-based and hard-coded to be aggressive, and a second learning agent utilizing the q-learning reinforcement learning algorithm. The objective of the testbed game mode was simply to eliminate the opponent via ranged combat. We chose the q-learning algorithm for the learning agent because of its simplicity and ease of integration within the Unreal Engine, along with its use of a learning rate and discount factor that could be easily modified between simulations.

The map for our experiment was small enough for both bots to find each other even through random wandering. The layout consisted of walls, floors, spawn locations, and a cover node graph overlay for the learning agent (see Fig. 1 & 2). The cover nodes signified covered locations on the map. Navigation between locations was handled by the engine and cover nodes consisted of a location vector and an array of connected nodes for the learning agent to move to and from. The map was symmetric to create an even playing field for both agents.

We built everything within the Unreal Engine using stock assets and one animation asset pack we modified from the Unreal Marketplace named the Advanced Locomotion Pack, created by user LongmireLocomotion. This asset pack significantly reduced the development time of our testbed and was modifiable for our needs.

Both agents had 100 health points and 20 rounds for their weapons. In the interest of time, we did not implement health or ammo pickups, and instead had health regenerate by 5 health points a second 5 seconds after not taking damage, along with unlimited reserve ammo. Reloading took 3 seconds, and the cooldown between successive shots was set to 0.20 seconds to prevent extremely rapid fire. A successful shot on an opponent dealt 5 damage. A small amount of shot variance was added for both agents (-1.0 to 1.0 degrees in their X, Y, Z aim vectors) in order to simulate natural aim. There was no game timer, instead the overall fitness of an agent was gauged by the amount of eliminations accrued while the learning agent was in its exploitation phase.

Each simulation round consisted of a set number of exploration iterations for the learning agent with a predefined learning rate and discount factor, after which the learning agent transitioned to its exploitation phase and eliminations were counted for about thirty minutes per simulation round.

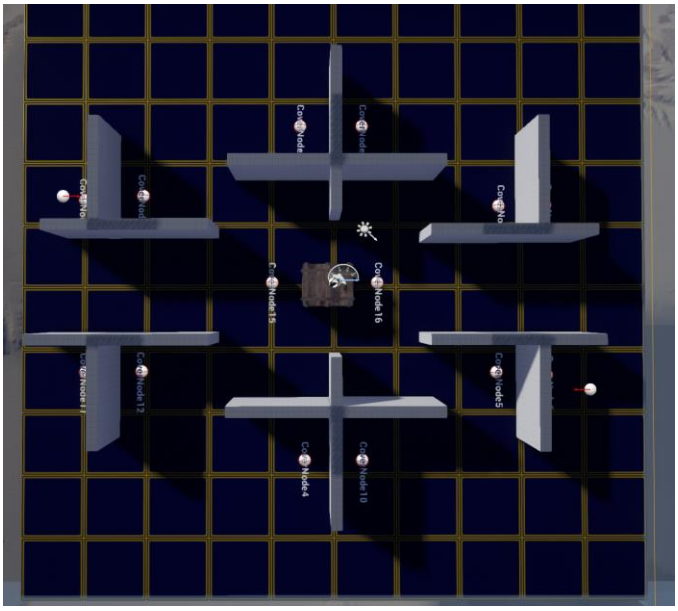

Fig. 1. The game map viewed overhead.

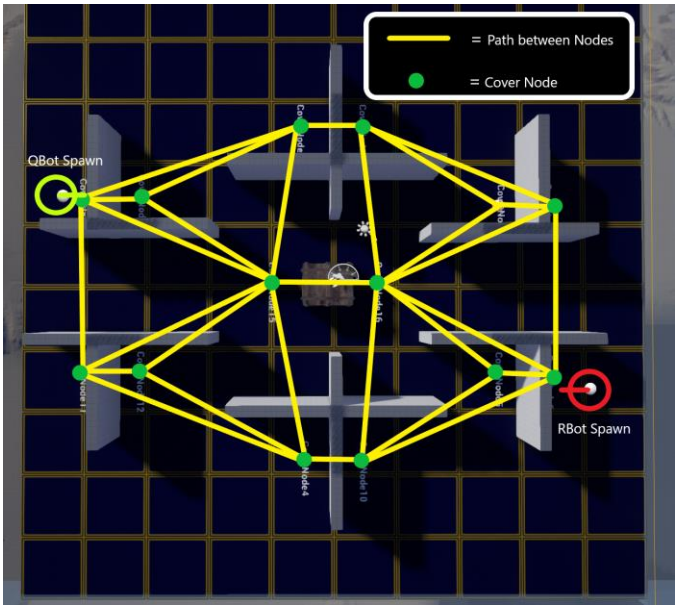

Fig. 2. The game map with cover node overlay and connected paths.

