Implementing Reinforcement Learning in Unreal Engine 4 with Blueprint

by
Reece A. Boyd

A thesis presented to the Honors College of Middle Tennessee State University in partial fulfillment of the requirements for graduation from the University Honors College

Spring 2017

Implementing Reinforcement Learning in Unreal Engine 4 with Blueprint

by
Reece A. Boyd

APPROVED:

_____
Dr. Salvador E. Barbosa
Project Advisor
Computer Science Department

_____
Dr. Chrisila Pettey
Computer Science Department Chair

_____
Dr. John R. Vile, Dean
University Honors College

Department of Computer Science

Middle Tennessee State University; Murfreesboro, Tennessee, USA.

I dedicate this thesis to Isabeau for always reminding me what is important in life.

Acknowledgements

I would like to thank my family for always being supportive in all my endeavors. I would also like to thank my fiancé for editing this thesis more times than she would like. I would like to thank MTSU's Computer Science Department for inspiring my love of all things Computer Science. And lastly, I would like to thank Dr. Barbosa and the Honors College for their guidance and patience in completing this thesis.

Abstract

With the availability of modern sophisticated game engines, it has never been easier to create a game for implementing and analyzing machine learning (ML) algorithms. Game engines are useful for academic research because they can produce ideal environments for rapid simulation and provide ways to implement Artificial Intelligence (AI) in Non-Player Characters (NPCs). Unreal Engine 4 (UE4) is a great choice for ML simulation as it contains many useful tools. These tools include Blueprint Visual Scripting that can be converted into performant C++ code, simple-to-use Behavior Trees (BT) for setting up traditional AI, and more complex tools such as AIComponents and the Environment Query System (EQS) for giving an agent the ability to perceive its environment. These built-in tools were used to create a simple, extensible, and open-source environment for implementing ML algorithms in hopes that it will reduce the barrier of entry for using these algorithms in academic and industry-focused research. Experimental results indicate that reinforcement learning (RL) algorithms implemented in Blueprint can lead to learning a successful policy in very short training episodes.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

The universal goal of game developers when writing AI is to create believable and compelling non-player characters (NPC) that challenge the player, while not being invincible; otherwise, the entertainment value is decreased [1]. Some of the most successful and influential games in terms of well-implemented AI are F.E.A.R, Half-Life, and Halo. Unfortunately, a very strong emphasis on multiplayer modes has slowed the pace of improvements in game AI, as players prefer to be challenged by other human players rather than by AI characters.

Many implementations of NPC AI in video games are currently based on dated finite state machines (FSMs), or behavior trees (BT). FSMs are used to organize a program's (or NPC's) execution flow. FSMs are composed of a series of NPC states such as "idle," "attack," or "flee," where only one state can be active at a time. A change in state is triggered when predefined conditions are met. FSMs are extremely common in video games, but are often regarded as outdated, as they can be difficult to modify and extend.

BTs are trees of hierarchical nodes that determine the flow of decision making for an NPC. The leaves of the tree are commands that make the NPC execute something. The nodes along the path from that leaf to the root are nodes with conditions that were satisfied for the flow to reach the leaf. While these solutions were satisfactory in the past, games have become more complex. Therefore, the importance of high-quality AI solutions has increased. This is especially true when a winning strategy requires hierarchical goal planning and real-time adjustment to a human player's actions [2].

Very few game studios have experimented with learning algorithms. In turn, this has led to stale and predictable NPCs. In the game studios' defense, there has not been a

substantial amount of research into the pursuit of making learning algorithms easier to implement in modern game engines. The goal of this research project is to begin an open-source environment for creating and testing ML agents in UE4. UE4 contains a suite of useful tools, such as Blueprint Visual Scripting that can be converted into performant C++ code, simple-to-use BTs for setting up traditional AI, and more complex tools such as AIComponents and EQS for giving an agent the ability to perceive its environment. Additionally, UE4 is much more extensible than other game engines because its complete source code is available for anyone to use and modify at https://github.com/EpicGames/UnrealEngine. UE4 is also a popular and proven AAA game engine. It has been used to create many blockbuster games, such as Kingdom Hearts III, Shardbound, Unreal Tournament, Final Fantasy VII Remake, and the Gears of War franchise. These financial successes make it an industry-leading engine where the presented research could prove more useful.

Chapter 2: Background

Machine Learning (ML) in Computer Science was defined in 1959 by pioneering computer scientist Arthur Samuel, who stated it to be a "field of study that gives computers the ability to learn without being explicitly programmed." This feat was achieved by constructing algorithms that could learn from, and make decisions based on, available data. ML is used extensively in areas where explicit algorithms are not feasible, such as implementing search engines or spam filters. These algorithms may be implemented in three classifications: 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 correct output without guidance. Lastly, RL is inspired by behavioral psychology and teaches an intelligent agent by rewarding correctness.

