

East Tennessee State University

Digital Commons @ East Tennessee State University

Undergraduate Honors Theses

Student Works

12-2021

Natively Implementing Deep Reinforcement Learning into a Game Engine

Austin Kincer

Follow this and additional works at: <https://dc.etsu.edu/honors>



Part of the [Artificial Intelligence and Robotics Commons](#)

Recommended Citation

Kincer, Austin, "Natively Implementing Deep Reinforcement Learning into a Game Engine" (2021). *Undergraduate Honors Theses*. Paper 653. <https://dc.etsu.edu/honors/653>

This Honors Thesis - Open Access is brought to you for free and open access by the Student Works at Digital Commons @ East Tennessee State University. It has been accepted for inclusion in Undergraduate Honors Theses by an authorized administrator of Digital Commons @ East Tennessee State University. For more information, please contact digilib@etsu.edu.

Natively Implementing Deep Reinforcement Learning into a Game Engine

By

Austin J. Kincer

An Undergraduate Thesis Submitted in Partial Fulfillment
of the Requirements for the
University Honors Scholars Program
Honors College
and the
Honors-in Computing Program
College of Business and Technology
East Tennessee State University


Austin J. Kincer 11/17/2021
Date


Dr. Jeffrey Roach, Thesis Mentor 11/17/2021
Date


Dr. Brian Bennett, Reader 11/17/2021
Date



UNDERGRADUATE HONORS THESIS AVAILABILITY AGREEMENT and NON-EXCLUSIVE DISTRIBUTION LICENSE

By signing and submitting this license, the author grants to Digital Commons @ East Tennessee State University (ETSU) the non-exclusive right to reproduce and distribute the author's submission in electronic format via the World Wide Web, as well as the right to migrate or convert it, without alteration of the content, to any medium or format for the purpose of preservation and/or continued distribution.

ETSU acknowledges that this is a non-exclusive license. Any copyrights in the submission remain with the author or other copyright holder and subsequent uses of the submitted material by that person are not restricted by this license.

The author agrees that ETSU may keep more than one copy of this submission for the purposes of security, backup and preservation.

The author declares that the submission covered by this license is his/her original work and that he/she has the right to grant this license to Digital Commons @ East Tennessee State University (ETSU). The author further represents that the submission does not, to the best of his/her knowledge, infringe upon any third-party's copyright. If the submission contains material for which the author does not hold copyright, the author represents that he/she has obtained the unrestricted permission of the copyright holder to grant this license to Digital Commons @ East Tennessee State University (ETSU) and that such third-party material is clearly identified and acknowledged within the text or content of the submission. In the event of a subsequent dispute over the copyrights to material contained in this submission, the author agrees to indemnify and hold harmless ETSU and its employees or agents for any uses of the material authorized by this license.

Digital Commons @ East Tennessee State University (ETSU) will make the submission available to the public using a Creative Commons Attribution / Non-commercial / No Derivatives license and will take all reasonable steps to ensure that the author's name remains clearly associated with the submission and that no alterations of the content are made.

Please select one of the following honors thesis availability options:

No restriction on availability

1 year embargo

2 year embargo

I agree to the terms of this Non-Exclusive Distribution License and Availability Agreement:

	Austin Kincer	11/17/2021
Honors Thesis Author Signature	Print Name	Date
	Jeffrey Roach	11/17/2021
Honors Thesis Mentor Signature	Print Name	Date

Acknowledgements

I would like to thank my supervisor, Dr. Jeffrey Roach, whose expertise in games, game engines, and design helped me design and implement a completely functional game engine. Your comments and questioning helped me stay in scope of the project.

I would like to thank Dr. Phil Pfeiffer for all his comments and proofreading. The after class discussions helped me generate many supporting ideas, and without your help, this thesis would have taken much longer with a much worse quality.

Abstract

Artificial intelligence (AI) increases the immersion that players can have while playing games. Modern game engines, a middleware software used to create games, implement simple AI behaviors that developers can use. Advanced AI behaviors must be implemented manually by game developers, which decreases the likelihood of game developers using advanced AI due to development overhead.

A custom game engine and custom AI architecture that handled deep reinforcement learning was designed and implemented. Snake was created using the custom game engine to test the feasibility of natively implementing an AI architecture into a game engine. A snake agent was successfully trained using the AI architecture, but the learned behavior was suboptimal. Although the learned behavior was suboptimal, the AI architecture was successfully implemented into a custom game engine because a behavior was successfully learned.

Contents

1	Introduction.....	1
2	Literature Review.....	1
2.1	Game Engines	1
2.2	AI Scripting.....	5
2.3	Case-Based AI.....	8
2.4	Deep Learning.....	10
2.5	PPO Algorithm.....	16
3	Methodology	18
3.1	Overview.....	18
3.2	Experimental Framework.....	18
3.2.1	The game-playing framework.....	18
3.3	Training.....	21
3.4	Testing.....	21
3.4.1	Criteria	21
4	Results.....	21
4.1	GRAVEngine.....	21
4.1.1	Fiber Job System.....	21
4.1.2	Rendering System	23
4.1.3	Layer System	23
4.1.4	Event System	23

4.1.5	Supporting Utilities	24
4.2	AI Architecture	24
4.2.1	Environment Manager.....	24
4.2.2	PPO Algorithm.....	24
4.3	Game	25
4.3.1	Statistics Graphs.....	25
5	Conclusions, Implications, and Recommendations	25
5.1	Conclusions.....	25
5.2	Implications.....	26
5.3	Recommendations.....	26
6	References	28

$$S = \frac{1}{(F + (1-F)/N)}$$

Figure 1. Amdahl's Law (Tulip et al., 2006)

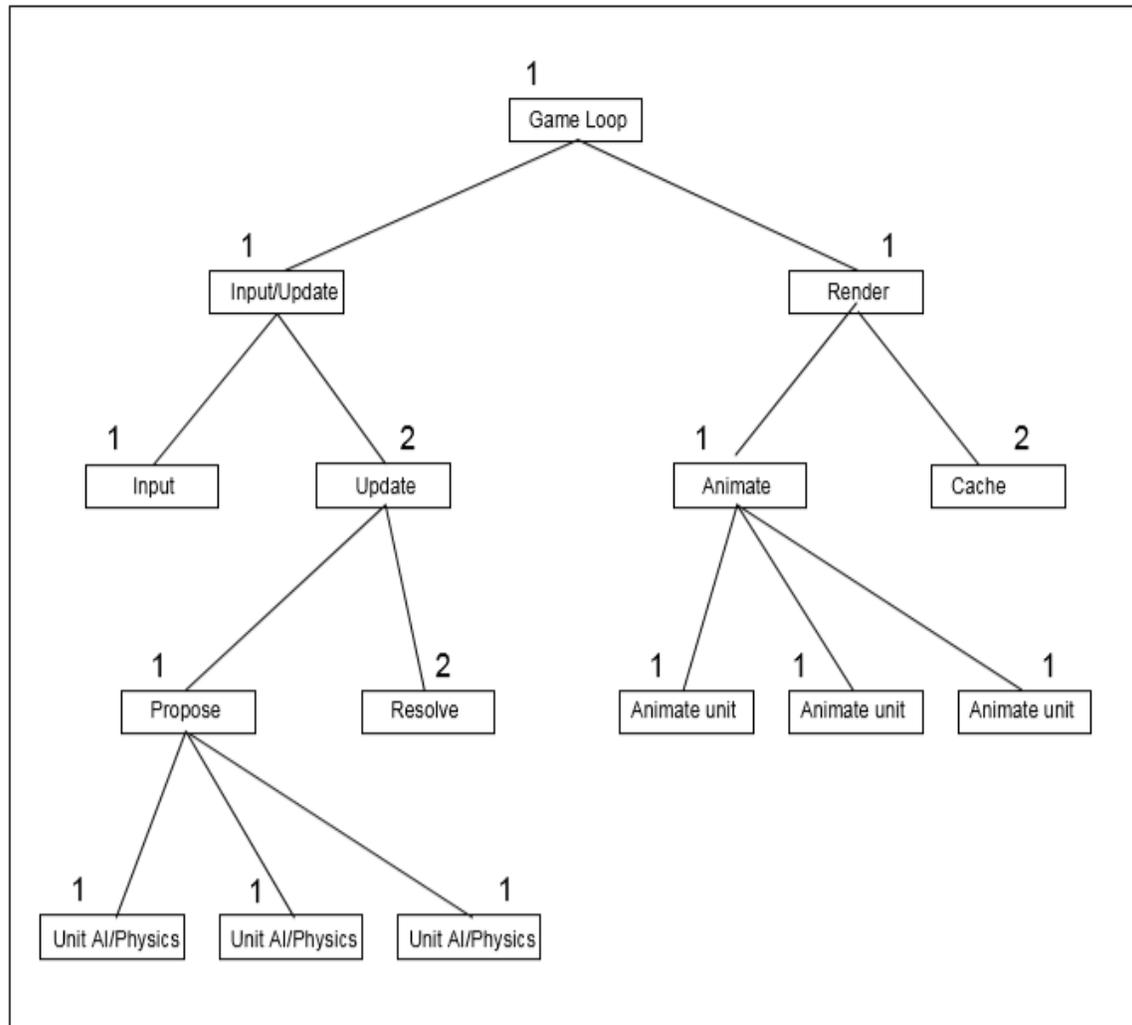


Figure 2. Task Tree (Tulip et al., 2006)