The reaction-based agent's behavior was governed by a simple behavior tree. The reaction-based bot was to wander randomly when no opponent was in sight, shoot on sight, follow its opponent when the opponent ran out of sight, patrol the last known location for a short while and go back to wandering after searching the last known location of its opponent. The directive to shoot on sight overrode all other actions. The reaction-based agent was designed to be aggressive in order to see if the learning agent could learn a strategy to compete with it, given a set of offensive and defensive actions.

Both agents had a sensor component with a 75-degree peripheral vision angle and 3000 Unreal Unit range, which allowed them to see each other across the map. This decision was made in order to simulate a human player's range of vision. Both bots also kept track of their opponent's last known location and updated it while their opponent was within sight. Both bots could also sprint when moving to a location, used primarily in the Move to Last Known Location function for both as it was primarily an aggressive action. The learning agent also had a collision mesh component extending about 25 Unreal Units around its skeletal mesh in order to update its state when being fired upon, in order to simulate an alarmed state.

The state space for the learning agent consisted of 5 boolean variables (see Fig. 3), which resulted in a total of 32 distinct states. The learning agent's action space consisted of 6 actions (see Fig. 4). The five Boolean variables that composed the learning agent's state space were mapped as a binary string. This string was enumerated and kept track of when updating the agent's Q-Table and used to map its reward table (see Fig. 5). The reward values chosen reflected rational decisions a player would make under the same circumstances, with large positive rewards for tactical behavior and large negative rewards for endangering behavior. Viable but less advantageous behavior was rewarded with values in between these. Reward values were kept in a range between -300 and 300, instead of -3.0 and 3.0 because the Unreal Engine tended to round off floating point values at about 7 points of precision.

We implemented the standard Q-Function (1) in order to update Q-Table values, with a dynamic reward function which rewarded the learning agent with 20 points for successfully hitting an enemy while firing and 200 for successfully eliminating the enemy, regardless of what state it was in. Likewise, the learning agent was rewarded -20 points for being shot, and -200 points for being eliminated regardless of what state it was in. The Q-Learning algorithm works by considering the current state of the agent, the action taken in that state, the next state the agent ends in, and the reward gained from performing that action. The reward is added to a prediction of future reward, calculated by taking the maximum Q-Value attainable from the ending state, multiplied by a discount factor which governs how much the agent valued future rewards opposed to current rewards. Finally, the current Q-Value is subtracted from this calculation (the purpose is to find the greatest change in reward values, not accumulate reward value) and multiplied by a learning rate which governs to what extent newly acquired information overrides old information. Furthermore, the Q-Learning algorithm is a model-free reinforcement learning algorithm, meaning it does not require a transition model to determine an optimal policy, but it does require training and a predetermined reward table. The intention behind a dynamic reward function was to create a bit of variance between simulations and see if it influenced how the agent learned strategies, as we were aiming to create "controlled unpredictability" on a small scale.

$$Q[s,a] \leftarrow Q[s,a] + \alpha\left(r + \gamma \max_{a'} Q[s',a'] - Q[s,a]\right) \quad (1)$$

STATE	DESCRIPTION
LowHp	Whether or not current health is below 30
LowAmmo	Whether or not current ammo is below 5
PlayerInView	Whether or not opponent is currently in sight
InCover	Whether or not agent is currently in cover (near a cover node)
BeingShot	Whether or not agent has been fired upon recently (5 second timer)

Fig. 3. State space for the learning agent.

ACTION	DESCRIPTION
Move Randomly (0)	Pick random point within navigable radius (2000 Unreal Units) and move to it
Aim and Shoot (1)	Set focus on enemy and fire a single round
Run to Cover (2)	If not In Cover: Move to cover node furthest from Last Known Location Else: Move to closest connected cover node
Move to Last Known Location (3)	Sprint within radius (150 Unreal Units) of Last Known Location
Reload (4)	Reload weapon (can be moving, but will break sprint)
Stay in Place (5)	Stand still at current location

Fig. 4. Action space for learning agent.