There are many different ML algorithms. However, the two most often used in video game research are RL and genetic algorithms (GAs), a type of USL. GAs mimic the process of natural selection and can produce a high-quality solution in any search space. Specifically, they thrive in very simple environments. This is accomplished by providing the NPC with a few goals whose achievement results in a higher fitness score. The agents with the highest fitness score at the end of a generation are then bred to produce a more suitable agent. This technique has been used very effectively in simple games, such as Super Mario World and Flappy Bird. However, in more complex environments, they require longer processing times [3].

On the other hand, multiple RL algorithms have proven successful in complex domains. In the First Person Shooter (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 come included in Unreal Tournament 2004[4]. Inverse RL algorithms have shown that it is possible to create more human-like AI while outperforming FSM bots in map exploration [5]. When tabular implementations of RL algorithms have become unfeasible due to extremely large state spaces, hierarchical RL algorithms that break complex learning tasks into multiple smaller subtasks can still learn quickly and solve very large and complex problems [6]. In addition to FPS games, RL has also proven an appropriate implementation of adaptive game AI in commercial roleplaying games [7]. For these reasons, I chose to implement the first ML NPC in this collection with RL.

Chapter 3: Methodology

This research project was completed in two phases. The first phase involved building the environment for testing different implementations of NPC AI. 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 much more. The simulation environment created for this thesis was designed for solo *Deathmatch* gameplay between a BT NPC and an RL NPC. These two competitors are mirrors of each other. 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, and one for running away. The primary difference between the two bots is how they select which action to use. The first phase concluded with the setup of the BT NPC, which is explicitly programmed to use specific actions depending on what it perceived in its environment.

The second phase involved implementing and testing the performance of the RL NPC. Unlike the BT NPC, the RL NPC is not told what to do. Instead, it is given a set of states, *S*, a set of actions, *A*, and set of rewards that it aims to maximize through its learning algorithm. For the RL NPC, a tabular-implementation of the Q-learning algorithm was used [8]. 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 agent's state is determined by what it senses from its environment. The RL NPC contains 5 flags that make up its state. These flags are *SeeEnemy, HearEnemy, TakingDamage, DealingDamage,* and *CriticalHealth.* Its 3 actions are *Explore, Attack,* and *Flee*. The *Attack* and *Flee* actions are unavailable unless

the RL NPC can see or hear the BT NPC. This makes for a table of expected values for 80 state-action pairs. The underlying data structure of the RL NPC's table 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 implementation of its current state and 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." For 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 can be represented as the string "11110." At this state, each 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. We represent *Explore* as "0", *Attack* as "1", and *Flee* as "2." The state-action strings are formed by appending one of these characters onto the state string. In turn, the state-action strings that the agent uses to determine the best action are "111100," "111101," and "111102." Initially, each reward value is set to 0. The Q-learning algorithm updates what this expected reward value should be for each state-action string in the table. The algorithm is as follows:

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

where $Q(s,a)$ is the expected reward value for the current state-action of the RL NPC, α is the learning rate of the algorithm, r is the reward attained for being in this state-action, γ is the discount factor, and $Q(s',a')$ is the expected reward for being in the next state-action.

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.

Lastly, the RL NPC contains one more variable in its behavior, the variable ε, or the randomness factor. The randomness factor is the chance that the RL NPC will not use its machine learning algorithm 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.

UE4 includes all artificial intelligence tools used to complete this project. The tools highlighted here are *Blueprint, Behavior Tree, Blackboard, AIController, AIComponents (AIPerception, Pawn Noise Emitter,* and *Pawn Sensing),* and the *Environment Query System (EQS).*

3.1 Blueprint Visual Scripting

There are two ways to code behaviors in UE4: C++ and Blueprint. Blueprint is UE4's visual scripting system. It comes with a much lower learning curve than UE4's C++ libraries, even if one is already familiar with C++. Blueprint is popular in the video game industry for fast prototyping. Its visual node-based design is functionally faster to work with and easier to debug. Blueprint classes can also be converted into native C++ classes, which optimize runtime performance. More information regarding Blueprint can be found at https://docs.unrealengine.com/latest/INT/Engine/Blueprints/. In this project,

Blueprint was used to script all behaviors for both the BT NPC and the RL NPC. This includes the implementation of the tabular Q-learning algorithm below.

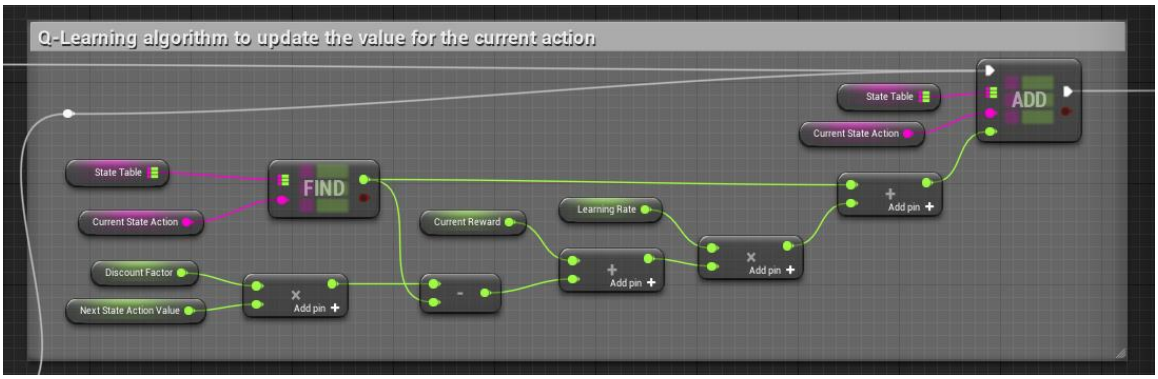

Figure 1: Q-learning algorithm implementation in Blueprint for the RL NPC.

