Machine Learning and Player Engagement: Integrating Deep Neural Networks into Video Game Mechanics

K. Muller
School of Electrical Engineering and Computer Science
University of North Dakota
Grand Forks, United States
karl.muller@ndus.edu

A. Holmes
School of Electrical Engineering and Computer Science
University of North Dakota
Grand Forks, United States
alexander.holmes@ndus.edu

***Abstract*—One of the struggles of modern game development lies in finding different ways to keep their players engaged. Instead of designing new game mechanics that could fail or not resonate with the players, a different approach can focus on designing the game to learn the player. This paper proposes increased integration of true enemy AI through Deep Neural Networks (DNNs) and Reinforcement Learning (RL). To demonstrate this concept, a classic action mech runner game was designed with a tutorial (As shown in Fig. 1), first level, boss sequence, each containing various enemies and environmental hazards. Each different enemy type received their own fully trained AI model implemented from Unity's ML Agents package and a local Pytorch training environment. We concluded that the inclusion of these models did enhance player engagement as each enemy would dynamically adjust their behavior to the skill level which a player is exhibiting, optimizing the gameplay experience without the need for a hardcoded dynamic difficulty adjustment system.**

Keywords—Artificial Intelligence, Machine Learning, Deep Neural Network, Reinforcement Learning

I. Introduction

Video games often advertise the integration of artificial intelligence, but many of these AI systems rely heavily on pre-programmed behaviors rather than leveraging the full potential of machine learning for dynamic decision-making. These algorithms typically guide the game's characters or entities through scripted actions, limiting the emergence of truly adaptive behaviors driven by machine learning algorithms. In taking advantage of the full capabilities of Artificial Intelligence, we can create an enhanced player experience catered dynamically to their gameplay. These fundamental items led as inspiration in the development of K7 Mech Runner, a game we sought to utilize in an exploration of the ways in which Artificial intelligence can transform the experiences of video game players.

II. Game Mechanics

In designing a side scrolling mech runner game we wanted to nurture players' complete immersion into gameplay. To accomplish this in K7 Mech Runner, we took on the project in a 3D space, adding another field of depth for players. This also broadened our horizons in the design of the levels allowing us to break from the traditional linear playstyle that the players of the genre are usually condemned to.

A. Mech Runner

The term 'Mech Runner' is geared to embody the games Mech-themed take on the classic side-scrolling genre. In working with a game of this style, pacing of the player, enemies, animations, and scene changes are super important to keep players in a mindset of constant reaction and stimulation.

Fig. 1: K7 Mech Runner Tutorial Level

B. User Experience and Engagement

User experience was the main concern of our team in K7 Mech Runner. Going against the grain of the classic roots of this genre, we wanted to explore the ways in which we can create an engaging play-environment using machine learning for deep neural network models from which enemies would base their pacing and decisions from, providing each player with a game experience uniquely tethered to their playstyle.

C. Player Design

Design of the player character for K7 Mech Runner was driven by a desire to put players at the wheel of their mech. This involves a streamlined mobile control interface, physics that meld a realistic and fast-paced feel, and animations that bring the game to life. Giving the player the opportunity to experience as real and personal an experience as possible with their surroundings allows the opportunity for our enemies to shine.

D. Level Design

K7 Mech Runner consists of three different main level components. These components consist of the tutorial level, level one (consisting of three different stages), and a boss fight sequence. In designing the levels, we sought to add an additional layer to the game challenging the player with the limits of their usage of controls and game mechanics in order to avoid enemies, and make leaps through crossfire, creating

an additional dimension to the challenges that the game provides.

III. Machine Learning Training Environment

A. Software

To create K7 Mech Runner we worked namely in the Unity game engine. The Unity game engine allowed us to script game sequences and mechanics through Visual Studio IDE in C# programming language, create the scenes and animations using game artistic models, and set up the scenes in which our reinforcement learning behaviors would be trained. Any artistic models that were not provided for us were created using blender, an open source 3d modeling software, then provided textures through text to image prompt in Meshy, an online machine learning API that allows you to generate text-to-image textures for artistic models using Artificial Intelligence.