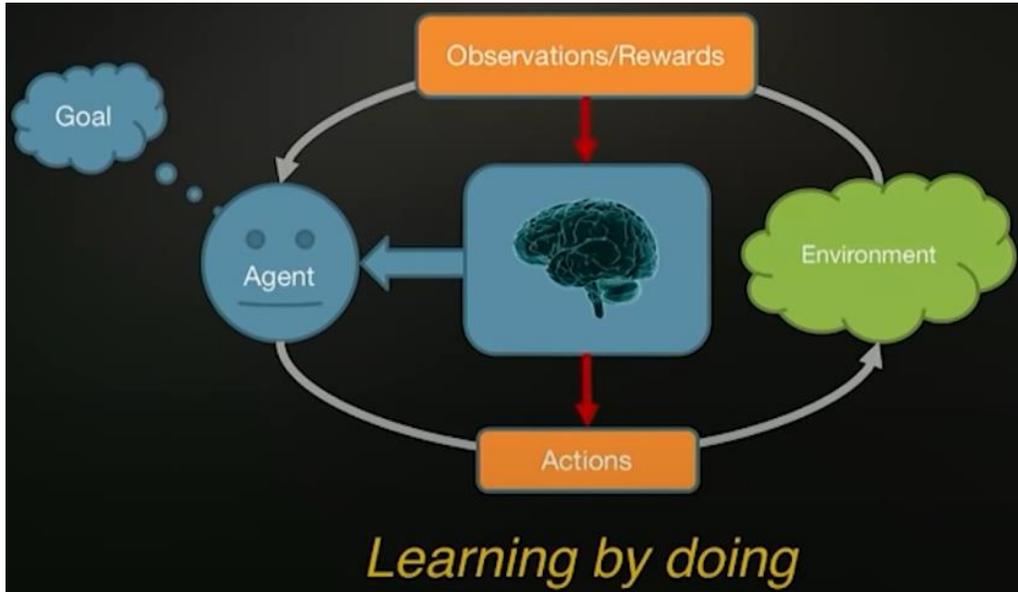


Figure 3. Deep Reinforcement Learning (Nordin, 2018, 15:00)

$$\hat{g} = \hat{\mathbb{E}}_t \left[\nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \hat{A}_t \right]$$

Figure 4. Common Gradient Estimator (Schulman, Wolski, Dhariwal, Radford, & Klimov, 2017)

$$L^{PG}(\theta) = \hat{\mathbb{E}}_t \left[\log \pi_{\theta}(a_t | s_t) \hat{A}_t \right].$$

Figure 5. Automatic Differentiation Objective (Schulman, Wolski, Dhariwal, Radford, & Klimov, 2017)

$$\begin{aligned} & \underset{\theta}{\text{maximize}} \quad \hat{\mathbb{E}}_t \left[\frac{\pi_{\theta}(a_t | s_t)}{\pi_{\theta_{\text{old}}}(a_t | s_t)} \hat{A}_t \right] \\ & \text{subject to} \quad \hat{\mathbb{E}}_t [\text{KL}[\pi_{\theta_{\text{old}}}(\cdot | s_t), \pi_{\theta}(\cdot | s_t)]] \leq \delta. \end{aligned}$$

Figure 6. TRPO Objective Function (Schulman, Wolski, Dhariwal, Radford, & Klimov, 2017)

$$\underset{\theta}{\text{maximize}} \quad \hat{\mathbb{E}}_t \left[\frac{\pi_{\theta}(a_t | s_t)}{\pi_{\theta_{\text{old}}}(a_t | s_t)} \hat{A}_t - \beta \text{KL}[\pi_{\theta_{\text{old}}}(\cdot | s_t), \pi_{\theta}(\cdot | s_t)] \right]$$

Figure 7. Unconstrained Optimization Problem (Schulman, Wolski, Dhariwal, Radford, & Klimov, 2017)

$$L^{CPI}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t | s_t)}{\pi_{\theta_{\text{old}}}(a_t | s_t)} \hat{A}_t \right] = \hat{\mathbb{E}}_t \left[r_t(\theta) \hat{A}_t \right].$$

Figure 8. Surrogate Objective Function (Schulman, Wolski, Dhariwal, Radford, & Klimov, 2017)

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min(r_t(\theta) \hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t) \right]$$

Figure 9. Clipped Surrogate Objective Function (Schulman, Wolski, Dhariwal, Radford, & Klimov, 2017)

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t) \right]$$

Figure 10. Combined Objective Function (Schulman, Wolski, Dhariwal, Radford, & Klimov, 2017)

$$\hat{A}_t = -V(s_t) + r_t + \gamma r_{t+1} + \dots + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} V(s_T)$$

Figure 11. Advantage Estimator (Schulman, Wolski, Dhariwal, Radford, & Klimov, 2017)

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \dots + \dots + (\gamma\lambda)^{T-t+1}\delta_{T-1},$$

where $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$

Figure 12. Generalized Advantage Estimator (Schulman, Wolski, Dhariwal, Radford, & Klimov, 2017)

Algorithm 1 PPO, Actor-Critic Style

```

for iteration=1, 2, ... do
  for actor=1, 2, ..., N do
    Run policy  $\pi_{\theta_{\text{old}}}$  in environment for  $T$  timesteps
    Compute advantage estimates  $\hat{A}_1, \dots, \hat{A}_T$ 
  end for
  Optimize surrogate  $L$  wrt  $\theta$ , with  $K$  epochs and minibatch size  $M \leq NT$ 
   $\theta_{\text{old}} \leftarrow \theta$ 
end for

```

Figure 13. PPO Algorithm (Schulman, Wolski, Dhariwal, Radford, & Klimov, 2017)

