2017 International Conference on Computational Science and Computational Intelligence

Reinforcement Learning for All: An Implementation using Unreal Engine Blueprint

Reece A. Boyd
Computer Science Department
Middle Tennessee State University
Murfreesboro, TN, USA
E-mail: reecealanboyd@gmail.com

Salvador E. Barbosa
Computer Science Department
Middle Tennessee State University
Murfreesboro, TN, USA
E-mail: salvador.barbosa@mtsu.edu

***Abstract*—Game engines, like Unreal, Unity, and Cryengine, provide cutting edge graphics, sophisticated physics modeling, and integrated audio, greatly simplifying game development. These advanced features are often accessible through visual scripting interfaces used in rapid prototyping and by non-programmers. The goal of this research was to demonstrate that these tools can support implementation of artificial intelligence techniques, such as reinforcement learning, that have the potential to yield dynamic characters that are not pre-programmed, but rather learn their behavior via algorithms. Its novelties are the implementation of a Q-Learning bot, created in Unreal Engine's visual scripting tool, known as *Blueprint*.**

Keywords-Artificial intelligence, bot learning, visual scripting, reinforcement learning, game technologies

I. INTRODUCTION

Use of artificial intelligence (AI) by game developers is usually geared toward creating believable and compelling non-player characters (NPC) that add to immersion and enjoyment of the game. If the NPC is too simplistic, the human player quickly loses interest in the game. On the other end of the spectrum, an NPC that is invincible reduces the entertainment value of the game, and equally leads to its abandonment [1]. Some of the most successful and influential games in in recent times are F.E.A.R, Half-Life, and Halo. All of these titles have a large following of users, as they offer both single player options against AI characters, as well as multiplayer play against other humans. A common complaint of individuals that play these games in single player mode is that the NPCs are quite easy to defeat, once the player deduces their behaviors and quirks, and are seen as non-adaptable to gameplay. This has led to a much stronger preference for multiplayer modes, as players prefer to be challenged by other human players rather than by AI characters.

An analysis of the methods of behavior representation of NPC AI in video games show that many are based on scripts and dated finite state machines (FSMs) or, more recently, behavior trees (BT). Scripts usually spell out the exact behavior(s) to be executed, given a set of circumstances. FSMs are used to organize a program's (or NPC's) execution flow via a set of "states" such as *Idle*, *Attack*, or *Flee*, where only one state can be active at a time. A change in state, or transition, is triggered when predefined conditions are met. FSMs are extremely common in video games, but are often regarded as inflexible, as they can be difficult to modify and extend. BTs are trees of hierarchical nodes that determine the flow of decision making. The leaves of the tree are actions that are taken by the NPC, while the nodes along the path from that leaf to the root are nodes with conditions that had to be satisfied for the flow to reach the leaf (and result in a particular action).

While the above solution methodologies were satisfactory in the past, games have become more complex, and players have become more demanding regarding the quality of NPC behavior. This is especially true when a winning strategy requires real-time adaptability to a human player's actions [2].

The widespread availability of games engines used in AAA games (a term used by the gaming industry to refer to titles created by the largest game studios, with very large budgets and level of promotions and, usually, of the highest quality) to independent developers, academic researchers, and individuals has led to some experimentation with learning algorithms for NPCs. However, the great majority of this experimentation has been restricted to those with software development and programming skills, and leave out many who may have great ideas for creating characters, but possess limited or no software development knowledge.

Game engines are software frameworks designed to simplify the development of video games. Unreal Engine, Unity Engine, and Cryengine are examples of the most popular modern game engines. All of these engines provide simple to use interfaces to enable development of player controlled characters and NPCs. These tools are usually graphical interfaces that shield users from having to code the behavior directly in a programming language. The goal of this research project is to experiment with creating a reinforcement learning controlled NPC via Unreal Engine 4's visual interface called *Blueprint*. We chose this engine and toolset because of its popularity, and due to the fact that the software's source code is freely available as open-source at https://github.com/EpicGames/UnrealEngine. We intend to make our project available as open-source on GitHub, in order to foster its use by those who are interested, and to encourage that its functionality be extended. The remainder of this paper is arranged as follows: Section II provides background on the Unreal Engine and a review related work in literature. In Section III describes the experimental setup and methodology, and the research results and analysis are

978-1-5386-2652-8/17 $31.00 © 2017 IEEE
DOI 10.1109/CSCI.2017.136

presented in Section IV. Finally, Section V contains the conclusion and recommended avenues for future research.