The figure shows the Blueprint implementation of the Q-learning algorithm for the RL NPC. The variable *State Table* represents the table for the RL NPC that stores the expected reward values of each possible state-action string. The *Find* node takes as inputs this *State Table* and the current state-action string of the RL NPC and returns a floating-point value of the expected reward. This value is then subtracted from the resulting value. The *Add* node takes as inputs the *State Table*, the current-state action string, and the new floating-point value to update its existing entry in the table.

3.2 AI Tools in UE4

UE4 contains an abundance of AI tools. Some of the basic and essential ones mentioned in this research were used to implement the BT NPC. These tools included *Behavior Tree*, *Blackboard*, and *AIController*. BTs in UE4 are accompanied by Blackboards, which act as an agent's memory in the form of Blackboard Keys. BTs use this memory to make decisions about which action it should perform. More information regarding BTs and Blackboards can be found at

https://docs.unrealengine.com/latest/INT/Engine/AI/BehaviorTrees/. In this project, a BT was used to define the behavior of the BT NPC. Figure 1 below shows the decision-making processes of this agent:

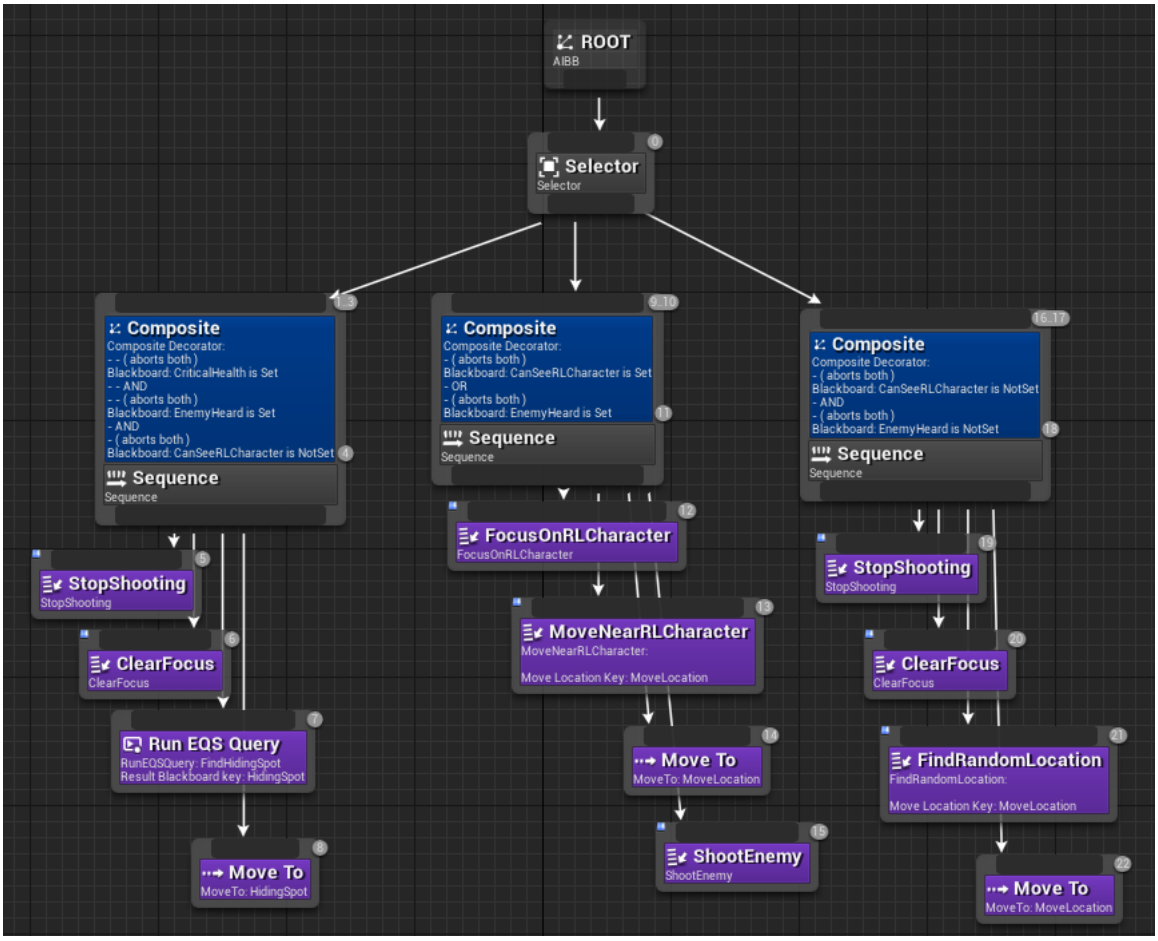

Figure 2: The Behavior Tree for the BT NPC.