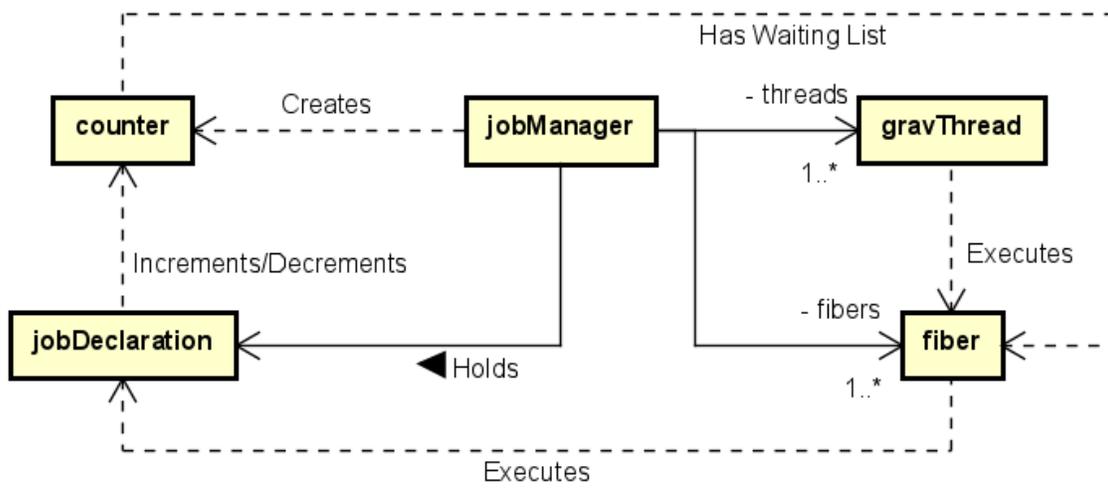


Figure 14. Job System

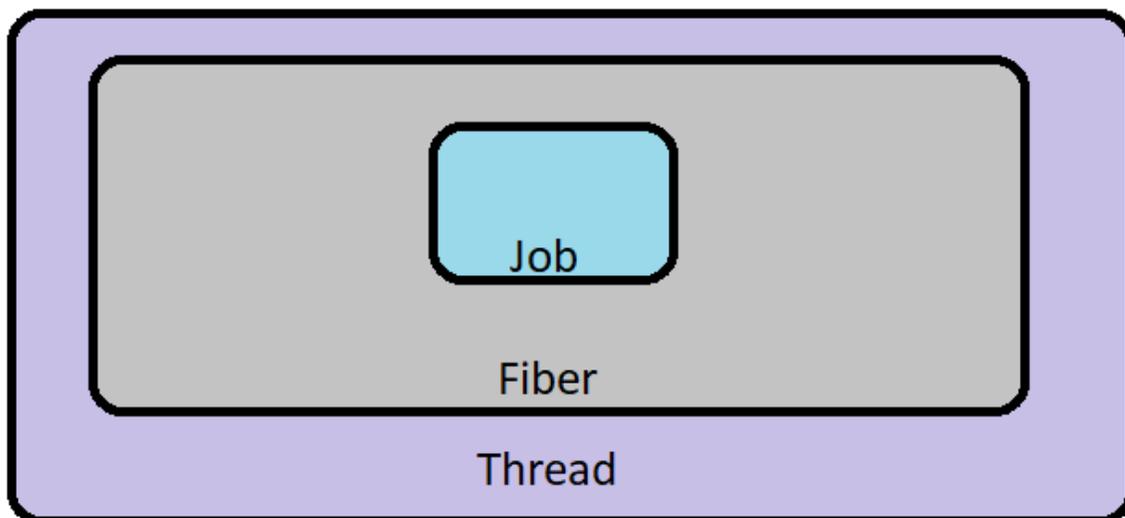


Figure 15. Thread and Fiber Execution Context

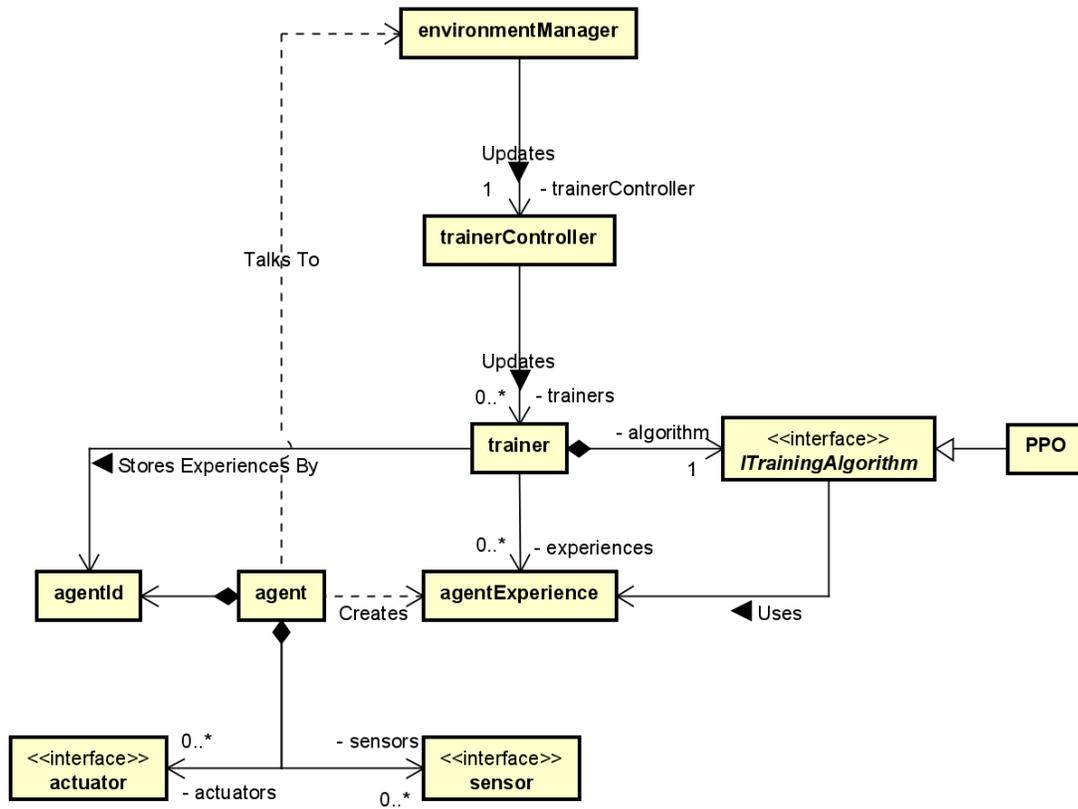


Figure 16. Environment Manager System

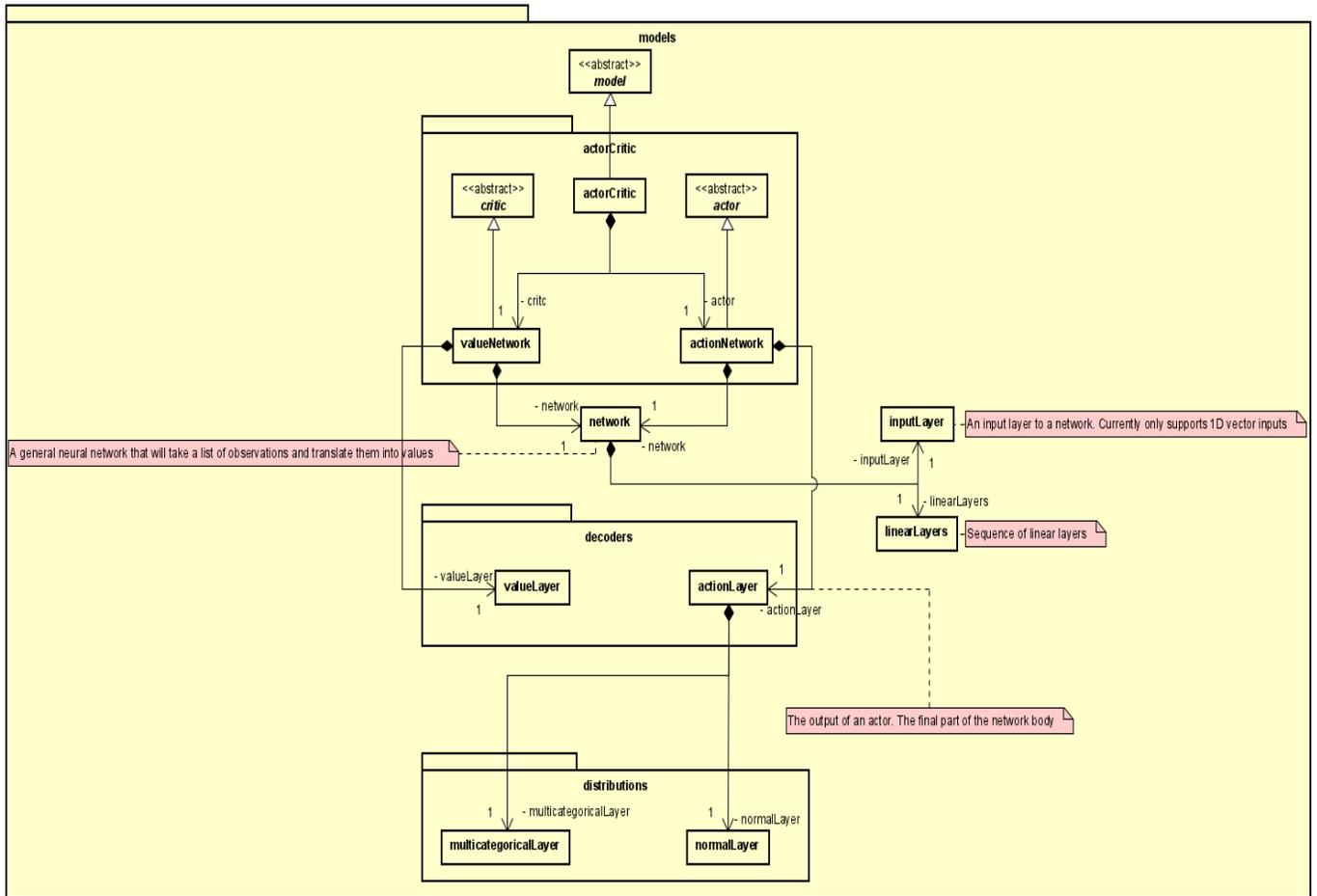


Figure 17. AI Models

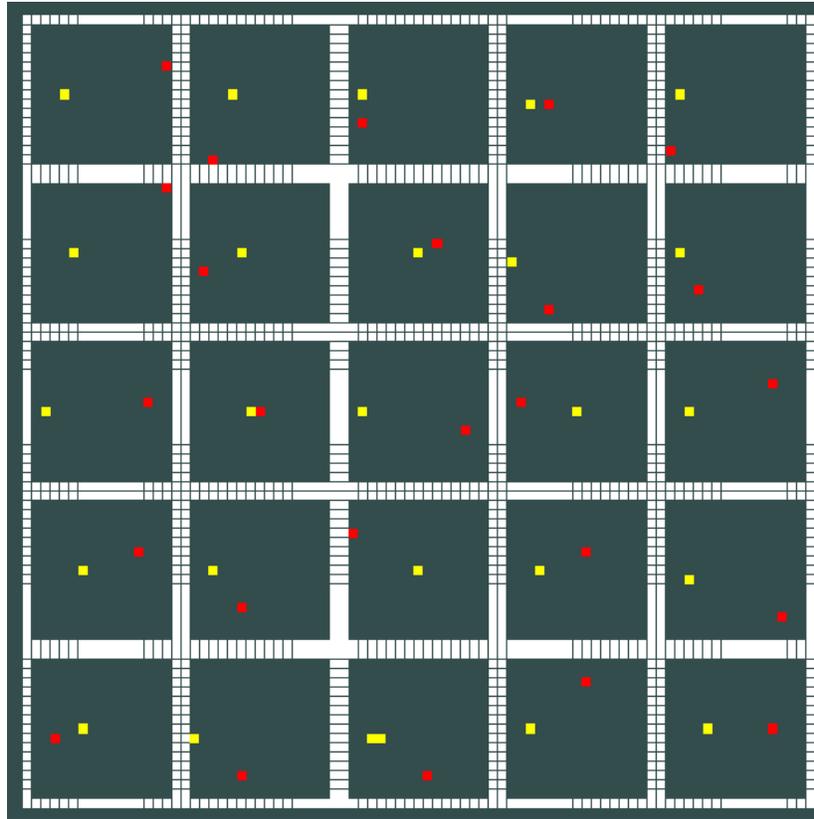


Figure 18. Example Snake Game

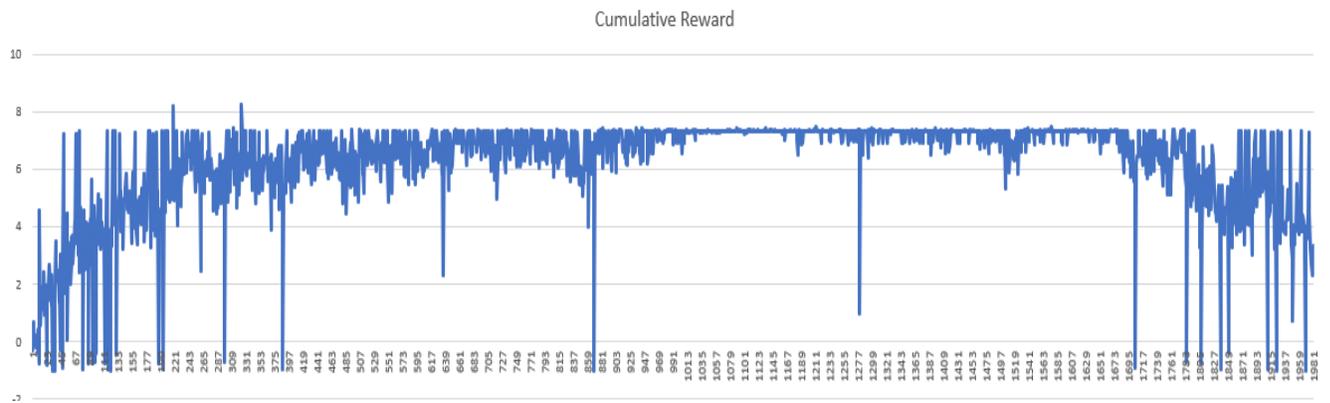


Figure 19. Cumulative Reward vs Epoch

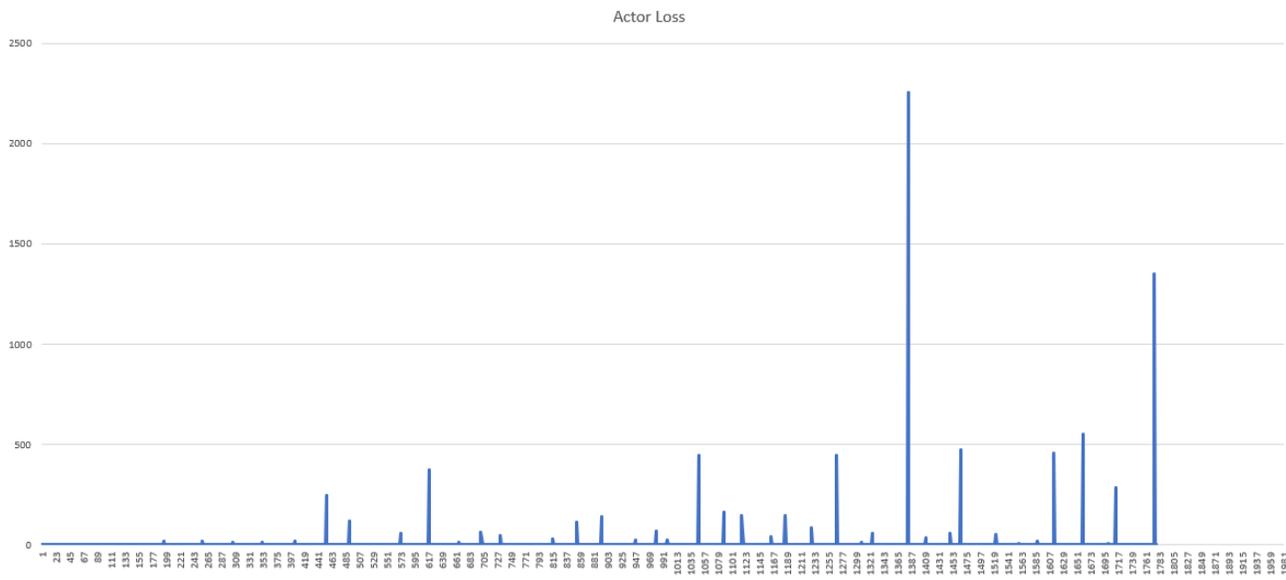


Figure 20. Actor Loss vs Epoch



Figure 21. Critic Loss vs Epoch

1 Introduction

In 2006, the gaming industry had a net revenue of 40 billion USD (Tulip, Bekkema, & Nesbitt, 2006). Early games were created such that developers had to create all the supporting functionality to run a game. Modern games are created using game engines as a middleware software. Developers are not interested in how games are supported, but developers are interested in designing the content of a game.

Artificial intelligence (AI) increases the immersion that players can have while playing games, but AI are not advanced in their behavior. AI is implemented into games in the same way that early games were created. Developers create all the supporting functionality along with any AI. Without middleware tools, designers are less likely to use advanced AI techniques like deep reinforcement learning in games due to development overhead.

Because game engines do not have AI middleware, a custom AI architecture needs to be designed and implemented into a game engine. A custom game engine and AI architecture was designed to test whether it is possible to have an advanced AI architecture natively implemented into a game engine. The AI learning models created in the AI architecture were assumed to only cover deep reinforcement learning models. A snake agent was successfully trained, but the behavior was suboptimal.

2 Literature Review

2.1 Game Engines

As early as 2006, gaming had grown into a major industry, rivaling Hollywood in revenue at 40 billion USD (Tulip, Bekkema, & Nesbitt, 2006). At that time, game consoles were owned by more than three quarters of U.S. households. Game technology was also being used by non-game-related industries like medicine and the military, for teaching and creating simulated environments.

Since 2006, games have increased in popularity, complexity, content, and costs. They are being released to market more quickly, have shorter shelf lives, and incur more financial risk. In their 2006 study, Tulip et al. found that only one out of ten games released would recover their development costs. Most of a game's development cost is due to generic functionality, such as entity rendering, scene

management, and physics simulation.

To reduce the cost of game development, developers use game engines: conglomerations of generic functionality that can be used to develop different types of games. Game engines are well tested and optimized, which allows developers to focus on the logic for their own games. Game engine developers typically profit from licensing their engines, amortized over multiple games.

Over time, changes in computer hardware have affected strategies for game development. Due to problems related to dissipating waste heat, changes in chip fabrication technology no longer produce significant speedups in successive generations of individual processors. As a result, chip manufacturers like Intel and AMD now deliver additional processing power using multi-core chips. Early game engines can't take advantage of these new chips, as those engines are optimized for single core processors with little multithreading; these older engines, in fact, can perform more poorly when run on multicore processors.