II. BACKGROUND AND RELATED WORK

A. Unreal Engine and Blueprint

Unreal Engine 4 is the latest version of the popular game and graphics engine that has existed in many forms since originally launched in 1998. In March of 2015 the complete code was released to everyone, free of charge. This powerful suite of tools is a proven AAA game engine, having used to create many blockbuster games, such as *Kingdom Hearts III*, Shardbound, Unreal Tournament, *Final Fantasy VII Remake*, and the *Gears of War* franchise. In addition, its near photorealistic capabilities have been employed in the making of short films and television shows, such as the children's television series *Lazytown*.

The engine provides two distinct avenues for game development, as well as a hybrid approach: one may choose to develop the game in the C++ language or through the rapid development facility known as *Blueprint*. The *Blueprint* Visual Scripting tool provides an interface to build game functionality for player characters and NPCs via simple-to-use behavior trees that are converted into performant C++ code. It also incorporates a set of additional and more complex tools such as *AIComponents* and the *EnvironmentQuerySystem*, or EQS, for giving an agent the ability to sense and perceive its environment. The tool suite also includes game level-editing tools, and user-interface construction components and widgets.

B. Maintaining the Integrity of the Specifications

The pioneering computer scientist Arthur Samuel described machine learning as a "field of study that gives computers the ability to learn without being explicitly programmed." There are many forms of machine learning approaches. Among these are supervised learning (SL), unsupervised learning (USL), and reinforcement learning (RL). In SL, the machine is given an input (problem), and output (answer), and through the guidance of a "teacher," learns to map inputs to outputs. On the other hand, USL is only provided with an input and must determine the output, based on similarity, without any guidance. Finally, RL is inspired by behavioral psychology, with an intelligent agent learning by being rewarded for correctness in behavior, or conversely "punished" for incorrect actions [3].

There are many implementations of RL algorithms. Among the most popular are the closely related *Q-Learning* and *SARSA* algorithms. This research utilizes the *Q-Learning* algorithm [4], an off-policy method closely related to the on-policy *SARSA* technique. We restrict ourselves to describing only *Q-Learning*, since it was used for this investigation.

The *Q-Learning* algorithm is a tabular storage algorithm that maintains the values of state-action pairs. An agent utilizing this algorithm would assess its state, and choose an action available in that state for execution. The expected reward value for all state-action pairs are maintained in data tables. The algorithm updates these values as follows:

$$Q(s, a) = Q(s, a) + \alpha(r + \gamma Q(s', a') - Q(s, a) \quad (1)$$

where $Q(s,a)$ is the expected reward value for the taking action a in state s, α is the algorithm's learning rate, r is the reward (or punishment) received for taking action a in state s, γ is the discount factor, and $Q(s',a')$ is the maximum value of an action a' that is available in the resulting state s'.

The learning rate represents the extent to which the agent should replace old information with new information. If α is 0, the agent will not learn anything. If α is 1, it will completely apply the new information at each iteration of the algorithm. The discount factor represents the extent to which the agent should value future rewards. If γ is 0, the agent is said to be "myopic" in that it only considers current rewards. As γ approaches 1, the agent becomes more interested in long-term rewards.

C. Related Work

The First-Person-Shooter (FPS) is a category of games centered on the use of firearms in combat, usually from the first person perspective. It is also possible to have a third person view of the combatant in these games, and we chose this view for our research.

RL algorithms have been successfully used in many complex domains. Within the FPS genre, a multiple reinforcement learner architecture using the tabular *SARSA* algorithm proved successful in its ability to learn quickly and defeat FSM bots that are bundled with an earlier version of the Unreal Engine known as Unreal Tournament 2004 [5]. The research learned via three separate RL learners, based on whether the NPC bot was exploring, attacking, or fleeing. In contrast, our experiment uses a single learner that covers all of these states.

Inverse RL algorithms have also shown that it is possible to create more human-like AI while outperforming FSM bots in map exploration [6]. Utilizing game data collected from human players, that research focused on learning the values of rewards by exploring the action set, thus inverting the RL process. In our research we used a standard reinforcement learning technique (learn the value of actions from rewards).