A singled learning step was defined as a loop (see Fig. 6). During the exploration phase, the learning agent would get its current state and perform a random action from its state space. While it was performing this action, the learning agent would calculate its reward. At the completion of the action, the agent would evaluate its ending state and update its Q-Table values. Because of the nature of its action space, with actions requiring a varying amount of time, the time step for each learning iteration was dynamic.

STATES						ACTIONS					
LOW HP	LOW AMMO	PLAYER IN VIEW	IN COVER	BEING SHOT	#	MOVE RAND. (0)	AIM AND SHOOT (1)	RUN TO COVER (2)	MV. TO L.K.L. (3)	RELOAD (4)	STAY IN PLACE (5)
0	0	0	0	0	0	100	-100	100	100	-100	10
0	0	0	0	1	1	-100	200	100	100	-100	-200
0	0	0	1	0	2	20	-100	100	100	20	100
0	0	0	1	1	3	-100	200	100	100	-100	-200
0	0	1	0	0	4	-100	300	50	20	50	-200
0	0	1	0	1	5	-100	300	50	50	-100	-200
0	0	1	1	0	6	-200	300	-100	50	50	-200
0	0	1	1	1	7	-200	300	100	-100	50	-200
0	1	0	0	0	8	50	-200	100	50	300	50
0	1	0	0	1	9	100	-100	200	-100	200	-200
0	1	0	1	0	10	-200	-200	-200	-200	300	100
0	1	0	1	1	11	100	-100	300	-100	200	-200
0	1	1	0	0	12	50	-200	50	50	300	-100
0	1	1	0	1	13	-100	-100	200	-100	200	-200
0	1	1	1	0	14	-100	-300	50	100	300	-100
0	1	1	1	1	15	-100	-100	300	-100	300	-200
1	0	0	0	0	16	50	-100	200	-200	50	200
1	0	0	0	1	17	-200	-200	300	-200	-200	-200
1	0	0	1	0	18	-100	-100	50	-200	50	200
1	0	0	1	1	19	-200	-200	300	-200	-200	-200
1	0	1	0	0	20	-200	300	200	200	-100	-200
1	0	1	0	1	21	-200	300	200	-100	-100	-200
1	0	1	1	0	22	-200	300	50	50	50	50
1	0	1	1	1	23	-200	300	200	-100	-100	-200
1	1	0	0	0	24	-200	-200	200	-200	200	-200
1	1	0	0	1	25	-200	-200	200	-200	-100	-200
1	1	0	1	0	26	-200	-200	-100	-200	200	200
1	1	0	1	1	27	-200	-100	200	-200	-200	-200
1	1	1	0	0	28	-200	-300	50	-100	300	300
1	1	1	0	1	29	-200	-200	200	-100	-100	-200
1	1	1	1	0	30	-200	-300	-100	-100	300	300
1	1	1	1	1	31	-200	-200	300	-200	-300	-300

Fig. 5. The learning agent's Reward Table. The column in the middle signifies the enumerated state of the agent, while the values to the left of it signify the boolean variables associated with that state.

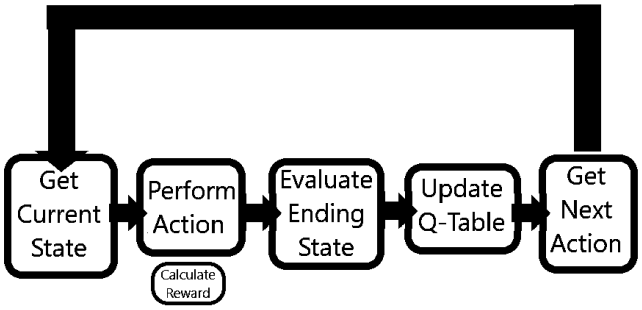

Fig. 6. The loop governing a single learning iteration of the learning agent.