The Machine Learning models for the game were developed using the ML Agents package, which allowed us to interface through Unity with a local Pytorch environment. This allowed us to utilize our training scenes to generate simulated game scenarios from which our models would use observations to make varying decisions from the parameters and subsequent rewards that we would provide. These simulations would be run over 500 thousand to 2 million episodes in hopes to reach stabilization in model behavior and a favorable player-enemy interaction. Each model takes in inputs sized through a *Vector Observation Space Size*, which pertains to the total number of parameters we want the model to have. Similarly, ML Agents denotes the output of a model through *Actions* which can be *Continuous* or *Discrete.* The models in our game only use *Discrete Branches*. The number of *Discrete Branches* coincides with the number of outputs for that model. Each discrete branch can also have its own size to specify how many possible discrete values that branch of output can have. All these come together in ML Agents to form the *Agent* script that allows you to train your agents through Pytorch, as shown running in Fig. 2.

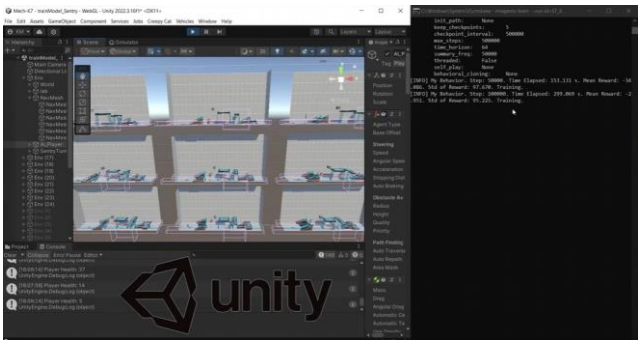

Fig. 2: Running of training simulations via Unity and Pytorch

B. Hardware

In running such a massive expanse of iterations, simulations take a massive hardware toll. To handle this, we run tests on a more robust machine consisting of: Graphics Unit- NVIDIA GEFORCE RTX 3070 TI 8GB GDDR6X, Processor- OEM INTEL CORE PROCESSOR I9-12900KF 8P/16 + 8E 3.20GH, Memory- ADATA 8GB DDR4-3200 XPG Z1, Storage- SAMSUNG 990 PRO Series - 2TB PCIe Gen4. X4 NVME 2.0c - M.2 Internal SSD

IV. Sentry Turret

The first of the enemies we worked on was the Sentry Turret. The Sentry Turret acts as a stationary enemy that shoots at the player forcing them to duck and dodge its fire. This model acts as the baseline of all others in what shooting style, accuracy and cadence different environmental factors call for. The exact reward weights and values for this model, however, are specific to the needs of this being the sole stationary enemy of the game and results in a behavior reflective of such.

Figure 3: Sentry Turret Enemy

A. Movement and Mechanics

The Sentry Turret is a stationary gameobject capable of three different behaviors, 1) *rotating towards player*, 2) *machine gun attack*, and 3) *laser beam attack*. The Sentry Turret always has access to the player's location and will aim appropriately towards it when within its vision range. The machine gun attack consists of an intermittent 5 bullet pulse, with a 0.5 second pause between each pulse where each bullet that collides with the player deals 2 damage. The laser beam attack consists of a two-pulse larger bullet from the turret that must wait 2 seconds to charge up before being sent towards the player. Each laser beam that hits the player deals 6 damage. The mechanics of the Sentry Turret focus on learning what attack pattern to utilize based on how much health the Player currently has left. In general, when the Player has higher health, the Sentry Turret should be focusing on using its higher damage *laser beam attack*. As the Player's health gets lower, it will switch to its lower powered *machine gun attack*. If the Sentry Turret itself gets close to dying, it will switch back to its *laser beam attack* in attempt to deal some damage before perishing. This attack pattern mechanic is what we were looking for the Sentry Turret model to learn through training.