To take advantage of multi-core processors, game engines should be designed for concurrent execution, including features such as rendering and IO. As of 2006, however, Tulip et al. (2006) found that there was little to no literature on applying multithreading and concurrency to game engines. Key considerations in concurrent programming that apply to multi-core game design include granularity, load balancing, task and data parallelism, and Amdahl's Law.

Granularity is the amount work that is executed per task. Fine-grained and coarse-grained tasks execute small and large amounts of work, respectively. Fine-grained task loads can increase processor utilization by ensuring a more even distribution of workload, at the expense of additional context switching overhead. Coarse-grained task loads exhibit less overhead but potentially poorer utilization due to increased idle time.

Task parallelism involves splitting an application's code into multiple tasks and synchronizing (i.e., coordinating) their execution at points where tasks need to share results. Task parallelism is suitable for groups of tasks that execute independently for long periods of time, relative to the amount of data they exchange. It's also suitable for high latency operations such as IO and network communication, as other

tasks can execute while a high latency task is waiting to finish.

Data parallelism involves applying a common operation to a group of datasets that can be processed independently; an extreme case of this is rendering individual frames of an animated film.

Intuitively, Amdahl's Law (Figure 1) states that the degree to which concurrency can speed a program's execution is limited by the amount of sequential code it contains. Here, S is the speed up acquired, F is the fractional amount of code that is sequential, and N is the number of processors. As Figure 1 indicates, potential speedup from parallelism is asymptotic; more than 90% of a program must be parallelizable in order to achieve what is commonly referred to as an order of magnitude (10x) speedup.

Computer games are generally controlled using game loops: routines that step through a simulation, interacting with the game's modules. After an initial step in which a game loads its resources, the game activates its loop, which repeatedly accepts user input, simulates changes to the environment, then renders the changed environment. Input can be obtained by polling input devices or using events to detect content pushed to buffers from hardware. The simulation step typically includes an update phase, which optionally uses input to change the game's state, and a resolution phase, which resolves any conflicts or interactions in the proposed game state. The rendering step renders the new state through visual and audio effects. Rendering should be decoupled from simulation; the simulation should work regardless of whether output is rendered.

This three-step event loop can be realized as a three-component modular system, where each component is further divisible into subsystems: i.e., resource management, networking, sound, rendering, scene management, GUI, water, weather, physics, and AI. The game engine's core is its kernel, which handles task coordination and synchronization.

Tulip et al. (2006) discuss which of a game's subsystems are task parallelizable, data parallelizable, and inherently sequential. Resource loading, input gathering, and networking, three information producing, high latency IO operations, are task parallelizable; Tulip et al. recommend dedicating a separate thread to each of these subsystems. Sound and visual rendering are also task

parallelizable: these are service tasks that produce the frames that games display. Individual frames can be rendered in parallel as multiple frames can be rendered in-between update phases. A final parallelizable task, scene management, is a preprocessing step that buffers information between the update and the rendering phases; it limits the amount of information sent to rendering processes. Long-lived service tasks like these have little overhead compared to normal tasks that are created and destroyed constantly.