We performed two series of tests. The first was to see if the learning agent could successfully learn to compete against the reaction-based agent and if the behavior learned was rational, in order to test our implementation. This testing was performed by running a succession of simulations with an increasing number of exploration iterations and a learning rate and discount rate of 0.5. The second series of tests was aimed at finding what amount of exploration iterations was required to converge to maximum reward values and gather enough Q-Table update data. These tests were carried out with varying learning rates and discount factors and then compared to one another. We expected the learning agent to learn an optimal strategy within at least 2500 exploration iterations, and to display defensive rational behavior such as running away to cover when low on health and reloading only when out of sight of the opponent agent.

IV. Results

For the first series of testing, we hit a roadblock in terms of bugs within the Unreal Engine having to do with collision mesh boundaries, collision traces, and ironing out reliable action function implementation. Because of these, a lot of early simulation results had to be discarded as collision detection and navigation was not reliable enough to accept the data. Since time was a factor for this project, we were able to perform three successful simulations for this portion of testing after fixing the bugs described above. The first simulation was run with 2500 exploration iterations. The results for the first simulation are displayed in Fig. 7.

ACTIONS->	0	1	2	3	4	5
State: 0	413.84	215.56	414.47	412.74	164.84	325.53
State: 1	132.29	515.45	351.12	342.68	125.21	36.64
State: 2	332.41	215.56	404.91	409.96	335.36	415.24
State: 3	211.53	515.53	351.88	268.83	211.95	18.68
State: 4	156.45	615.37	338.23	263.2	363.25	112.31
State: 5	133.08	619.49	323.85	304.16	183.93	114.62
State: 6	22.43	305.67	183.4	277.59	135.9	76.3
State: 7	59.25	611.86	391.3	113.45	270.96	55.39
State: 8	0	0	0	0	0	0
State: 9	0	0	0	0	0	0
State: 10	0	0	0	0	0	0
State: 11	0	0	0	0	0	0
State: 12	0	0	0	0	0	0
State: 13	0	0	0	0	0	0
State: 14	0	0	0	0	0	0
State: 15	0	0	0	0	0	0
State: 16	261.83	102.23	0	52.23	311.67	255.67
State: 17	59.11	114.51	610.12	115.15	67.72	91.18
State: 18	85.31	180.21	339.16	61.92	323.94	497.66
State: 19	73.91	72.93	599.03	102.17	102.4	109.66
State: 20	0	0	0	0	0	0
State: 21	95.35	630.19	500.82	213.72	201.4	113.02
State: 22	0	0	0	0	0	0
State: 23	96.68	617.11	509.11	201.75	182.31	90.69
State: 24	0	0	0	0	0	0
State: 25	0	0	0	0	0	0
State: 26	0	0	0	0	0	0
State: 27	0	0	0	0	0	0
State: 28	0	0	0	0	0	0
State: 29	0	0	0	0	0	0
State: 30	0	0	0	0	0	0
State: 31	0	0	0	0	0	0

Fig. 7. First simulation results with 2500 exploration iterations, learning rate of 0.5 and discount factor of 0.5.

We immediately noticed that about half of the Q-Table values were unpopulated, even after a long period of testing. While this is unusable data, it did tell us something important, that we needed to rework our exploration function. The values unpopulated in Fig. 7 were in the range of states 8-15, and 24-31. These state ranges all had to do with the boolean variable LowAmmo described in Fig. 3. Since the learning agent only fired a single round when randomly aiming and shooting, the chances of the agent randomly firing 15 rounds before reloading were incredibly slim. Therefore, we reworked our exploration phase so that whenever the learning agent respawned, it had a chance to spawn with a combination of low health, low ammo, and in cover on a randomly chosen cover node on the map in order to populate those missing values of the Q-Table. The second simulation results are shown in Fig. 8.