B. Observations & Decisions

Unity's ML Agents package allows the Sentry Turret to interact with, and adapt to, its environment through observing its surroundings and making appropriate decisions. The observations, or inputs, for the Sentry Turret model are *playerPosition* (Vector3), *turretPosition* (Vector3), *playerHealth* (float), and *turretHealth* (float). The total *Vector Observation Space Size* for the ML Agents behavior parameters is 8. These observations will be obtained by feeding in the appropriate value from the Player and Sentry Turret in real time. The model consists of 3 discrete branches on which to make decisions with each branch size only having a size of 2. These outputs are *weaponChoice*, *whenToFire*, and *accuracy*. A value of 0 for *weaponChoice* results in the *machine gun attack*, and a value of 1 equips the *laser beam attack*. A value of 0 for *whenToFire* tells the Sentry Turret not to fire, and a value of 1 triggers the Sentry Turret to fire. A value of 0 for accuracy keeps the turret accurate, while a value of 1 introduces some inaccuracy into the Sentry Turret.

C. Reward Mechanisms

ML Agents integrates Reinforcement Learning to allow for the Sentry Turret Agent to receive positive or negative rewards dependent on the model exhibiting our desired behavior. The reward mechanism was designed with the thought of keeping the Player engaged with each Sentry Turret it encountered. We determined the best way to control this was rewarding certain attack patterns dependent on the player's health. If the Player's health is high, it will exhibit different tactics to when the Player is low, as well as if the Sentry Turret itself is low. This allows the Sentry Turret to display a unique behavior pattern each interaction the Player has with it, giving the player a new experience and keeping them engaged. As such, the reward mechanism is as follows: each bullet the turret hits the player with is +4 reward, if it hits with the specified attack in the correct Player's health range, it gets +1.5 more reward. If it misses the Player, -0.2 reward. If the turret dies, it gets –0.1 reward, but if it kills the Player, it gains +1 reward. The reason behind this reward system is that the Sentry Turret is a very basic enemy so most of the time it will die to the Player. We don't exactly want to reward the Sentry Turret for killing the Player either since the game is most engaging and fun while the Player is in combat, and when their health is close to low, but not dead. As such, these rewards allow us to steer the model in the direction of keeping the Player within a certain health range by promoting certain attack patterns, without attempting to always kill the Player. This is the reward mechanism we started training each model with, but this system did vary slightly back and forth as we reached our final model.

D. Testing and Training Models

In order to begin training multiple models with the parameters and reward system, we first had to test to ensure basic mechanics were working. During development of the Sentry Turret's basic behaviors in Unity, there were several bugs that we encountered at first, mainly with how the bullet and laser mechanics interacted with the Player, as well as ensuring the rotation of the Sentry Turret was locked towards looking at the Player on the correct axis. These bugs were found relatively quickly and corrected.

With the correct behavior implemented, a training environment was created in Unity for the Sentry Turret models. This training environment was developed as a Unity scene that would best allow the Sentry Turret to get a feel for the Player at any possible position, direction, and health. Each episode randomized each of these values to serve as the training data and inputs for each episode of the model. How the model reacted and performed to each new episode was evaluated through its reward system. Therefore, each new episode enabled the model to learn from the last episode. The ML Agents package in Unity greatly helped streamline this process, speeding up the time it took to go through each episode. This training scene in Fig. 4 was copied over nine times to allow the model to experience nine different environments at once, further speeding up training. The local Pytorch environment we had installed enabled the Deep Neural Network for the model to use Reinforcement Learning to train each model. The hyperparameters and structure of the Neural Network were detailed in a configuration YAML file for each model shown in Table 1.

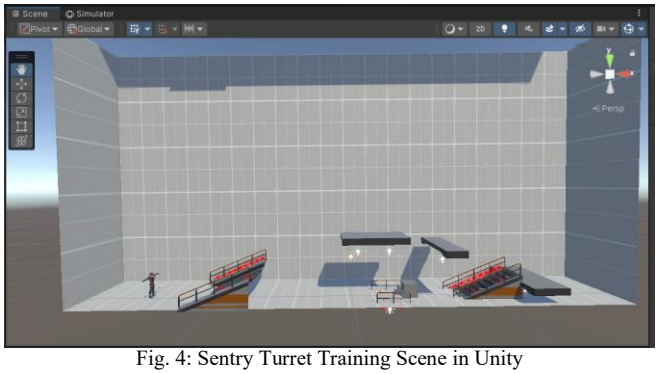

Fig. 4: Sentry Turret Training Scene in Unity

TABLE 1. REINFORCEMENT LEARNING TRAINING PARAMETERS IN YAML FILE

Hyperparameters	
batch size	1024
buffer size	10240
learning rate	0.0003
beta	0.005
epsilon	0.2
lambd	0.95
num epoch	3
learning rate scheduler	linear
beta scheduler	linear
epsilon scheduler	linear
Network Settings	
normalize	false
hidden unit	128
num layers	2
vis encode type	simple
Reward Signal - Extrinsic	
gamma	0.99
strength	1.0