In complex problems tabular implementations of RL algorithms may become unfeasible due to extremely large state spaces. In these cases, hierarchical RL algorithms that break complex learning tasks into multiple smaller subtasks can still learn quickly to solve the large and complex problems [7]. Our methodology differs in that it has a relatively small state space, and uses tabular storage for reward values.

III. EXPERIMENTAL SETUP AND METHODOLOGY

This research project used the *Blueprint* tools and was completed in two phases. In this section we introduce the capabilities of the *Blueprint* toolset employed in the research, and provide a description of each phase of the research.

A. Blueprint and Unreal AI Components

The *Blueprint* visual scripting system (hereafter *Blueprint*) was introduced by Unreal as a no-programming-required toolset to enable use of the game engine by non-

developers. It comes with a much lower learning curve than Unreal's C++ interface, and may be used alone or integrated with C++ code. *Blueprint* is popular in the video game industry as it enables fast prototyping by developers, and extends the engine's user base to non-programmers. It has visual node-based design where nodes offering different capabilities are simply connected together to compose increasingly more complex actions and environments. In this project, *Blueprint* was used to script all behaviors for both the behavior tree NPC and the RL NPC. The implementation of each of these NPCs, while representing common functionality using the Blueprint toolset, differs significantly, and a description of the architecture of each NPC type is given later in this document.

Blueprint contains many sub-tools that serve a myriad of functions. The essential ones used to construct the NPCs in this research were the *Behavior Tree*, the *Blackboard*, and the *AIController*. The *Behavior Tree* describes the actions that may be performed by NPCs. BTs are accompanied by the *Blackboard*, which act as the NPC's memory. Finally these components are tied together and managed by the *AIController* component, which established how the agents are controlled without explicit human player input.

With possible NPC actions defined, there is a need to sense and affect the environment, in order to make the appropriate action selection possible. This is accomplished via the use of *AIComponents*. This research used three AI components:

- The *AIPerception* component was used to give both NPCs visual input, enabling response via sighting the enemy bot.
- The component *PawnSensing* was used to give both agents auditory input. When noise is heard, the *Blueprint* function *OnHearNoise* is triggered, toggling the Blackboard's *EnemyHeard* flag.
- The component *PawnNoiseEmitter*, allows both agents to broadcast noise whenever their feet hit the ground. This enables response based on hearing the enemy NPC.

Finally we made use of the *EnvironmentQuerySystem* (EQS) to allow NPC's to collect data from the environment and make decisions based on this data. In this project, the *Flee* action uses the EQS to query the environment for the best current hiding spots from the enemy. The outcome is based on current enemy line of vision, and returns the closest nearby area that does not fall within this line of vision.

B. Phase I

The first phase involved building the environment for testing different implementations of NPC. The setup of this environment included creating an arena to conduct simulations, attaching weapons to each NPC, creating a laser projectile that spawns from the NPC's weapon barrel when the character chooses to fire, and many more details that enabled gameplay. The simulation environment created for this research was designed for *Deathmatch* gameplay between a BT NPC and an RL NPC. This type of contest continues until one of the combatants is dead.

The two described competitors are mirrors of one another. They both have the same abilities in terms of being able to see and hear. They also share the same three actions—one for exploring the environment, one for attacking the other NPC, and one for fleeing. The primary difference between the two bots is how each selects the action to carry out: The BT NPC is explicitly programmed using a behavior tree, to execute specific actions when certain conditions are met. The RL NPC, on the other hand, employs the *Q-Learning* algorithm to learn to operate in the environment. The first phase concluded with creation of the BT NPC using the *Blueprint* Visual Scripting toolset. The outcome of this process is the Behavior Tree created with *Blueprint*, and shown in Fig 1. The leftmost branch of the tree is the *Flee* action. The middle branch is the *Attack* action. And, finally, the *Explore* action is shown by the right branch.

C. Phase II

The second phase involved implementing, and testing the performance of the RL NPC. Unlike the BT NPC, the RL NPC has no explicitly programmed behavior. Instead, it is given a set of states, *S*, a set of actions, *A*, and obtains rewards at various points during gameplay, which it then attempts to maximize via the learning algorithm. As was previously mentioned, a tabular-implementation of the *Q-Learning* algorithm was used. Tabular *Q-Learning* is implemented via a table of values that correspond to the expected reward value for every possible state-action combination that the agent can perform. An NPC's state is composed from what it senses from its environment, including the action and location of the other bot.