ACTIONS->	0	1	2	3	4	5
State: 0	387.65	217.65	417.5	387.64	216.72	327.65
State: 1	42.77	517.34	279.51	357.66	127.53	58.04
State: 2	333	217.65	417.64	415.6	336.87	417.65
State: 3	181.75	512.65	377.46	318.04	172.17	89.8
State: 4	217.03	614.77	364.41	142.74	365.78	82.59
State: 5	202.53	618.55	287.14	256.59	89.1	78.85
State: 6	88.23	615.67	131.53	338.52	237.99	21.82
State: 7	50.79	626.48	317.41	111.22	250.41	51.45
State: 8	297.63	117.65	390.11	366.48	617.65	362.27
State: 9	280.45	417.42	505.82	198.01	488.32	43.31
State: 10	-6.45	117.65	117.65	117.61	616.4	367.65
State: 11	367.06	399.87	582.82	103.22	481.32	108
State: 12	321.68	364.4	355.16	358.51	599.99	158.99
State: 13	168.32	417.67	394	206.31	469.87	6.48
State: 14	190.4	308.8	321.68	313.23	183.8	108.8
State: 15	163.2	392.17	425.28	111.87	507.69	-29.99
State: 16	351.26	185.08	516.27	88.23	353.2	516.61
State: 17	108.87	117.55	568.59	85.87	106.91	117.56
State: 18	143.66	190.45	0	60.76	337.79	258.8
State: 19	80.63	115.57	605.78	89.03	114.05	113.92
State: 20	58.8	0	248.83	230.7	108.8	0
State: 21	96.82	624.96	512.1	199.7	211.11	73.78
State: 22	0	0	0	0	0	0
State: 23	68.23	0	218.8	197.65	143.22	34.27
State: 24	18.24	114.1	498.02	70.84	483.7	105.23
State: 25	109.6	76.92	0	66.44	108.83	59.21
State: 26	58.8	0	108.8	58.8	0	388.24
State: 27	45.96	356.52	0	85.84	3.92	58.83
State: 28	0	0	0	0	0	308.83
State: 29	77.21	0	472.21	54.27	190.98	58.83
State: 30	0	0	0	0	0	0
State: 31	68.2	48.8	386.17	0	-7.1	0

Fig. 8. Second simulation results with 2500 exploration iterations, learning rate of 0.5, discount factor of 0.5, and reworked exploration phase.

The second simulation results were much more interesting, although there were still a few gaps in the table for state 22 and 30, which had to do with being in cover while having the opponent agent in view. The values from this simulation were very close to what could be expected from the reward table, with some variance for states where there were multiple optimal reward values, such as for states 1 and 2, where the learning agent was being fired upon and in cover, respectively. From the exploration phase, the agent learned that aiming back and shooting while being fired upon without having low health or ammo was optimal. Likewise, it learned that staying in place while in cover without seeing the opponent was optimal. Unfortunately, due to another bug that wasn't fixed until after the third simulation had begun, during the exploitation phase the learning agent kept performing the highest valued action in the Q-Table, which was to aim and shoot, without moving or reloading, rendering the exploitation phase inconclusive. This bug was fixed shortly and the results for the third simulation, shown in Fig. 9, yielded very promising results. We were able to run the exploitation phase and gather conclusive data. After 5000 exploration iterations, the learning agent learned an optimal policy, which dictated that the agent fire when the opponent is in view and the learning agent does not have low health or ammo, seen by the Q-Values for states 1 and 3-7, which is rational behavior. Furthermore, the agent learned to reload when low on ammo, not low on health and not in cover, regardless of being fired upon, shown by the Q-Values for state 9, but to run to the next closest cover when low on ammo, not low on health, being fired upon, and in cover, shown by the Q-Values for states 11, 13, and 15. After about 2 hours spent in the exploitation phase, the reaction-based bot scored 127 eliminations while the learning bot scored 91. While the learning agent did not win out over the reaction-based bot, it did display interesting behavior which was not expected, such as hiding in cover for a majority of the exploitation phase until the reaction based agent came around, firing some rounds, then running to cover again. The learning agent also surprisingly learned to sprint right to the last known location of its opponent after spawning, indicated by the Q-Value for state 0, which was not expected and resulted in the learning agent consistently finishing off the reaction-based bot from an earlier fight.