The left column of steps represents the *Flee* action. This action is triggered when the Blackboard Keys for *CriticalHealth* and *EnemyHeard* is set but *CanSeeRLCharacter* is not set. The middle column of steps represents the *Attack* action. This action is triggered when the Blackboard Keys for *CanSeeRLCharacter* or *EnemyHeard* is set. The final and right-most column of steps represents the *Explore* action. This action is triggered when the Blackboard Keys for *CanSeeRLCharacter* and *EnemyHeard* is not set.

AIControllers determine how the agents are controlled without explicit player input. More information regarding AIController can be found at https://docs.unrealengine.com/latest/INT/Gameplay/Framework/Controller/AIController/. For the BT NPC, an AIController was used to assign the BT to the BT NPC since the BT could control the BT NPC from there. For the RL NPC, an AIController was used for executing all actions.

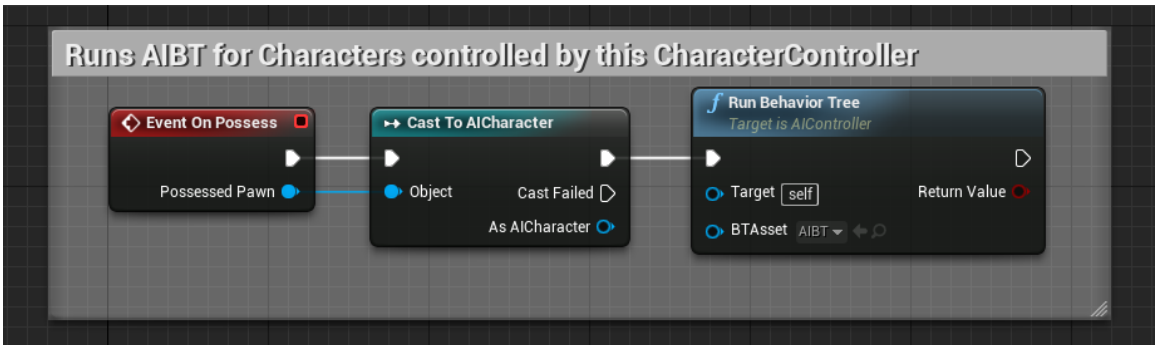

Figure 3: An AIController assigns a behavior tree to the BT NPC.

3.3 AIComponents

UE4 contains 3 tools for enabling agents to receive sensory-like data from their environment. These are *AIPerception*, *Pawn Noise Emitter*, and *Pawn Sensing*. More information regarding these three components can be found at https://docs.unrealengine.com/latest/INT/Engine/Components/AI/. In this project, AIPerception was used to give both agents visual input. Figure 3 shows the Blueprint code for how to toggle the Blackboard Key for *CanSeeRLCharacter*.