E. Model Analysis

We trained a total of seven different models for the Sentry Turret. Each model had certain tweaks and changes to the reward system in order to reach the desired behavior and stabilization of the model. As shown in Fig. 5**,** the cumulative reward for each model increased with each new model, starting from very low and negative rewards to more and more positive over time. Each model here went through 500K steps. The model we ended up choosing to implement into the game was ST_7, as it gave us the desired behavior we were looking for, while being the most stabilized and rewarded model. ST_7 exhibited visible dynamic tendencies to towards the player. It was skilled at choosing the right attack depending on the Player's health, as well as being slightly more inaccurate when it detected the Player was consistently on low health. The Sentry Turret is a basic enemy, so analyzing its model was relatively quick and produced consistent and expected results. Being ingrained into K7 Mech Runner, a Player running through the level will now

always be going up against a Sentry Turret that isn't just static and predictable, but adaptable to their current playthrough.

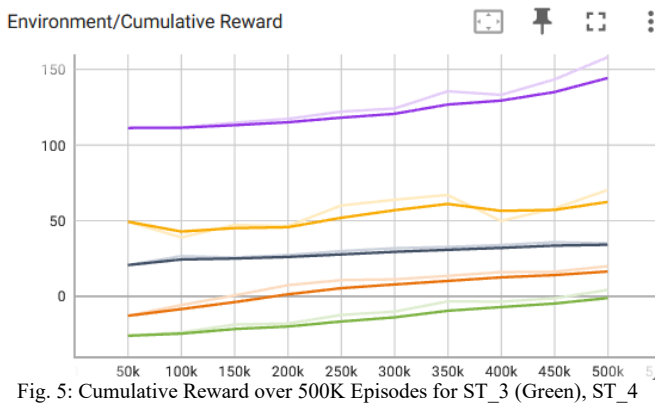

Fig. 5: Cumulative Reward over 500K Episodes for ST_3 (Green), ST_4 (Orange), ST_5, (Gray), ST_6 (Yellow), and ST_7 (Purple)

V. Drone

Building off the foundation laid by the Sentry Turret, we have the drone enemy. The drone enemy utilizes similar shooting mechanics to the Sentry Turret; however, the drone enemy moves back and forth tethered to a predetermined path in the air. This forces players to pay attention to the skies in order to avoid an onslaught of bullets from above. Another unique component to this enemy is its small stature. Not only is the drone more difficult to hit because it's moving back and forth, but also because it is smaller than any other enemy in the game. This makes trying to avoid and find cover from drones entirely a very reasonable course of action. The model created for this enemy accounts for the Drone's added level of difficulty through its reward system while still adapting to the Player's skill level.

Fig. 6: Drone Enemy

A. Movement and Mechanics

The Drone enemy is a flying gameobject on a set movement pattern capable of three different behaviors, 1) *rotating towards player*, 2) *machine gun attack*, and 3) *dashing*. Similar to the Sentry Turret, the Drone always has access to the Player's location and will aim appropriately towards it when within its vision range. The *machine gun attack* is also the same as the Sentry Turret's *machine gun attack*. The *dashing* behavior follows its movement pattern where the Drone flies back and forth horizontally, not changing its position on the y-axis on a set starting and ending x-axis bound. When the Drone gets hit by a bullet from the Player, it dashes away in the opposite direction for a distance that varies depending on the Player's and Drone's health. The distance for which the drone decides to dash is interpreted from its behavioral machine learning model. The guidelines for the *dashing* mechanic includes dashing in the opposite direction of Player movement if it is hit, dashing the least distance when the Player has lower health, and dashing the most distance when the Player has higher health.