ACTIONS->	0	1	2	3	4	5
State: 0	1,307.73	1,134.14	1,311.58	1,323.60	1,133.63	1,226.13
State: 1	1,089.43	1,434.17	1,322.25	1,288.51	1,131.06	937.3
State: 2	1,234.17	1,134.18	1,300.57	1,331.88	1,239.14	1,334.16
State: 3	1,096.59	1,433.62	1,321.00	1,247.93	1,121.92	1,013.20
State: 4	1,017.78	1,501.89	1,229.04	1,264.96	1,256.75	1,011.19
State: 5	1,071.21	1,542.58	1,284.09	1,264.60	1,130.57	1,033.98
State: 6	986.14	1,439.84	1,094.02	1,229.51	1,277.13	996.42
State: 7	981.71	1,513.92	1,309.56	1,034.03	1,227.65	996.16
State: 8	1,282.90	1,024.16	1,334.16	1,283.50	1,533.46	1,283.90
State: 9	1,254.88	1,133.54	1,318.58	1,106.14	1,350.25	999.17
State: 10	983.08	1,034.18	962.59	1,032.47	1,534.11	1,334.14
State: 11	1,293.26	1,129.59	1,527.14	1,076.91	1,433.47	1,032.27
State: 12	1,208.00	1,021.47	1,232.38	1,262.89	1,482.69	1,005.92
State: 13	1,089.81	1,129.80	1,407.29	1,108.16	1,371.69	991.66
State: 14	1,068.00	904.11	1,247.44	1,300.51	1,485.84	1,028.74
State: 15	1,056.88	1,103.06	1,481.90	1,059.63	1,428.24	781.14
State: 16	1,063.65	823.76	1,032.21	1,007.38	1,125.01	1,216.74
State: 17	1,022.77	1,005.34	1,529.76	1,010.17	1,026.49	1,033.41
State: 18	1,026.62	1,083.46	1,183.86	912.98	1,208.68	1,353.19
State: 19	1,005.48	1,015.13	1,521.51	1,011.35	1,028.49	1,014.05
State: 20	0	0	696.6	0	567.09	508.28
State: 21	1,014.94	1,526.42	1,410.13	1,117.48	1,122.99	1,017.24
State: 22	285.29	0	0	0	0	0
State: 23	1,008.07	1,506.54	1,364.04	1,065.14	1,020.13	971.98
State: 24	0	0	698.87	572.87	0	508.28
State: 25	959.65	715.22	1,299.22	981.02	996.11	899.64
State: 26	508.28	129.17	552.31	0	697.79	353.49
State: 27	801.43	903.6	1,049.14	985.08	980.51	1,013.49
State: 28	0	0	0	0	0	0
State: 29	992.58	1,015.69	1,348.07	1,102.60	1,102.04	741.74
State: 30	0	0	0	0	0	0
State: 31	989.63	947.73	747.79	973.55	783.15	670.5

Fig. 9. Third simulation results with 5000 exploration iterations, learning rate of 0.5, discount factor of 0.5, and reworked exploration phase.

For the second series of testing, we performed a total of 12 more simulations, this time recording the maximum reward values of each simulation (see Fig. 10) and the amount of eliminations in the exploitation phase where applicable. We aimed to try to find out what number of iterations would yield the maximum Q-Values before variance between the maximum values would diminish. We found that this occurred between 1000 to 2000 exploration iterations, where the maximum Q-Values appeared to reach about 1400 points and stop growing as quickly as they had from 100 to 1000 exploration iterations. However, this did not signify that the bot had learned the optimal policy yet as the learning agent only averaged around 1 win to the reaction bot's 4 at 1000 iterations, while averaging around 1 to 1 eliminations at 2000 iterations. (Below 1000 exploration iterations the learning agent did not win at all and showed very irrational behavior such as running into the enemy while low on health and reloading continuously. For this reason we are not considering simulations with exploration iterations below 1000 for the exploitation phase). This was most likely because while the maximum Q-Values had been reached, the rest of the Q-Table had not been filled out and all the states had not been fully explored. Similar to our first round of testing, where the first simulation we ran did not fill out about half the Q-Table because the learning agent did not spend any time in about half of his possible states, simulations with 1000 iterations not perform enough exploration for about half of the learning agent's possible states, and required more time to train even after our alteration to the exploration method. Furthermore, we noticed that even though we were changing the learning rate and discount factor, the spread and maximum Q-Values stayed relatively constant. This was not expected and pointed to a possible flaw in our implementation. However, changing the learning rate and discount factor, or possibly simply by re-running simulations, we were able to notice different but still rational behavior from the learning agent. For the three rounds of simulation with 2000 exploration iterations, the learning bot would display varying degrees of aggressiveness, in terms of engaging the opponent. For the first round with a low learning rate, the bot learned to engage the opponent until it had low health, then running to cover. For the simulation with a low discount rate, the bot learned to run as soon as it was under fire and run around cover nodes until it lost the reaction bot, then waiting in ambush.