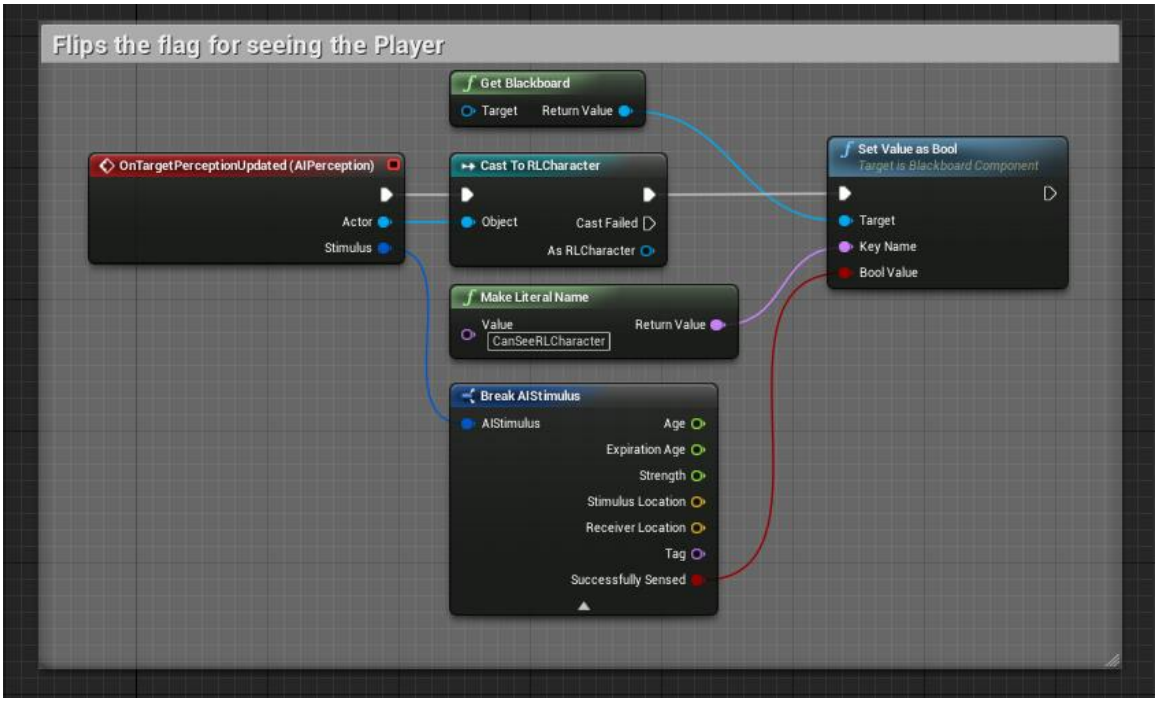

Figure 4: Blueprint code that allows the BT NPC update its blackboard with information regarding whether it can or cannot see the RL NPC.

The AIComponent *PawnSensing* was used to give both agents auditory input. When noise is heard, the Blueprint function *OnHearNoise* is called, which toggles the Blackboard Key for *EnemyHeard.*

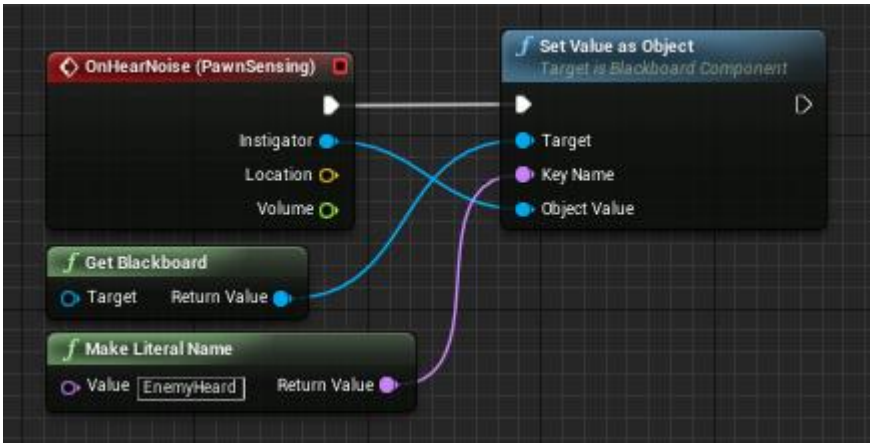

Figure 5: Blueprint code that allows BT NPC to hear ML NPC.

The final AIComponent used, called *Pawn Noise Emitter*, allows both agents to broadcast noise whenever their feet hit the ground.

Figure 6: Assigning the function PlayStepSound whenever either of ML NPC's or BT NPC's feet hit the ground.

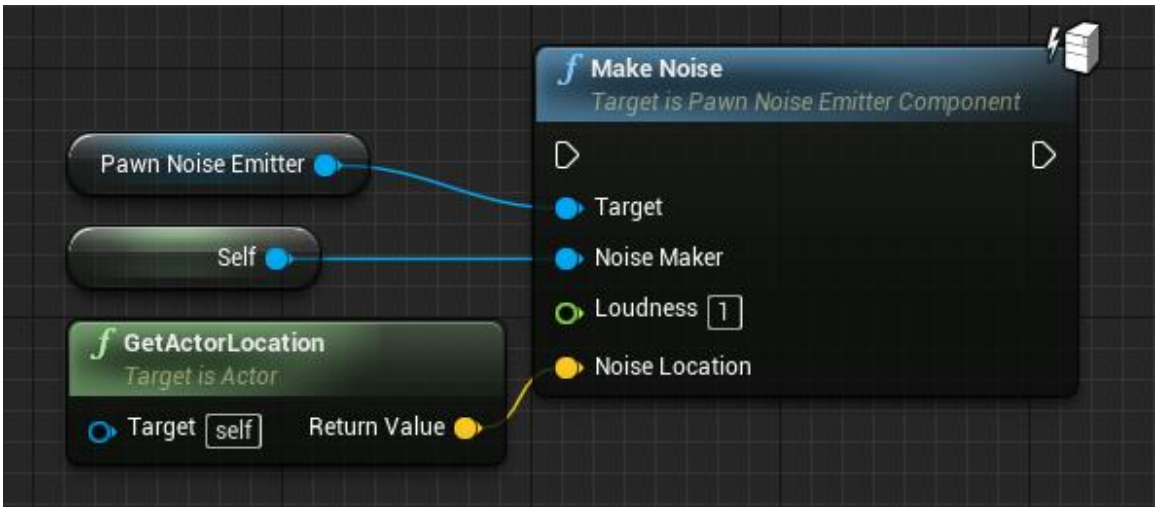

Figure 7: Blueprint code that broadcasts noise whenever PlayStepSound is triggered.