B. Observations & Decisions

Using ML Agents, the Drone enemy can also interact appropriately and respond to its environment. The starting observations for the Drone model were *playerPosition* (Vector3), *dronePosition* (Vector3), *playerHealth* (float), and *droneHealth* (float). In future iterations, *playerPosition* and *dronePosition* were determined to add unnecessary noise to the model, restricting the input of the model to *playerHealth* and *droneHealth*. The total *Vector Observation Space Size* for the ML Agents behavior parameters started at 8 and was later reduced to 2. These observations will be obtained by feeding in the appropriate value from the Player and Drone in real time. The model will consist of 2 *Discrete Branches* on which to make decisions with the first branch having a size of 4, and the second branch only having a size of 2. These outputs will be *dashDistance* and *dashDirection*. The *dashDistance* output covers range *{1, 2, 3, 4}* where each value serves as a multiplier to how much distance the Drone should dash. The *dashDirection* will output a value of 0 for dashing left, and a 1 for dashing right.

C. Reward Mechanisms

The reward system for the Drone agent promotes its *dashing* mechanic to attempt to avoid the Player when they have high health and to dash less when the Player is low. Since the attack of the Drone is static, the dynamic dashing distance allows the Drone to gauge its difficulty based on how the Player is progressing. If the Player is struggling and close to dying, we don't want the Drone to automatically make it a harder challenge for the Player. Again, the aim of enhancing Player engagement lies in our reward mechanism which we determined to be keeping the Player low, but not necessarily always killing them. However, the Drone agent will change its tactic to dash more once it senses its own health is low. As such, the reward mechanism is as follows: each bullet that hits the player is +4 reward, but if it misses it is –0.2 reward. If the model correctly chooses max dashDistance when Player's health is above 80%, it gets +0.5 reward. If the model correctly chooses minimum dashDistance when Player's health is below 20%, +0.5 reward. If the Drone dies, it gets –0.2 reward but if it kills the player, it gains +1 reward. Here again we don't want the agent to necessarily kill the Player, but to keep their health low enough to where they know they must pay attention to the game or they will die, thus keeping them engaged. The reward mechanism here varied slightly with each model we trained as well.

D. Testing and Training Models

The first iteration of the Drone presented some errors that were overlooked in development. Even though the Drone seemed to just be a translation of the Sentry Turret to be in the air and moving on a bounded y-axis, the movement physics and the dashing mechanic caused issues. The dashing mechanic was incorrectly multiplying the *dashDistance* and jumping far off screen. Any rotations in the environment during gameplay were not rotating the Drone as well, causing it to move onto another axis off screen. Once these base behaviors were fixed, we were ready to start training the model correctly.

The training environment for the Drone was similar to the Sentry Turret, except there were more areas for the Player AI to hide under cover or maneuver away from the Drone as shown in Fig. 7. Again, this varying environment along with randomization of variables at the start of each episode proved to be efficient at acquiring training data and rewards for the model. The same approach was used where the training scene

in Unity was copied nine times to speed up training times. The training parameter used by the Sentry Turret in Table 1 worked well for the Drone as well, except there was one model trained for 2M steps alongside the other models being trained for 500K episodes.