The RL NPC was constructed with five Boolean flags to make up its state. These flags are *SeeEnemy*, *HearEnemy*, *TakingDamage*, *DealingDamage*, and *CriticalHealth*. These are each encoded as a string of zeros and ones ("0" if the flag is *False* and "1" if it is *True*).

As with the BT NPC, there are three actions available to the RL NPC: *Explore*, *Attack*, and *Flee*. The *Attack* and *Flee* actions are only possible if the RL NPC can see or hear the BT NPC, resulting in a table with 80 expected values for state-action pairs. At the start of the game, each of these reward values is initialized to zero.

The table data structure used by the RL NPC is a map of strings to floating-point values. The RL NPC accesses expected reward values in its table by indexing it with a string that's a concatenation of its current state and the possible action. For each of the flags that make up its state, if the flag is activated, it is represented as a "1." If it is not activated, it is represented as a "0." The order of the flags from left to right are *SeeEnemy*, *HearEnemy*, *TakingDamage*, *DealingDamage*, and *CriticalHealth*. As an example, if the agent can see the enemy, hear the enemy, is taking damage, is dealing damage, and is not at critical health, its state is represented as the string "11110" (e.g., every flag is activated except for the *CriticalHealth* flag). The agent must then decide what action it should perform based on the expected reward values in its table.