3.4 Environment Query System

UE4's EQS allows an agent to collect data from its environment and run tests on this data to determine the best result. More information regarding EQS can be found at https://docs.unrealengine.com/latest/INT/Engine/AI/EnvironmentQuerySystem/. In this project, the *Flee* action that is shared between both agents uses an EQS Query to query the environment for the best hiding spots from the enemy based on current enemy vision and closest matching area that does not fall within this line of vision. This query and the resulting figure below is from UE4's Quickstart for EQS: https://docs.unrealengine.com/latest/INT/Engine/AI/EnvironmentQuerySystem/QuickStart/12/

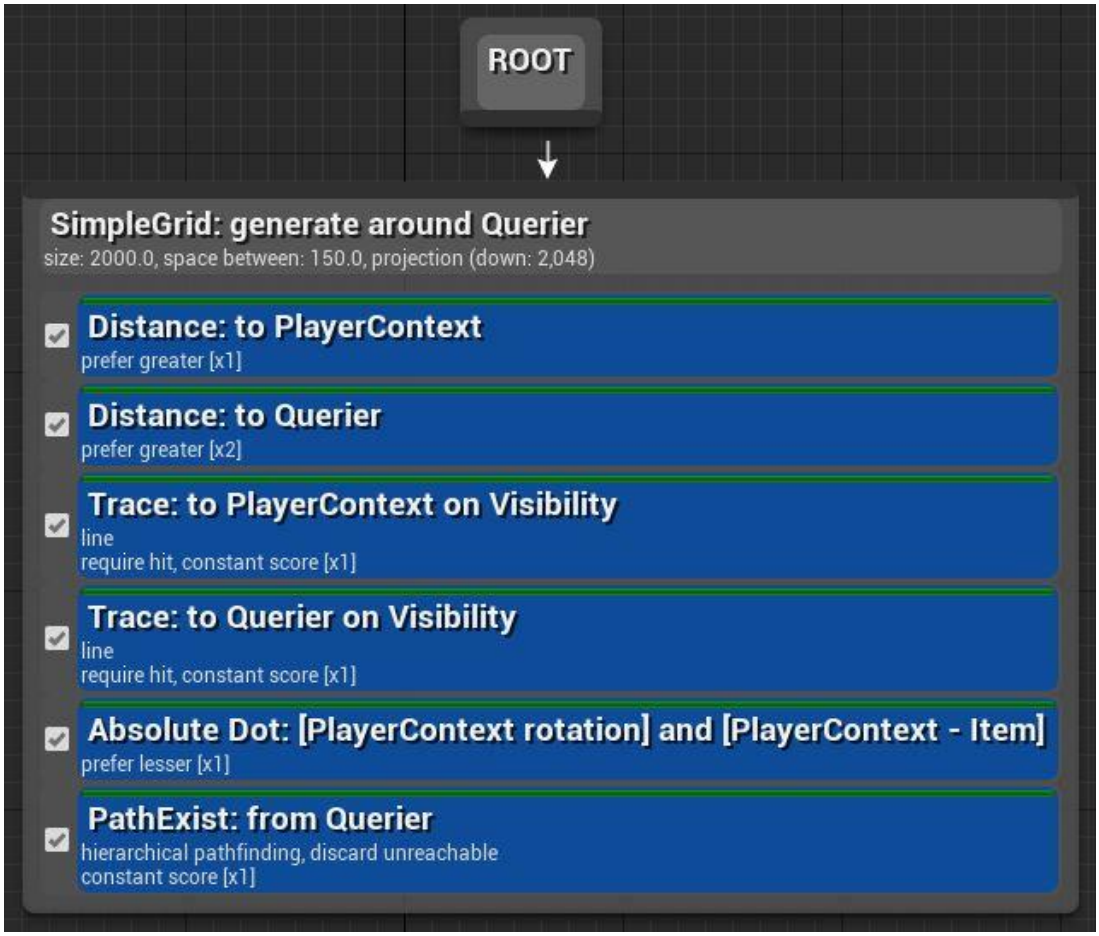

Figure 8: The FindHidingSpot Query used in this project from UE4's EQS QuickStart

Chapter 4: Experimental Results

I conducted many simulations to assess the RL NPC's ability to learn to defeat the BT NPC and the results are contained in the table below. This was accomplished by rewarding the RL NPC 1 point for killing the BT NPC and 0.1 point for damaging the BT NPC. The RL NPC was given a punishment of -1 for dying to the BT NPC and a punishment of -0.1 for being damaged by the BT NPC. A simulation ended when the combined kill score between the RL NPC and BT NPC summed to 10. Simulations were viewable from 3 cameras within UE4 and these angles are shown in the figures below.