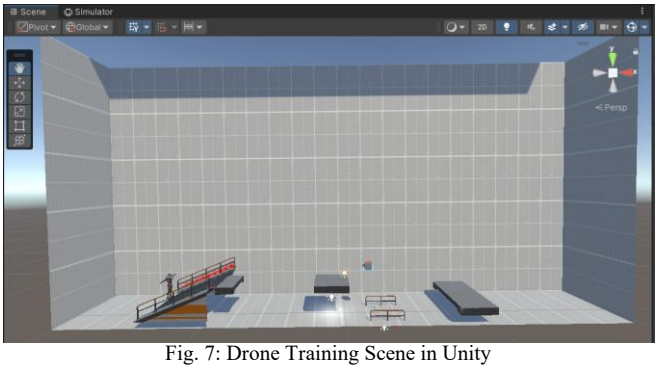

Fig. 7: Drone Training Scene in Unity

E. Model Analysis

After experiencing the training of the Sentry Turret and preemptively checking our reward system beforehand, we were able to train the Drone in less models, especially since it also had a relatively simple network with some of the models only have two inputs and two outputs. Five total models were trained, D_1, D_2, D_3, D_4, and D_5. Discussed a little bit in the *Reward Mechanism*, by the D_2, we realized having an input of *playerPosition* and *dronePosition* was unneeded noise to the model since the reward system did not care about the position of the Drone or the Player, nor did we want where the Drone or Player was located to effect the Drone's behavior. The reward system was also slightly configured to reward for a *dashDistance* of 2 or 3 when the Player's health was between 20% and 80%. As such, subsequent models D_3, D_4, and D_5 were trained without the noisy parameters, which resulted in a much greater overall performance for these models as shown in Fig 8. The Drone model D_5 was chosen for its stability and overall desired performance during gameplay. This model dashed dynamically as expected when encountered with varying levels of the Player's health.

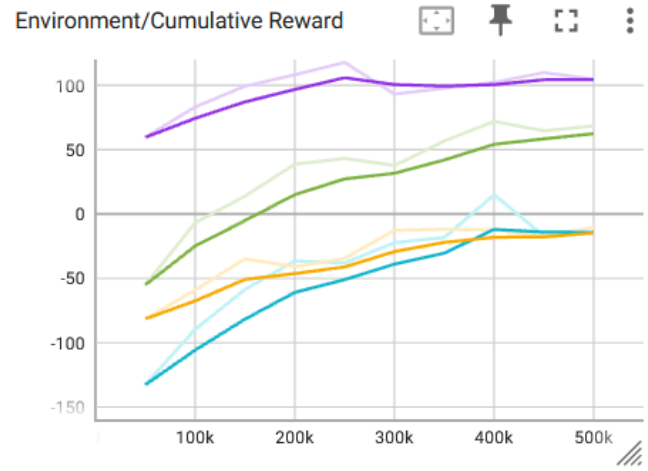

Fig. 8: Cumulative Reward over 500K Episodes for D_1 (Blue), D_3, (Yellow), D_4 (Green), and D_5 (Purple)

VI. MiniMech

Expanding our enemy field further is the non-stationary MiniMech enemy. The MiniMech enemy acts as the Player's counterpart adopting all layer capabilities with slightly different physics, increased agility, and a varying spread of tactics and abilities. This set of tactics and movements, specifically designed for the enemy, are almost entirely reactive allowing the MiniMech to adjust maneuvers according to decisions made by the incoming player. These aspects of the MiniMech enemy paired with its advanced situational awareness introduce a new level of challenge and excitement to the game.

Fig. 9: MiniMech Enemy

A. Movement and Mechanics

The MiniMech enemy serves to give the player more of a challenge and has slightly more complex behaviors. It can move, run, and jump away from and towards the Player. It has two attacks, a *machine gun attack* and a *long-range attack*. The movement and attack patterns of the MiniMech are determined on numerous factors, especially its *awareness* mechanics. This mechanic allows the MiniMech to the number and quality of other enemies its surrounding area and changes its tactic towards the player accordingly. The *awareness* facet allows the enemy to determine its tactile approach as learned through its AI model. There are three levels of *approach*: *don't approach*, *neutral*, and *approach*. The more health the Player has, the more aggressive the tactic and higher the level of *approach* the MiniMech will have. It will also move toward avoiding tactics as its own health gets lower. The *awareness* feature interacts with the *approach* mechanic:

1) If the *awareness* of the MiniMech is *low* (*enemyCountInRange* = *0*), the MiniMech tactic will stay in its current state
2) If the *awareness* of the MiniMech is *moderate* (*enemyCountInRange* = *1*), if the MiniMech is currently at an *approach* tactic, it will move to a *neutral* tactic, otherwise it will stay in its current state
3) If the *awareness* of the MiniMech is *high* (*enemyCountInRange* ≥ *2*), the MiniMech tactic will stay *neutral* unless the *awareness* increases to which it will switch to *don't approach*

Each *approach* tactics results in the MiniMech attempting to keep a further distance from the player. This same mechanic can also influence its attack patterns. The MiniMech's *machine gun attack* is an intermittent 3 bullet pulse with 0.7 seconds between each pulse and deals 2 damage to the Player each bullet. The *long-range attack* is a one pulse laser attack that takes 3 seconds to charge up and deals 10 damage. This attack is only used when the Player's health is high or the MiniMech is in its *don't approach* tactic. Its attack pattern is also learned through Reinforcement Learning.