After scene management, some rendering operations are data parallelizable. Interpolating between keyframes, skinning models, and applying textures are data intensive operations that can be done independently. Data parallel scene management operations include visibility and audibility testing.

Render tree building and resolving entity interactions, by contrast, are inherently sequential. Render -tree building includes a step that sorts graphical primitives based on previous rendering transformations. Entity interactions, as a rule, require that the modeling of previous, related interactions be complete before their interactions can be resolved. While actions that are determined to be mutually independent can be resolved in parallel, such interactions can be difficult to identify.

Tulip et al. found that a game loop's three-step processes are mostly independent. Input gathering can be split into subtasks. The environment simulation can have data parallelizable subtasks but has a core serial task of interaction resolution; like input gathering, rendering is composed of subtasks and data parallel operations.

Tulip et al. tested their three-step game loop architecture using a task tree (Figure 2). The tree is a hierarchical scheduling model that splits the game loop into subtasks. The model's leaves are executable tasks and nodes are execution groups. Leaves and nodes have scheduling numbers that determine their scheduling order; tasks with equal numbers execute in parallel and lower-order groups must finish before higher order groups are scheduled. Rendering and Input/Update nodes can execute concurrently as they have the same schedule ordering. In the Input/Update node, Input is scheduled before the Update node as the Update node has a higher schedule order.

To assure that this task tree is efficient and scalable, Tulip at al. use a thread pool as their underlying executing platform. A thread pool uses reusable worker threads to eliminate the overhead for

creating and destroying threads. The tree is scalable because the thread pool's thread count is independent of the tree structure; the tree schedules executable segments and the thread pool handles how tasks execute.

Tulip et al. found that their task tree helps with scheduling and exposes opportunities for parallelizing a game engine's functions, including input gathering, updating, and rendering.

2.2 AI Scripting

Games are designed to entertain players. Challenges such as combat entertain by giving difficulties for players to overcome. In (2009), Szita et al. assert that AIs for combat games are interesting to the degree that they use effective and diverse tactics to engage players. AIs that are effective and diverse, however, are difficult to devise. Effective AIs usually use specific sequences of tactics with little variety. Adaptive game AI should improve effectiveness while creating diversity through trying new tactics. However, according to Szita et al., adaptive AI has not advanced to where it can be used in commercial games; at that time, games that implement adaptive AI had generated ineffective AIs that caused players to easily lose interest.

Scripting is one of the dominant ways to create game AIs. Simple AI scripts are easy to implement, interpret, and modify. More complex scripts, however, are labor intensive to create and time consuming to change by hand. Designers can also introduce weaknesses into AIs by overlooking scenarios when hand-coding scripts. Scripts, however, produce predictable AIs; players can quickly find patterns in scripts and exploit them. Finally, scripts, being set codes, cannot adapt to novel situations.

To create more diverse game AI, designers can write multiple scripts for different situations. More scripts, however, require additional programming and testing. The individual scripts are still also static; they can contain easy-to-exploit tactics and won't adapt to new situations. While randomness can yield more diverse AIs, it can also produce undesirable behaviors. This causes AI to seem out of place and degrade players' sense that AIs show intelligent behavior.

Dynamic scripting is a reinforcement learning technique where game AI scripts automatically

learn effective strategies. It is fast, efficient, and can handle multiple different situations. For each opponent that an AI faces, a dynamic script references a custom rule base; this allows for the use of effective strategies that can change over time. Rules inside rule-bases are designed by hand with knowledge about the situations game AI will encounter and are chosen by using probabilities proportional to the weights associated with a rule's success rate. Behaviors are changed by reweighting rules; successes increase a rule's weight while failures lower it.

Although dynamic scripting can adapt to new game situations, it cannot guarantee convergence and diversity. Although that may sound like a detriment, dynamic scripting's inability to guarantee convergence is an advantage. If a dynamic script converges, it cannot adapt to a changing game environment. Also, optimal strategies from convergence can overfit to specific situations in games. When overfitting occurs, dynamic scripts fail to properly handle other situations: a characteristic of unintelligent behavior.

Diversity with dynamic scripts is hard to achieve. Because dynamic scripting uses reinforcement learning, learning is incremental. For a game AI to appear intelligent, it must use different strategies to compete in different game environments. Since transitioning a dynamically scripted game AI to a new environment takes time and many increments, AIs can be expected to use suboptimal strategies during their transitions. Due to this limitation, dynamic scripting alone cannot be relied on to improve AI diversity. Dynamic scripts will converge to static and predictable scripts.

Szita et al. describe a variant of dynamic learning that preserves a base strategy's effectiveness while increasing its diversity. Their method uses macros and interestingness to augment scripts. Macros are sequences of actions (singular rules) that are treated as a single action. They augment dynamic scripting by using bigger building blocks that can expedite switches between playing styles. They are also found in literature under other names, like portions, behaviors, and skills.

Interestingness is the art of balancing the use of previously acquired knowledge with knowledge from new situations. The authors define interestingness in terms of boring knowledge. Boring knowledge is knowledge that is either trivial—i.e., well known—or so hard to understand that it cannot be readily

used. Interesting areas are ones in-between boring ones; they are not as well-known and fluid, allowing for lots of knowledge to be obtained. Szita et al. found that most of the literature on knowledge exploration describes policies for acquiring optimal strategies for achieving goals. This focus fails to address the need for diversity, a key component of interestingness.

Macro learning should have effective macros, diverse macros, and appropriately sized macros. To construct macros, Szita et al.'s AI selects rules from its rule base using a probability distribution based on macro fitness. Strong macros have a higher probability of one of their rules appearing in other macros. Szita et al.'s fitness function rewards macros when they are strong and effective or when they differ from other known macros. Strong macros create effective game AI and differing macros create diverse game AI.

Szita et al. implemented macro learning in AIs for the open-source combat simulator MiniGate. MiniGate was selected for the large variety of its feasible tactics, its different playing styles, and its simple rule set. In the authors' simulator, two wizards duel to defeat one another. Szita et al.'s implementation used a static script to control one wizard and a dynamic script to control the other. Each of the dynamic script's macros begin as a new learning loop with a new rule probability vector. A script is then created from the loop, evaluated, and the results are recorded. The script's fitness is determined after its effectiveness and diversity is gauged from the results and the script's probability vector is updated from the fitness. Finally, the macro is added to the list of macros.

Szita et al. tested dynamic scripting against four static scripts that employ tactics and spells wizards can use during combat: summoning, offensive, optimized, and novice. The authors measured their dynamic scripts' changes in adaptation speed, playing strength, and diversity over 500 trials. They gauged their training algorithm's success by the number of battles won in the last 100 of 500 battles tested and their consistency by the average turning point: i.e., the point where the adaptive player's scores are better than the static player's scores over ten iteration steps. Low average turning points were taken to indicate that their algorithm is efficient in training dynamic scripts.

Szita et al. found that dynamic scripts that used macros outperformed dynamic scripts that used

only singular rules. Macro based dynamic scripting uniformly outperformed the four static script test cases while exhibiting more diversity. As such, Szita et al. concluded that dynamic scripting with macros is an effective strategy that can produce strong opponents with multiple tactics.

2.3 Case-Based AI

Modern video games appeal to users by providing immersive environments that feature consistent gameplay and realistic visual and auditory elements. Unfortunately, these AIs are normally of limited intelligence, due to a lack of readily accessible tools for designing complex AIs. AIs, for example, are easily exploitable once their strategies are learned; they cannot learn new tactics based on a player's play style (Bakkes, Spronck, & Herik, 2009). This lack of sophistication makes game AIs seem clunky and unentertaining, degrading a game's overall appeal.

To counter these problems, strategies for designing AIs with human-like behavior are needed. Adaptive game AI achieves this using machine learning. An example of this is scaling a game's difficulty based on the game's player; doing so yields a more immersive environment.

Incremental adaptive game AIs continuously make small changes to themselves to adapt to players based on the current game state. However, these AIs cannot change quickly enough to adapt to changing game states with reliable behaviors. They require large and high-quality in-game knowledge domains with many trials to learn effective behavior. This is undesirable in games as multiple trials or long game times are not normal. As such, incremental adaptive AI is not useful for complex games.

Case-based adaptive AI, an alternative to incremental adaption, uses immediate domain knowledge gathered from the game state without the resource-intensive learning required by incremental adaptive AI. Precomputed cases that match the current game state are used to find the best strategy to achieve an AI's goal. This reliance on cases requires an AI to have already seen a given state to provide an optimal response; thus, case-base adaptive AI requires an accessible knowledge base to train AIs. The upfront training is done either using automated systems with predefined AI to learn from or human played games.

Adapting a case-based AI to a game is typically a three-phase process, requiring offline processing, AI initialization, and online strategy selection. Offline processing involves saving timestamped, game states from multiple games using an indexing and clustering system for quick runtime access into the case base. Indexing and clustering are done offline because of the extensive calculations involved. To save space, redundant states are not saved. When starting a game, the AI is initialized with a known successful strategy based on the known opponent state. The initial strategy can change as the game progresses; it is initially set just so that the AI will have an effective starting strategy. During live gameplay, the AI's strategy will change based on the evolving game state. The current state is indexed as in offline processing, but it is used to determine which saved strategy will most likely achieve the AI's goal.