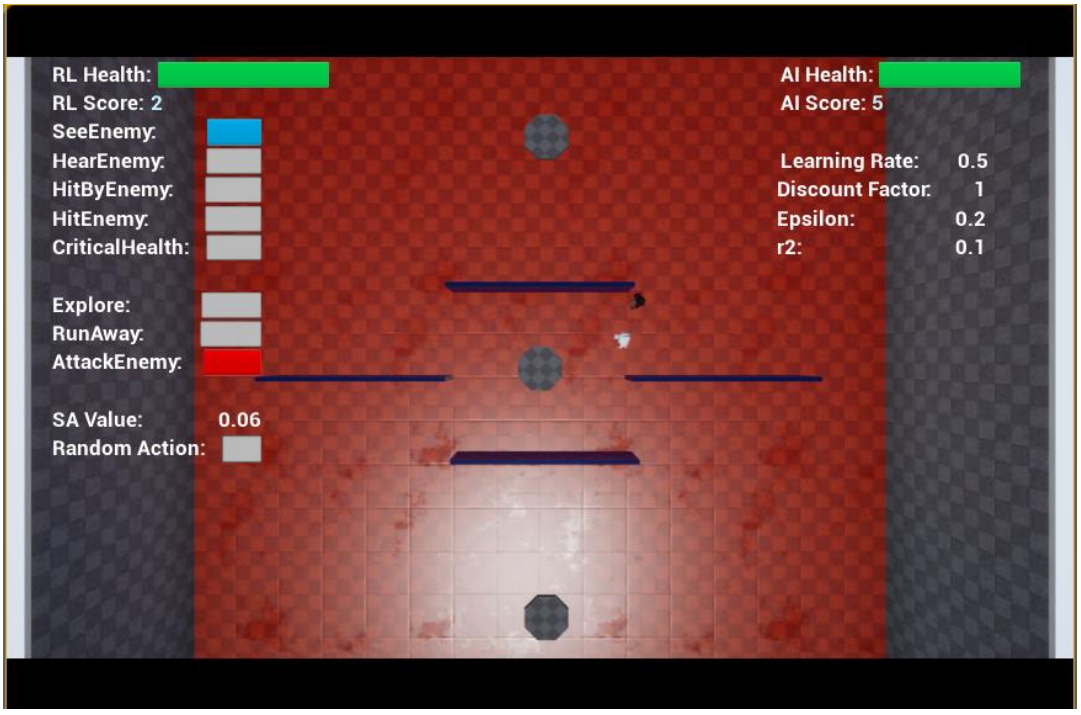

Figure 9: A top down view of the simulation environment.

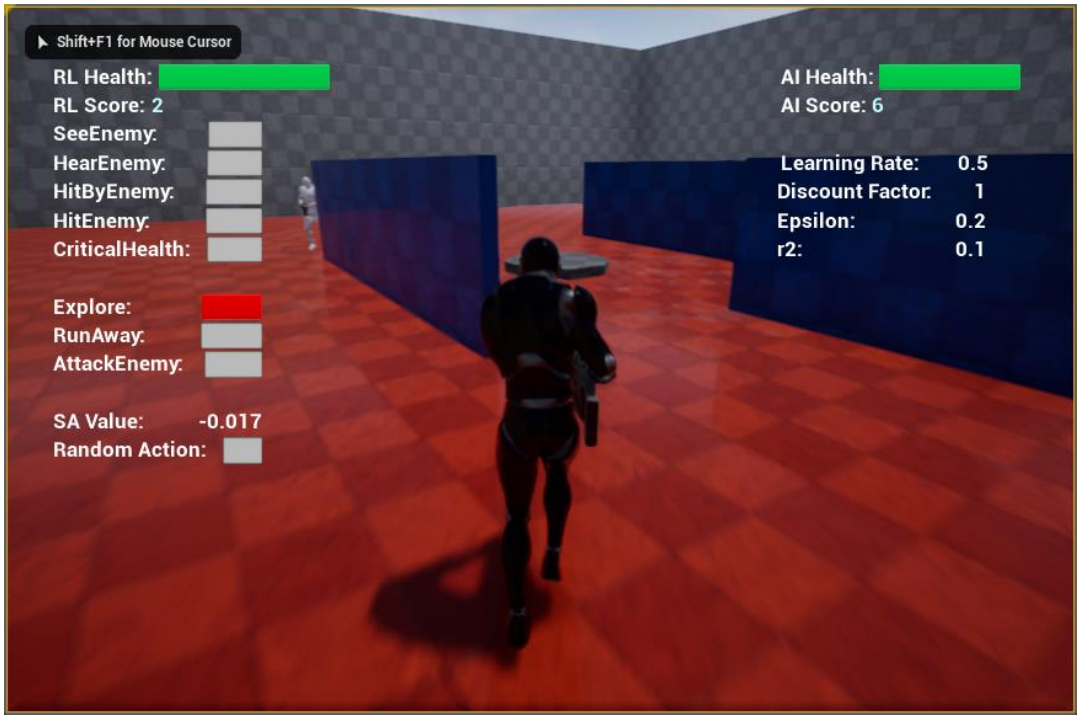

Figure 10: A view of the simulation from the perspective of the RL NPC.

Figure 11: A view of the simulation from the perspective of the BT NPC.

All 3 angles show the same diagnostic information for the simulation. The 5 flags for the RL NPC are shown directly below *RL Score,* and a blue box indicates that the flag is activated. The 3 actions are shown directly below these flags, and a red box indicates the selected action. Below the actions lie the current state-actions expected reward value and a box that will turn blue when the Q-learning algorithm was bypassed in selecting the next action and instead a random action was selected.