B. Observations & Decisions

The observations for the MiniMech were similar to the Sentry Turret, except it also included the MiniMech's awareness as an input. As such, the MiniMech's model takes

in *playerPosition* (Vector3), *miniMechPosition* (Vector3), *playerHealth* (float), *miniMechHealth* (float), and *awareness* (float). The total *Vector Observation Space Size* for the ML Agents behavior parameters totaled 9. These observations will be obtained by feeding in the appropriate value from the Player and MiniMech in real time. The model consisted of 2 *Discrete Branches* from which to make decisions with the first branch having a size of 3, and the second branch only having a size of 2. These outputs were *approach* and *weaponChoice*. The *approach* exhibits range *{0, 1, 2}* whereas a 0 represents *don't approach*, 1 for *neutral*, and 2 for *approach*. The *weaponChoice* will be similar to the other agents, 0 for *machine gun attack* and 1 for *long range attack*.

C. Reward Mechanisms

The outputs of the MiniMech's model, approach and weapon choice give us enough output about the MiniMech current situation to allow us to guide its behavior it through Reinforcement Learning. The main feature of the MiniMech's reward system is its ability to determine how to accurately select the correct *approach* and *weaponChoice* given the input parameters. This is achieved in a similar manner to the two agents we discussed previously, again focusing on not simply having the model train to kill the Player, but to train the model to get the Player to a heightened stage of engagement and fun. When the MiniMech hits the Player, they get +2 reward, -0.2 reward if they miss. There is an extra +1 reward per Player hit if the MiniMech hits a long-range attack. If the Player's health is greater than 80% and the model chooses an *approach* tactic, it receives a +1 reward. If it chooses *neutral* when the Player's health is less than 20%, it gets a +0.5 reward, but if it uses a long-range attack in this same health range it gets –0.5 reward. When the MiniMech's health reaches below 20% and it uses the *don't approach* tactic, it gains +0.5 reward and another +0.5 reward if it hits using *long range attack*. If the MiniMech dies, it receives –0.2 reward, but if the Player dies it receives +1 reward. The *awareness* feature was implemented into the reward system as well. If *awareness* is *high*, the model is rewarded +0.2 for a *don't approach*, -0.1 for a *neutral*, and –0.5 for an *approach*. If the *awareness* is *moderate* and the output was still *approach*, the model was given –0.2 reward.

D. Testing and Training Models

Implementing the MiniMech was a little more challenging than the last two enemies, as the MiniMech was able to do almost every action the Player was able to do, but on a smaller scale and with more agility, and overall movement capabilities. We ran into plenty of physics bugs and the same rotation issues as with the Drone; However, by this time we were able to fix them relatively quickly and get right to training.

We once again created a separate training environment in Unity specifically for the MiniMech's abilities. As such, they weren't simply just 9 copies of the same training scene as we wanted the MiniMech to train to use its *awareness* feature. Thus, we trained it in 3 environments with *low awareness*, 3 with *moderate awareness,* and 3 with *high awareness*. The same training configuration was used as in Table 1, along with training one model for 2M episodes alongside the other 500K step models. One of the training environments is shown in Fig. 10.

Fig. 10: MiniMech Training Environment in Unity

E. Model Analysis

There was a lot of effort put into planning out the reward system for the MiniMech, which resulted in not having to train as many different variations of models. Instead, we went with a certain approach for each model and trained each one numerous times with that approach. This resulted in three main models, MM_1, which did not factor in *awareness*, MM_2, which factored in *awareness* and an extra negative 0.5 reward if the MiniMech used its *long-range attack* when the Player's health was below 20%, and MM_3, which simply trained for 2M episodes instead of 500K. As shown in Fig. 11, MM_1 seemed to perform better, however it did not incorporate any *awareness* into its model, so it wasn't the intended behavior we were looking for. MM_3 had a very similar performance to MM_2 and exhibited the desired behavior we were looking for. During gameplay it changed tactics appropriately, which was especially evident in sections of the game where multiple enemies appeared. MM_3 would back up and switch to *long range attack* as intended. Thus, MM_3 was chosen to be implemented into K7 Mech Runner.