Case-based adaptive AI can be used with a preexisting AI, allowing developers to have even more control of game behavior. Training data from online and multiplayer games can easily be obtained and used to create an extensive knowledge.

Case-based adaptive AI has been found to be successful with a generic real time strategy (RTS). Bakkes et al. tested an adaptive AI to see if it could successfully win against an original AI with known behaviors and an AI with previously unobserved behaviors. In these two cases, the AI successfully adapted to the opponent's strategies and improved its winning rate, compared to a control group where the AI's adaptive mechanism was disabled.

Another test involved trying to tie the game for as long as possible as a form of difficulty scaling. If the AI can tie with its opponent for extended period, it can give the player the feeling that they are evenly matched. This is a desired effect as some players enjoy being challenged by an opponent of equal skill. While case-based adaptive AI can keep a tie balance throughout a game, eventually a tipping point is reached where the balance fails. Compared to the control, the case-based adaptive AI more effectively delayed this tipping point.

Case-based adaptive AI sometimes fails, due to an inherent randomness in games. But this is also a desirable outcome; an AI that wins every time will frustrate players and lose its immersive effects.

2.4 Deep Learning

Deep learning is using artificial neural networks (ANNs) to solve problems. ANNs are function estimators that produce an output from any well-formed input. While people do not fully understand how ANNs model problem spaces, they have been trained to solve a wide variety of problems, thus eliminating the need to write classic programming logic. Nordin (2018, 2:00) gives multiple examples of problems that have been solved using ANNs, including developing elements of complex games, converting voice to text and vice versa, and turning pictures into descriptions and vice versa. These include some problems that have not yielded to classic logic programming.

The use of ANNs in game development includes the development of game-playing ANN like OpenAI's agent for a limited version of the MOBA (Massive Online Battle Arena) Dota (Nordin 2018, 2:40). This agent competed effectively against real humans. Deep learning has also been used to capture human poses; large motion studios have used this to assist in animating characters. Other companies have sped game creation through dynamic content generation. Nordin (2018, 6:40) references Anastasia Opara's use of deep learning to create room interiors; every asset, from the wallpaper to the mirror's frame, was created dynamically.

Google's DeepMind and Nvidia researchers created an ANN that converts a voice to a different voice. This technology generates speech from a set of spoken words, instead of having to have a voice actor record every single line; this dramatically speeds content creation. These researchers also devised techniques for animating a face from a voice, eliminating the need for manual animation or motion capture. This could potentially allow a player's avatar to sound like the character they are playing and look like they're speaking what they say. Nordin (2018, 6:34) states that Nvidia's animation example was the best he has seen for voice to face animation.

In 2017, Nvidia used parameterized rules to create fake 2D faces using 30,000 celebrity images (Nordin, 2018, 7:50). None of the generated faces were from the initial training set; all looked realistic. While 3D models still cannot be generated, Nordin (2018, 8:40) believes that useful 3D generation should

be possible in a couple of years.

Finally, Nordin cited Google's DeepMind as an example of how emergent behavior can be created for rich environments (2018, 8:45). Procedurally generated worlds are often lifeless. In order to make them seem more real, the worlds should include life. DeepMind trained an agent that was merely designed to move forward; this agent could only control its musculature and started with no knowledge of how to control it properly. After training, the agent learned how to navigate around obstacles such as walls, barriers, pits, and holes.

Nordin observes that ANNs can be used to create new features by combining other, existing features (Nordin, 2018, 10:20). Modern games are normally violent because violence is one of the simplest interactions to model. Social interaction is hard to model due to the large numbers of subtle and unsubtle social cues and visualizations. World of Warcraft (WOW), a massive multiplayer online role-playing game (MMORPG), is largely limited to game-mechanic interactions because much of the intended social interaction proved too difficult to implement (*ibid.*, 11:50). Because of this, true role playing through board games is more popular than ever (*ibid.*, 12:00).

As a first step towards creating a true role-playing experience, Nordin (2018, 12:40) created a prototype VR skinning experiment. This experiment allows a user to control a character who can walk through its environment. If voice conversion and facial animation could be implemented in Nordin's prototype, a user could turn into any character. Unfortunately, a lack of practical models or technology for converting voices, animating, and moving characters currently precludes further development of Nordin's environment (*ibid.*, 13:00).

Nordin (2018, 14:20) described the use of reinforcement learning to train AIs. Reinforcement learning is an old technique that uses rewards to teach an agent to achieve goals (Figure 3). The technique assumes that agents will optimize their actions to get higher rewards. When agents act, their environment gives rewards that characterize that action's effectiveness. An agent can also observe the environment to see what its actions have changed.

As an example, Nordin (2018, 15:10) describes the use of reinforcement learning to train an AI to

seek dots on a screen. The screen is a display for a game that uses a blue dot to represent the game's player; a red dot to represent a negative reward; and a green dot for a positive reward. The game drops the blue dot into this environment with no prior knowledge of the environment. After hours of training, an AI can play the game better than any human.

DeepMind trained an AI to play 57 Atari games based on pixel information alone. While the quality of its play varied by game, the AI could play all the games well (ibid., 16:10).

Due to technological advances since the release of the tested Atari games, AI are now tasked with playing complicated 3D games. These games' additional complexity makes them poorly suited for the style of training that DeepMind used to play Atari games. As an example, in 2017, an FPS (First Person Shooter) game was tested using the same techniques as DeepMind (ibid., 17:00). The AIs so trained, according to Norton, proved to be "pretty stupid" (ibid., 17:29), due to artificial neural networking architecture limitations. As such, they were ineffective.

The FPS AI was also limited to doing one action at a time, which rendered it unsuitable for playing most modern games. Most modern games require users to do multiple, concurrent actions. DeepMind tried to solve this problem in their Atari games by treating concurrent actions as single action multi-actions, like pressing forward and the button at the same time (ibid., 18:42). The number of combinations this can require increases exponentially based on the number of inputs. For example, a PS5 controller has 20 buttons, for a total of $O(2^{20})$ combinations. Using naïve reinforcement learning to support multi-button actions would result in button mashing: around half of the buttons would typically be pressed at a time. Training would take too long to complete (ibid., 19:32). To accelerate reinforcement learning, Nordin (2018, 19:55) combined it with an initial period of imitation learning, which was phased out over time. Having the reinforcement learning initially imitate an expert dramatically reduced button mashing. This allowed for multi-actions to be taken without having to implement single action multi-actions.

As a test of 3D reinforcement learning with multi-actions, Nordin (2018, 20:37) created a simple FPS environment. The environment was kept simple to make learning feasible; a more realistic learning

environment would have dramatically increased the training's difficulty. The test pitted an agent against 10 classical, hand-coded agents. The agent being trained was required to protect an objective area in an environment with buildings, ammunition, and health drops. The agent was provided with 12 actions to choose from, based on a 128x128 pixel first-person view with a short-range sound radar.

Using reinforcement learning with a small amount of imitation learning, the agent learned multiple behaviors that were not explicitly coded: navigation, protection and patrolling, and supplying (ibid., 21:47). The environment did not have Nav Meshes the agent could use to calculate its path; the agent learned how to navigate with only pixel information. After arriving at the objective area, the agent began to protect it while patrolling for enemies. Finally, the agent learned how to collect supplies when needed.

After testing the reinforcement learning with a complex 3D FPS, Nordin (2018, 23:03) generalized the learning by testing the same algorithm with a 3D racing game. After a few minutes of learning, the agent learned how to lap the track. The same FPS agent could perform well in a different environment, after being retrained. This need for retraining showed that the agent was not completely generalizable; it needed to be updated for a different application, even though the process was the same.

After the initial tests, Nordin (2018, 23:32) worked with Dice in their complex game of Battlefield 1. Battlefield 1 is far more complex than the initial FPS environment: it includes more players, classes, gameplay, game modes, and maps. Nordin implemented new training techniques to cope with Battlefield 1's complexity. Battlefield 1's visuals were simplified: ray-traced obstacles and enemies were used for visual input. The agents learned from self-play, which created an additional challenge: when the same agent model was used for both sides, the model learned that "If I don't shoot, I don't get shot." (ibid., 26:48). To fix this, one team used the current model and the other used the last generation model; a small number of classical, agent models were also used on both sides to guarantee that some combat would occur.

Nordin's models made efficient use of their platforms' hardware. During gameplay, the computer's GPU is dedicated towards a game's graphics, making it hard to AIs to access. While training a

model requires a GPU, model inferencing (calculation) can be done efficiently on a CPU alone.

Models generated after training worked, albeit imperfectly (ibid., 28:29, 27:20). When an agent was near the waypoint, could not see enemies, and did not need supplies, it would spin in circles because of having nothing to do. In other cases, agents would show relevant and realistic behavior. With its success, Nordin (2018, 30:26) commented, “This is the first time to my knowledge that anyone has been able to play a first person immersive modern game.”

Nordin described other issues with the agents’ training and behavior (ibid., 30:30). Training models required six days, using eight machines running in parallel, accounting to 15,000 game rounds or 300 days of real-time experience accounting all agents. Developing a system of rewards to elicit desired behaviors was a hard and sometimes unpredictable process: the emergence of surprises and unexpected behavior proved difficult to debug. The resulting agents were black boxes; the authors could not tell exactly how they worked.

Nordin (2018, 31:35) believes that learning algorithms, for now, will only be used to control portions of AI behavior. As such, deep reinforcement learning should integrate with classical AI seamlessly.