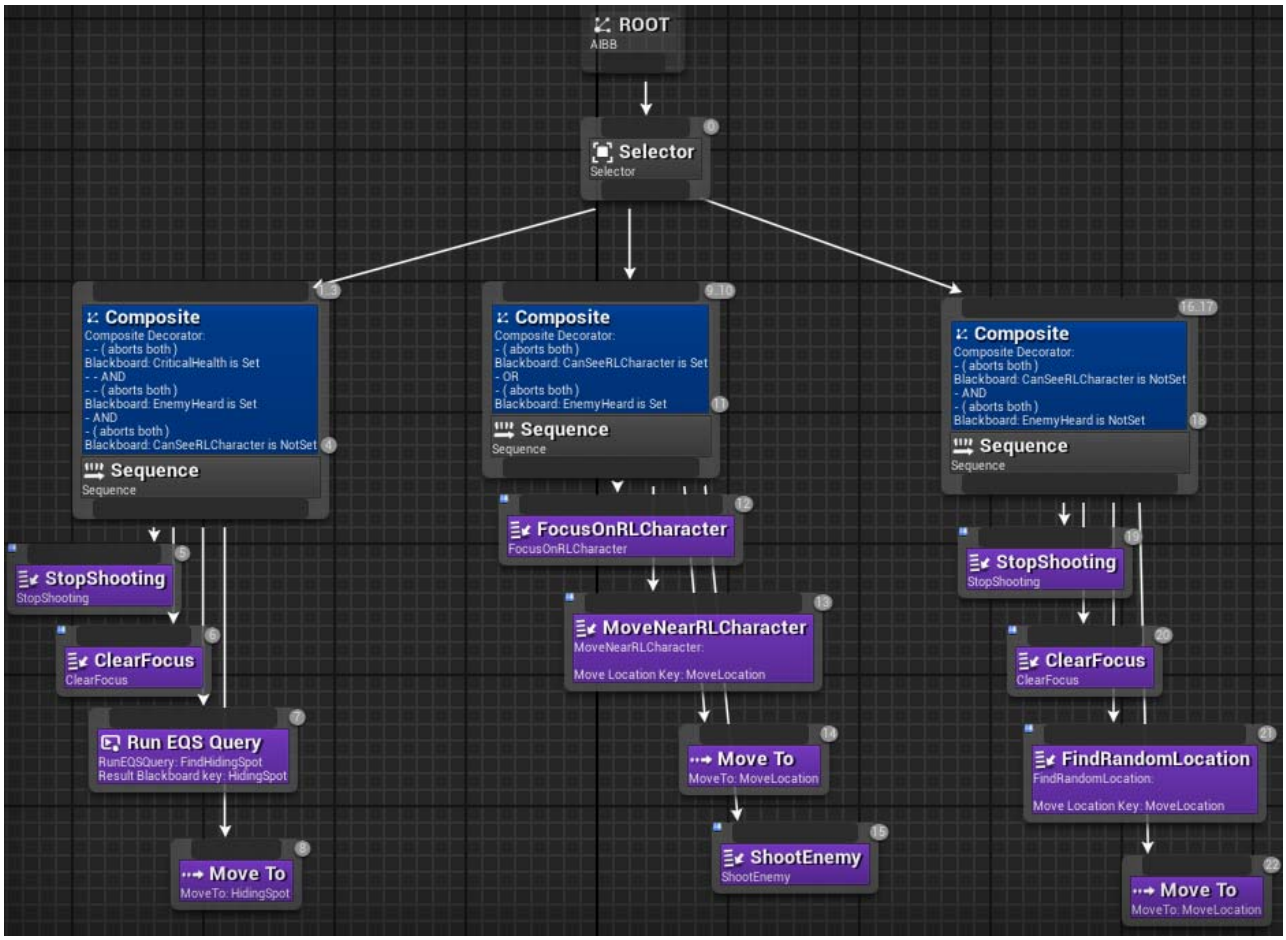

Figure 1. Behavior Tree for Non-Player Character

We represent Explore as "0", Attack as "1", and Flee as "2." The state-action strings are formed by appending one of these characters to the end of the state string. In turn, the state-action strings that the agent uses to index the table and determine the best action to choose in this case are "111100," "111101," and "111102."

The RL NPC utilizes one more variable in determining its behavior, the variable ε, or the randomness factor. This variable is used to support the NPC's action selection policy, which is known as ε-greedy and is described in the text that follows.