V. Conclusions and Future Work

Looking at our results, it is safe to conclude that we need to rework our learning agent's state and action space. When we performed our simulations, it was clear that the agent spent most of its time in about half of the states. Also, from looking at the results for both rounds of testing, the bot did not even populate some states regardless of how many exploration iterations were used. For example, for the third simulation in the first round of testing, state 28 and state 30 were never populated, which corresponded to having low ammo, low health, and having the opponent in view. This was most likely because the agent would regenerate health or reload before encountering the opposing agent or would be between learning iterations and not register the opponent coming into view before regenerating health. This leads us to believe that we should consider implementing a static time step for updating the Q-Table instead of having it be based on a single action loop as mentioned in Fig. 6 above. This could possibly lead to a more widespread population of the Q-Table as state and reward evaluation could be done in parallel to performing actions.

We were successful in implementing Q-Learning and our implementation utilized the Unreal Engine, which is a large and robust game development engine. The action space and state space was set up to be modifiable so addressing the issues related to them should be possible without reimplementing the entire project, which was one of our goals at the onset of the project. We were also able to successfully train a Q-Learning agent which showed unpredictable but rational behavior, although we cannot comment on the consistency of its training without running more tests and simulations and addressing the current issues. Unfortunately, the agent required a lot of time to train, and would not be acceptable for a commercial video game implementation anytime soon. We did not expect to have sunk so much time into setting up the testbed in Unreal, which led to hasty testing and simulation. This is something that we intend to fix with future work, given that we will have more time and resources.

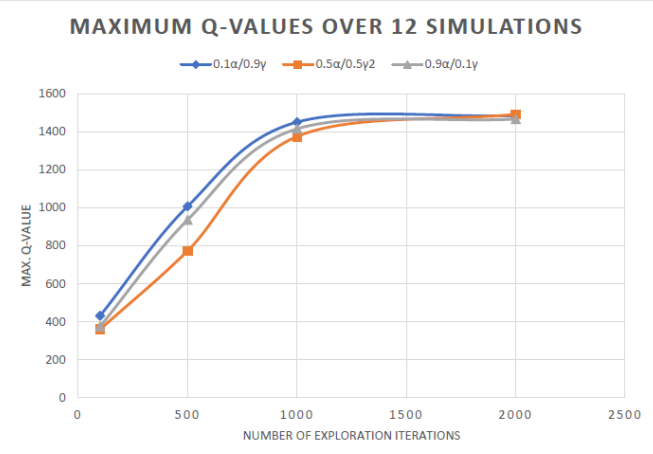

Fig. 10. Recorded maximum Q-Values for 12 simulations with three sets of learning rates and discount factors.

For future work, we intend to first and foremost implement a new action and state space for our learning agent. As it stands, the current action and state space has led to inconsistent results and many headaches in the form of bugs. We also intend to change our learning iterations to be on a timed interval and for Q-Table updates and reward calculations to be performed in parallel with action execution. Once these aspects are changed, we intend to expand our prototype and perform many more simulations while changing

and observing a wider variety of variables, such as varying learning iteration time steps and a wide variety of reward tables. Furthermore, we intend to train the learning agent against reaction-based bots with varying characteristics, as opposed to just the aggressive bot we had used for this version of the project. Training the learning agent in different environments would be beneficial as well, along with a variety of game modes and mechanics. Implementing health and ammo pickups would be a must, as this mechanic is widespread in modern video game titles and could lead to interesting behavior. Also, there is the possibility of implementing a different learning algorithm, or even a combination of learning algorithms, to see if we can combat the long training time required to attain acceptable results.

REFERENCES

[1] Kempka, Michał & Wydmuch, Marek & Runc, Grzegorz & Toczek, Jakub & Jaśkowski, Wojciech. (2016). ViZDoom: A Doom-based AI Research Platform for Visual Reinforcement Learning.

[2] McPartland, Michelle & Gallagher, Marcus. (2011). Reinforcement Learning in First Person Shooter Games. Computational Intelligence and AI in Games, IEEE Transactions on. 3. 43 - 56. 10.1109/TCIAIG.2010.2100395.

[3] Policarpo, D & Urbano, Paulo & Loureiro, T. (2010). Dynamic scripting applied to a First-Person Shooter. 1 - 6.