The table below shows the average kill scores of the RL NPC and the BT NPC over 10 simulations for different learning rate values.

Table 1: Performance of the NPCs with varying Learning Rates.

RL	BT	Learning Rate	Discount Factor	Randomness Factor
2.9	7.1	1	1	0.1
3.2	6.8	0.6	1	0.1
3.4	6.6	0.5	1	0.1
3.1	6.9	0.4	1	0.1

The table below shows the average kill scores of the RL NPC and the BT NPC over 10 simulations for different discount factor values.

Table 2: Performance of the NPCs with varying Discount Factors.

RL	BT	Learning Rate	Discount Factor	Randomness Factor
2.8	7.2	1	0	0.1
1.9	8.1	1	0.33	0.1
2.3	7.7	1	0.66	0.1
2.9	7.1	1	1	0.1

The table below shows the average kill scores of the RL NPC and the BT NPC over 10 simulations for 4 pairs of parameters. In each pair, the only difference between the sets of parameters are that one has a randomness factor value of 0.1 (meaning a random

action will be chosen every 1 out of every 10 actions) and the other has a randomness factor of 0.2 (1 out of every 5).

Table 3: Performance of the NPCs with varying Randomness Factors.

RL	BT	Learning Rate	Discount Factor	Randomness Factor
2	8	0.6	0.9	0.2
2.9	7.1	0.6	0.9	0.1
3.2	6.8	0.4	1	0.2
3.1	6.9	0.4	1	0.1
3.3	6.7	0.6	1	0.2
3.2	6.8	0.6	1	0.1
2.7	7.3	1	1	0.2
2.9	7.1	1	1	0.1

The table below compares the best performing variant of the RL NPC with a variant of the RL NPC where learning was disabled. Average kill scores of the RL NPC and the BT NPC were recorded over 30 simulations for the two variants.

Table 4: Performance of the best RL NPC vs the RL NPC with learning disabled.

RL	BT	Learning Rate	Discount Factor	Randomness Factor
3.83333333	6.16666667	0.4	0.8	0.1
0.8	9.2	1	1	1

Our experimental analysis shows that the performance of the RL NPC was significantly better with its learning algorithm enabled than without. While we could not find a variant of the RL NPC that could defeat the BT NPC, the best variant of the RL NPC that we found and tested performed at 63% of the performance of the BT NPC without being explicitly programmed. This is good given that the learning episodes were very short—reaching a combined kill score of 10 gives very few opportunities to receive large positive or negative rewards—and the state-action space was very small with the table being composed of only 80 state-action pairs. Additionally, with such a small state-

action space to work with, it was easy to implement the BT NPC with perhaps the most optimal policy for winning.

Our results also indicate that the algorithm was not significantly affected by different values of learning rate, discount factor, or randomness factor. Once again, this could be due to the small state-action space and simplistic level design. In a more complex environment with a larger state-action space, one could expect these variables to have a larger impact on performance.

Chapter 6: Conclusion

This thesis has described an implementation for applying ML to NPCs using UE4's AI tools and Blueprint visual scripting system. While the RL NPC did not outperform the BT NPC as the environment and state-action space were too simple, the experiments show that the RL NPC is capable of learning very quickly. This project provides an environment that makes RL algorithms easier to implement in academic research. To our knowledge, this is the first Blueprint implementation of an RL algorithm. With Blueprint's gentle learning curve, this will help lower the barrier of entry for testing ML algorithms in academia, industry, and the gaming community.

6.1 Future Work

While this paper successfully presented an environment for quickly implementing and testing ML algorithms in UE4 with Blueprint (C++ implementations can also be added to the project), there is a lot of opportunity for future work. Creating a more complex environment for this RL NPC, implementing an ML NPC with a different RL (or other kind of ML) algorithm, or conducting simulations against a human player, are all avenues that can be explored next given the groundwork laid by this thesis. This work is entirely viewable and modifiable under the GNU General Public License at https://github.com/reecealanboyd/RL4NPCs.

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, Ohio

[2] Umarov, I. and Mozgovoy, M., 2014. 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.

[3] McPartland, M. and Gallagher, M., 2012, September. Interactively training first person shooter bots. In Computational Intelligence and Games (CIG), 2012 IEEE Conference on (pp. 132-138). IEEE.2015.

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

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

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

[7] Spronck, P., Ponsen, M., Sprinkhuizen-Kuyper, I. and Postma, E., 2006. Adaptive game AI with dynamic scripting. Machine Learning, 63(3), pp.217-248.

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

Appendix

ML – Machine Learning

AI – Artificial Intelligence

NPC - Non-Player Character

UE4 – Unreal Engine 4

BT – Behavior Tree

EQS – Environment Query System

RL – Reinforcement Learning

FSM – Finite State Machine

SL – Supervised Learning

USL – Unsupervised Learning

GA – Genetic Algorithm

FPS – First Person Shooter