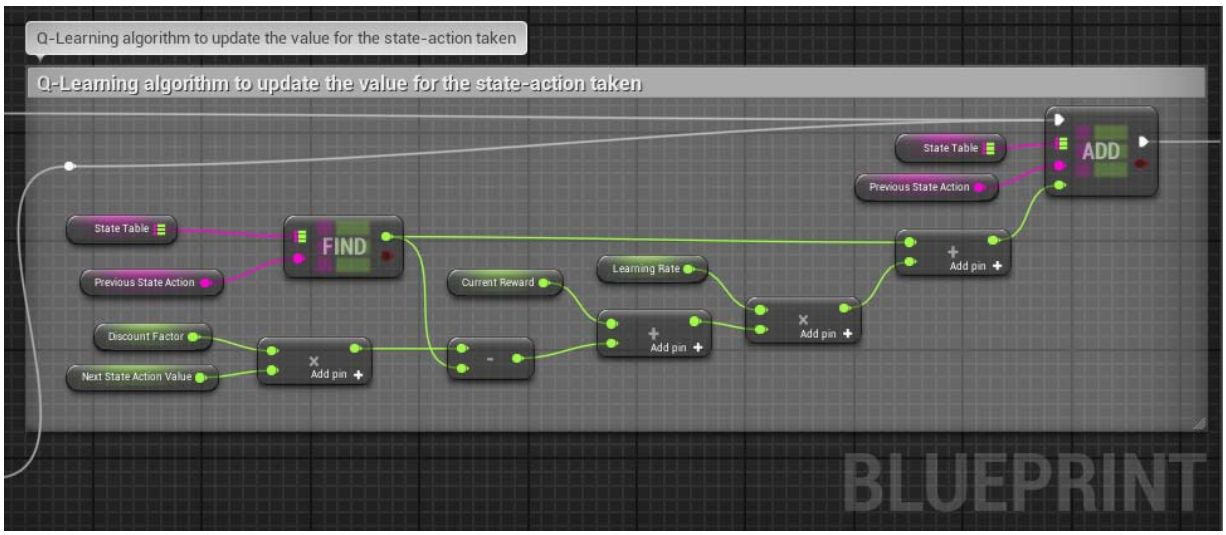

Figure 2. Reinforcement Learning Algorithm in *Blueprint*

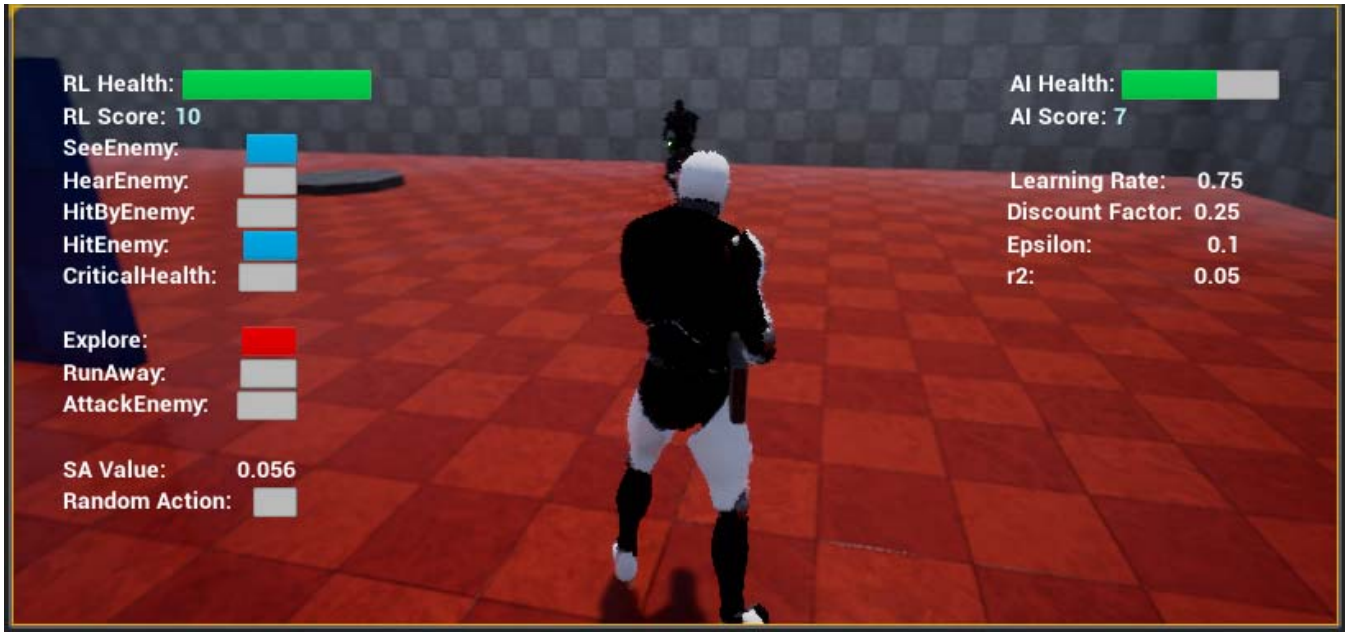

Figure 3. Screen capture during game play

The randomness factor is the probability that the RL NPC will not use its learning algorithm and table to find the next expected best action, but will instead randomly choose between available actions. This is done in order to allow the agent to discover possibly better actions that aren't immediately supported by the table entries. If ε is 0, the agent will never randomly choose an action. If ε is 1, the agent will always choose random actions and never use its learning algorithm. The *Q-Learning* algorithm implemented using *Blueprint* is shown in Fig 2.

IV. Results and Analysis

To assess the RL NPC's performance against the BT NPC we executed the *Deathmatch* simulation multiple times with varying settings for the learning rate (α), discount factor (γ), and randomness factor (ε). Each trial was composed 10 rounds, and the number of wins for each NPC type was noted. We ran 30 trials of 10 rounds for each the setting combinations. The state-action table was retained throughout the 10 rounds, allowing the RL NPC to learn across multiple rounds. At the beginning of each new trial, the state-action tables were reset to zero values. The reward values used were +1 for killing, and +0.1 for dealing non-lethal damage to the BT NPC. Conversely if the RL NPC was killed or received non-lethal damage during play, the rewards given (punishments in this case) were -1 and – 0.1 respectively.

In order to establish that learning was taking place, we set the randomness factor to 1, meaning that the RL NPC would choose a random action at every opportunity. The BT NPC was the superior performer, winning on average 9.2 of the 10 rounds in each of the 30 trials. This was expected since the RL NPC acted randomly, and usually detrimentally, throughout the matches and did not have learning ability.