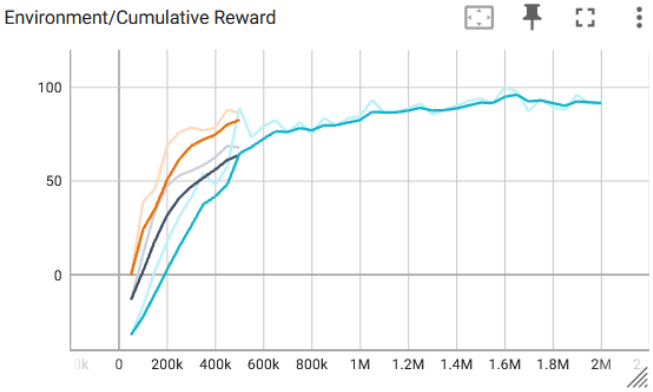

Fig. 11: Cumulative Reward over 2M Episodes, MM_1 (Orange, 500K), MM_2 (Dark Gray, 500K), and MM_3 (Blue, 2M)

VII. Conclusion

In this paper, we've discussed K7 Mech Runner's integration of Deep Neural Networks and Reinforcement Learning into video game mechanics, with a main focus on increased and sustained player engagement. Through the leveraging of machine learning, we created a dynamic gaming experience specifically catering to the skill level of each player.

In the development and implementation of AI models for various enemies such as the Sentry Turret, Drone, and MiniMech, we were able to demonstrate how these models

can learn and adapt their behaviors based on real-time observations of player and environmental factors. By providing each enemy with unique decision-making capabilities influenced by factors like player health and enemy awareness, we created a challenging yet immersive and stimulating environment for players. We found that the dynamic adaptation of enemy behavior fosters a constant sense of challenge, while avoiding the defeat of impossibility for players, making prolonged interaction with the game more likely. Even further, the modular and dynamic nature of creating these enemies and behaviors introduces a pipeline for creation of which game developers like us can create custom-tailored gameplay experiences without the intervention of extensive manually scripted events.

In conclusion, the integration of Deep Neural Networks and Reinforcement Learning into video game mechanics holds significant promise for the future of the video game industry, offering an avenue to create dynamic, immersive, and engaging experiences that adapt to each player's unique style and skill level. As technology continues to advance, we look forward to seeing how these approaches shape the future of the game industry as seen in our experience with K7 Mech Runner.

ACKNOWLEDGMENTS

This work was supported by advisors Fatima Kuehn and Cole Levine, artists Xiaorui Ma and Mark Leon, and Sponsor DevTeam7 and its founder Dennie Guy.

REFERENCES

[1] Spronck, Pieter, et al. "Adaptive game AI with dynamic scripting." *Machine Learning*, vol. 63, no. 3, 9 Mar. 2006, pp. 217–248, https://doi.org/10.1007/s10994-006-6205-6.

[2] Spronck, Pieter et al. "DIFFICULTY SCALING OF GAME AI." 2004. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=bbdb1658669fce390099d6f7f3820c8bed6b3e91

[3] Bowling, Michael, et al. "Machine learning and games." *Machine Learning*, vol. 63, no. 3, 10 May 2006, pp. 211–215, https://doi.org/10.1007/s10994-006-8919-x.

[4] M Charity, Michael Cerny Green, Ahmed Khalifa, and Julian Togelius. 2020. Mech-Elites: Illuminating the Mechanic Space of GVG-AI. In International Conference on the Foundations of Digital Games (FDG '20), September 15–18, 2020, Bugibba, Malta. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3402942.3402954

[5] Arzate Cruz, Christian, and Jorge Adolfo Ramirez Uresti. "Player-centered game AI from a flow perspective: Towards a better understanding of past trends and Future Directions." *Entertainment Computing*, vol. 20, May 2017, pp. 11–24, https://doi.org/10.1016/j.entcom.2017.02.003.