Deep learning has generated much hype. According to Nordin (2018, 35:05), a record number of talks at GDC 18—20 in all—were on machine learning. Most of this hype is centered around artificial general intelligence (AGI): i.e., the claim that machines will become just as good as humans at doing most tasks. Nordin (2018, 35:36) believes that AGI will be realized if technology continues to progress and intelligence is not magic. Currently, Nordin cites the 2040’s as a median expert prediction for when AGI will be possible (ibid., 36:20).

Nordin (2018, 37:00) also believes that some of this hype is unfounded. This includes an unwarranted belief that an AI will solve every problem. Startup companies are now just AI startups, and all features are AI features. Deep learning works in practice, but the theory behind why it works is not known. Network architecture creation and hyper parameter turning is an art, not a science. Theories have not yet been developed that allow practitioners to identify effective architectures instead of using

guesswork to create them. Some experts anticipate the advent of an AI winter where progress stalls and research plateaus. A final concern is opposition from individuals and organizations who believe all AI should be banned (ibid., 38:53).

Deep reinforcement learning (DRL) is difficult and only works well in games (ibid., 39:00). It is hard to guarantee behaviors, break down large problems, sample efficiently, learn without prior knowledge, and not be a “one-trick pony”.

Because behavior is developed through reward shaping, wholly desirable outcomes can be difficult to guarantee. Learning only attempts to find the optimal strategy to solve a problem. This will lead DRL to find exploits and have unexpected behavior; the optimal strategy can be completely different than that was intended (ibid., 40:00).

Hierarchical reinforcement learning is the division of problems into high-level subtasks that, when solved, complete the original task through sequences of lower-level subtasks that are considered as primitive actions of high-level subtasks (ibid., 41:15). While the smaller problems can be simpler to solve, training models to complete each subtask can prove inefficient and costly.

Deep reinforcement learning is hard to use effectively. It takes a long time to train DRL models, due to a lack of knowledge priors (ibid., 41:56): models begin with zero knowledge of the environments into which they’re placed (ibid., 42:24). To improve training efficiency, transfer learning can be used to transfer parts of preexisting models’ knowledge bases to new models. Transfer learning is also hard to implement. One alternative to transfer learning is to allow models to explore and learn their environments before requiring them to solve problems (ibid., 43:40).

Another difficulty of DRL is the need to retrain models when applying them to different problems. DeepMind attempted to solve this difficulty by training a model that plays 30 different games effectively without the need to retain it (ibid., 44:24).

Deep reinforcement learning, though difficult to manage, has value. Nordin (2018, 48:00) cited four reasons to use deep learning. Within the last 30 years, Nordin (2018, 48:30) has seen the largest boost in computer capability he has seen, allowing for the use of more complex models. Deep learning

allows computers to solve new classes of problems, such as computer vision or pose estimation (ibid., 48:48). Learning methods can quickly surpass years of engineered software; even with lots of effort, deep learning can easily surpass human software engineering. Finally, many current problems are still unsolved; many future problems have yet to be defined, and the field is still more of an art than a science. As such, field of deep learning has ample potential to find and solve problems.

2.5 PPO Algorithm

Common reinforcement learning algorithms with neural network approximators have multiple disadvantages. Deep Q-Learning will fail on simple problems; policy gradient methods have poor data efficiency and problem robustness, and trust region policy gradients and natural policy gradients are complicated and incompatible with noisy architectures (Schulman, Wolski, Dhariwal, Radford, & Klimov, 2017).

Schulman et al. introduce a new family of policy optimization algorithms that is scalable, data efficient, and robust. A new objective function is introduced that clips the probability ratios to create a pessimistic lower bound. A policy is optimized by sampling from the policy and optimizing the sample. The new family of algorithms was tested against other common reinforcement learning algorithms and was found to perform better in continuous action spaces.

Policy gradient algorithms compute an estimate of the policy gradient and use it in a stochastic gradient ascent algorithm. Figure 4 shows the common gradient estimator (ibid.). The estimator calculates the expected empirical average of a finite batch of samples from an algorithm that alternates between sampling and optimization. If gradient differentiation is calculated using a program, the estimator is created by differentiating the objective function in Figure 5 (ibid.). Performing multiple optimization steps on the loss L^{PG} using the same sample trajectory will lead to destructive policy updates. The updates will be large and are not empirically justified.

In Trust Region Policy Optimization (TRPO), the objective function is maximized subject to a constraint on the policy's update size. Figure 6 shows the TRPO objective function (ibid.). KL is the

Kullback-Leibler divergence in the policy’s state before and after the update. In theory, TRPO should use a penalty for some value of β instead of a constraint for the unconstrained optimization function in Figure 7 (ibid.). In practice, TRPO uses a hard constraint instead of a penalty due to the difficulty in finding a value for β . Schulman et al. found that choosing a fixed coefficient for β and optimizing the function in Figure 7 failed to yield monotonic improvements in the policies being optimized.

TRPO maximizes the surrogate objective function in Figure 8 (ibid). CPI stands for the conservative policy iteration. Maximization of L^{CPI} would create an extremely large policy update. To stop the large policy update, changes that move the probability ratio $r_t(\theta)$ from 1 should be penalized.

As an alternative, Schulman et al. introduce the clipped surrogate object function found in Figure 9 (ibid.). The first term in the minimization is L^{CPI} . The second term clips the probability ratio, penalizing moving r_T from the interval $[1 - \epsilon, 1 + \epsilon]$. Taking the smaller of the unclipped objective and the clipped objective creates a lower, pessimistic bound on the unclipped objective. This causes the probability ratio to ignore changes when the advantages are positive and track them when the advantages are negative. Positive advantages signify that the objective function is improving while negative advantages signify that the objective function is degrading.

Techniques that compute a variance-reduced advantage-function estimator use state-value functions $V(s)$. Neural network architectures that share the parameters between the policy and value function must combine the policy surrogate and the value function error. An entropy term can be added to ensure exploration of the state space. Combining these terms creates the objective function found in Figure 10 (ibid.). S is the entropy bonus for the policy at timestep t and L_t^{VF} is the squared-error loss.

Schulman et al. found that running a policy for a trajectory horizon of T timesteps and collecting samples for an update is good for recurrent neural networks. An advantage estimator that only looks for T timesteps is shown in Figure 11 (ibid.). Figure 12 shows a generalized version that resolves into Figure 11 when $\lambda = 1$. Schulman et al. proposes a version of the proximal policy optimization (PPO) that uses a fixed-length trajectory in Figure 13. This algorithm is based on an actor-critic style model.

Schulman et al.’s PPO algorithm outperforms most other effective methods for continuous problems. This included TRPO, cross-entropy method (CEM0, vanilla policy gradient with adaptive step size, advantage actor critic (A2C), A2C with trust region, and A3C.

3 Methodology

3.1 Overview

To demonstrate that game engines can use deep RL techniques, a multi-threaded game engine was created that incorporates a multi-threaded AI component into its architecture. This architecture features an API that can accommodate multiple types of deep reinforcement learning algorithms. Given this study’s focus on proof of concept, only one such algorithm was implemented.

The game engine and its supporting AI component were tested using a simple game that was created for this study. As the game’s AI was trained, its progress was tracked to determine if the architecture was properly implemented and that the games’ AI model improves.

3.2 Experimental Framework

3.2.1 The game-playing framework

3.2.1.1 *GRAVEngine*

Like contemporary game engines, the study’s game engine, GRAVEngine, was designed for multi-core processing. One of the engine’s two core systems manages jobs, including multi-threading. The other supports rendering. The engine’s other subsystems include a file system, an object importing system, an event system, a mathematics library, and general utilities.

The rendering system used OpenGL (Khronos Group, 2021) as its default rendering API, along with GLFW (GLFW, 2021), which supports the OpenGL-based rendering of contexts, window creation, and user input. Dear ImGui, a user interface rendering library, was used to render user interface systems (ocornut, 2021).

The engine’s mathematics library, glm (OpenGL Mathematics, 2021), uses a syntax like’s OpenGL, which was one of the key reasons for its use.

3.2.1.2 *AI Architecture*

The game engine supports the use of multithreading to implement multiple asynchronous agents, whose observations would be synchronized with a single environment manager. The manager would oversee the calls for updating agents, training, and updating models.

Agent training was controlled with trainers. When an agent was created, a trainer was created if a trainer for an agent's model and training algorithm did not exist. Multiple agents could share the same trainer if the agent's used the same model and training algorithm. A single trainer controller contained and updated all trainers, and the trainer controller was updated by the environment manager.

Agents interact with the manager to pass their recorded observations for training. This is done because agents are updated in an asynchronous manner. An agent's observations need to be recorded and stored for when the training occurs.

3.2.1.3 *The game*

A game of snake was created as a test game. The testing environment consisted of a two-dimensional array of agents, each with its own playable area. Each playable area consisted of a snake agent, fruit, and the walls of the area. Each snake agent was created with a direction sensor, a head sensor, a body size sensor, a fruit sensor, and a wall sensor. Each agent had a single actuator that controlled all of that agent's behavior.

At the start of each episode, a snake agent was repositioned to the middle of its playable area; its body reset to a single square; and the fruit placed at a random location in the playable area.

3.2.1.3.1 *Agent Sensors*

An agent's direction sensor reported the direction that an agent was moving in, which included the four egocentric coordinates of up, down, left, and right.