We then enabled reinforcement learning by setting the value for ε at 0.1 (10% of the actions are randomly selected, and remaining actions are based on state-value pairs stored in the table). Since our goal was only to establish that reinforcement learning could be implemented using *Blueprint*, and not to optimize the performance of the RL NPC, we did only minor experimentation with values for the learning rate and discount factor. We settled on α having a value of 0.4 and gave γ the value 0.8. Fig 3 is a screen capture of a round of play. The RL NPC's status is on the left of the screen (under RL), and includes information used by the Q-Learning algorithm, such as state-action values, and selected action. The BT NPC's status is shown on the right (under AI).

Running the simulation with these values clearly showed that the RL NPC's learning was effective, as indicated by a performance increase of nearly 500%. The mean number of wins for the RL NPC was approximately four wins to the programmed BT NPC's six wins (in rounds of 10 deathmatches). This near parity performance clearly demonstrated that *Q-Learning* can be effectively implemented using a graphical visual scripting system such as *Blueprint*. Doing so opens up the use of reinforcement learning to gamers who are not software developers, increasing the likelihood that the adaptability of non-player-characters can be enhanced to the level that they behave in natural and believable ways.

V. Conclusions and Future Work

The contributions of this research are summarized in this section, and possible future work is described

A. Summary of Research Contributions

This research has described an implementation of reinforcement learning to control NPCs, using Unreal Engine's *Blueprint* visual scripting system. To our knowledge this is the first documented research that implements an RL algorithm using *Blueprint*, and this fact is the novelty of our research. This result make a manifold contribution to the body of knowledge and to the gaming community: 1) We have demonstrated that visual scripting tools can be thought of serious tools that support the implementation of real algorithms, such as *Q-Learning*; 2) Our results lowers the entry bar for those who wish to explore with RL algorithms, but are not software developers or programmers. This expansion of possible RL users and researchers is another significant research contribution; 3) Finally, releasing our experimental data and setup as an open source project further increases the likelihood that newcomers to this area of research will actually use the methodology, and build upon it.

B. Future Work

The aim of this research was to demonstrate that an RL algorithm, like *Q-Learning*, could be implemented using Unreal Engine's *Blueprint* visual scripting system, and in doing so extending the user base for experimenters of such algorithms. Our approach was to implement a very simple autonomous bot, in order to demonstrate that actual learning was taking place. The extension of both the environment and bot functionality are the primary avenues that we see for future research work. Changes in either of these two areas would likely drive modifications in the other, as explained below. Furthermore, the implementation of different learning algorithms is also a possible next step.

The environment can be extended to include a more complex arena with a greater number of walls, columns, and other structures that may be used as cover for the bots. Additionally additional items such as health pack or ammunition clips could be dispersed throughout the arena for the bots to find and use. In turn, the functionality of both bots would have to be extended to include interactions with these objects (such as search and pick up), as well as the effects of use of the items. The RL NPC would have to have a more extensive and complex action set to deal with the objects, and various additional rewards could be crafted to support those interactions.

Another direction for future work is adding different algorithm(s) for bot control, such as SARSA. Such work could be of stimulating impact, with a comparison of its results against the *Q-Learning* results obtained through this research being a particular point of interest.

REFERENCES

[1] T. Bowersock, V. Kerr, Y. Matoba, A. Warren, A. Coman. I AM AI: Interactive Actor Modeling for Introducing Artificial Intelligence: A Computer Science Capstone Project, Ohio Northern University (2015).

[2] I. Umarov and M. Mozgovoy, "Creating Believable and Effective AI Agents for Games and Simulations: Reviews and Case Study." *Contemporary Advancements in Information Technology Development in Dynamic Environments*, pp. 33-57, 2014.

[3] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*, MIT Press, Cambridge, MA, USA.

[4] C. J. Watkins and P. Dayan, "Q-Learning". *Machine Learning*, 8(3-4), pp. 279-292, 1992.

[5] F. Glavin and M. Madden. DRE-Bot: A hierarchical First Person Shooter bot using multiple Sarsa (λ) reinforcement learners. In *Computer Games (CGAMES)*, 17th International Conference on Computer Games, pp. 148-152, 2012. IEEE.

[6] B. Tastan and G. R. Sukthankar, Learning Policies for First Person Shooter Games Using Inverse Reinforcement Learning. In AIIDE, 2011.

[7] S. Mahajan,"Hierarchical reinforcement learning in complex learning problems: a survey". *International Journal of Computer Science and Engine Science and Engineering*, 2(5), pp.72-78, 2014.