An agent's head sensor reported a snake agent's current head position relative to its playable area. Along with an agent's current head position, the head sensor reported the agent's previous head position. This sensor, when used in conjunction with the direction sensor, would allow an agent to know how it is

moving.

An agent's body size sensor was a wrapper around an agent's body size. It would report an agent's current body size to give the agent knowledge on its body.

An agent's fruit sensor reported the fruit's position, relative to snake agent's current and previous head positions. This would allow an agent to know which direction it needs to move to eat the fruit.

The wall sensor reported either the closest wall or body part compared to a snake agent's current head position. Because the length and position of all a snake's body parts are variable in length, they could not be directly reported for training. Instead, only the closet object was reported. This still allows an agent to sense its position in the environment, but it is not a perfect information system.

3.2.1.3.2 Agent Actuators

A snake agent's actuator controlled all of the agent's physical logic. When an agent asked its brain for a decision and action, the actuator returned a buffer with a single value. This value dictated the next direction in which the agent should move; based on the new movement direction, special logic would be used to control the agent's behavior.

Before checking anything, an extremely small negative reward was given to an agent. Giving this reward incentivized an agent to take any action that yielded a positive reward as quickly as possible.

If an agent's current head position was closer to the fruit than its previous head position, the agent was given a small reward, equal to $1 / (\text{agent's wall width} + \text{wall height})$. Giving this positive reward incentivized an agent to move toward its fruit.

After moving, an agent checked if its new head position was intersecting with the fruit. If so, like other snake games, the snake's body grew by one segment and another fruit was placed in a new random position. A positive reward was then given to the agent to influence it to move towards the fruit. Finally, an agent checked to see if its new head position was intersecting with either of its playable area's four walls or any of its body parts. When a collision was detected, the agent's training episode was ended, and a negative reward was assigned to influence the agent to avoid future collisions.

3.3 Training

PyTorch, an open-source machine learning library, was used for model creation and inferencing. PyTorch allowed for the gradient descent necessary for training to be done automatically and for neural networks to be created easily. Using PyTorch eliminated the need to implement the inference engine, which was beyond the scope of the experiment. A graphics card was also optional for speeding up model inferencing because of its built hardware support.

3.4 Testing

3.4.1 Criteria

Similar to Szitla et al., the architecture's effectiveness was evaluated based on its efficiency and consistency: i.e., how quickly it could be trained and how well it performed once its trained. A successful AI will increase its cumulative reward as the training period elapses while minimizing the loss. The reward should increase while the loss decreases.

Training could result in an AI having a suboptimal behavior. Although unwanted, the creation of a suboptimal AI would show that AIs can be trained natively. The reward would still increase over time while the loss decreases, but a local minimum would be found instead of the desired behavior.

4 Results

4.1 GRAVEngine

4.1.1 Fiber Job System

A fiber-based job system was developed for GRAVEngine to handle all multi-threading functionality of the game engine. The job system consisted of the job manager, threads, fibers, and jobs (Figure 14). Threads were kernel-managed threads, and fibers were process-managed. Job execution was handled inside of a fiber's execution context, and a fiber's execution context was handled inside of threads. Jobs run inside fibers, which are the execution context of a thread (Figure 15).

All synchronization was handled through the job manager, which handled all the job system's

public API. On the job manager's initialization, a thread was created for each core. Each thread was allocated its own core, assuring that any job that ran on that thread's current fiber would run on its core. Along with thread creation, a pool of fibers was also created that all the job manager's threads could pull from. Free fibers were placed inside of the fiber pool for later use.

Whenever a thread was created, it ran a function that first identified then switched to a currently free fiber for an execution context. Fibers, like threads, have their own execution contexts and stacks. Unlike threads, inter-fiber context switches are fast, due to in-process scheduling. Fibers repeatedly checked the job manager, repeatedly acquiring, then running, available jobs.

To run a job in parallel to other jobs, a user created a job declaration. This declaration contained a pointer to a function to run, a pointer to the function's arguments, and the job's priority. The declaration was then passed to the job manager, which added the declaration to one of the job manager's four queues, each corresponding to a specific job priority: critical, high, medium, and low.

To synchronize jobs, users used job counters, which were the job system's lone synchronization tool. For each job that it was given, the job manager returned a job counter: an atomic counter that was incremented as associated jobs were added to the manager's job list and decremented as associated jobs finished execution. Jobs are synchronized by waiting for a job's counter to reach a specified target.

When any synchronization is done, the current fiber is placed into a waiting queue along with the job and a new fiber is pulled from the fiber pool to start execution on the thread. This policy assures that a thread is always executing a fiber's execution context. If a thread continually waits for a counter, the core is not being fully utilized. As such, switching to a different fiber to run other jobs allows for higher efficiency.

When a job's counter reached its specified target, any fibers associated with the counter that were inside the waiting queue could continue execution. The current fiber switched to the waiting fiber, essentially swapping the program counter to the fiber's job.

4.1.2 Rendering System

A simple rendering system was created for GRAVEngine. The public rendering API for GRAVEngine consisted of shaders, textures, cameras, and renderers. To enable support for multiple rendering APIs other than OpenGL, shaders and textures were abstracted with OpenGL specific implementations.

Two cameras were created: a perspective camera for 3D visuals and an orthographic camera for 2D visuals. Two different renderers were created: a 2D renderer for thread-safe interaction with the job system, and a 3D renderer for use with a single threaded application. While the 3D render, like the 2D renderer, could be redesigned for thread safety, this work was beyond the scope of this research.

To render a renderable object, a user would submit that object to a renderer. The 2D renderer could only render squares; anything else was beyond the scope of this research. The 3D renderer could render any mesh that was generated from the importing utility.

4.1.3 Layer System

The layer system was implemented to update user code. A user would inherit from the abstract layer and implement the `OnUpdate` and `OnImGui` functionality. `OnUpdate` and `OnImGui` would be called by the application to update user code. Layers were added to the application's update list by pushing layers to the application's layer or overlay stack. Pushing to the layer stack appended the layer to the update list; pushing to the overlay stack prepended the layer to this list. When the application updated, the update list would then update every layer that has been added to the layer stack. Dear ImGui interfaces were updated during the `OnImGui` functionality.

4.1.4 Event System

An event system was created to allow users and GRAVEngine to have lists of arbitrary functions be runnable at a single function invocation. Users would create events and then add arbitrary actions to be called when that event was invoked. Events were handled through the job system. For each action added to an event, a job would be created. After creating all an event's job declarations, the event added all the

jobs to the job manager for scheduling. After the counter that was returned from adding the event's jobs reached zero, the event would be finished.

4.1.5 Supporting Utilities

Additional utilities that were created to enable the GRAVEngine to function included code for IO operations, time management, locks, memory management, instrumentation, logging, and triangulated mesh importing.

4.2 AI Architecture

4.2.1 Environment Manager

AI in GRAVEngine was handled through the environment manager, which synchronized updates to the environment, agents, and training algorithms (Figure 16). When the environment manager was updated, agents requesting a decision would have a decision made and then perform an action based on the chosen decision. After its update, the environment manager would update the trainer controller and all associated trainers. The environment manager's update was done in this fashion due to the asynchronous nature of agents.

Agents could be updated in multiple different threads, so their training observations were stored temporarily for later updates. Agent observations were stored in their associated trainer. To add observations, agents would access their associated trainer through the trainer controller.

4.2.2 PPO Algorithm

The example reinforcement learning algorithm that was implemented was principal policy optimization. As explained by Schulman et. al., PPO is an easy to implement and efficient algorithm, so that was why it was chosen as the example training algorithm.

As the PPO algorithm uses an actor critic model, an abstract model system was designed (Figure 17). An actor critic model consisted of both a critic and actor module, each of which had an internal deep neural network encoder that was passed to an internal decoder. An actor's decoder consisted of an action layer; an action layer would return a multi-categorical and normal distribution for both discrete and

continuous action spaces. A critic's decoder consisted of a value layer, which would just return a single critic value.

4.3 Game

Snake was successfully implemented and used to create a 5 x 5 grid of playable areas of training agents (Figure 18). Each agent used the same training program for learning; although each agent had a different brain, agents would upload their experiences to the same trainer.

The method of training used in these experiments produced snake agents that exhibited suboptimal behaviors. Agents would move towards the fruit but would not capture it unless it was extremely close to its starting position. An agent would get close to it then oscillate between moving away from the fruit and moving towards the fruit. A positive reward was only given when the agent moved closer towards the fruit or when the fruit was captured; a negative reward was not given for moving away from the fruit. This reward scheme allowed for the agent to easily maximize its reward by just oscillating between moving to and from the fruit.

4.3.1 Statistics Graphs

The Snake agent that was trained increased its cumulative reward as time moved on. As seen in Figure 19, the first iterations of the agent had a cumulative reward that started near zero and climbed towards eight, where it stabilized for most of the training session. Dips in the cumulative reward graph show areas where the agent was discovering new actions and was exploring the environment.

Figures 20 and 21 show the actor loss and critic loss over time for the implemented PPO training algorithm. There are jumps in the actor loss, which correlates decreases in the cumulative loss in Figure 19. The jumps show that the snake agents took bad actions that dramatically decreased the cumulative reward. As seen in Figure 21, the critic loss starts high near 0.25 and quickly approaches 0.

5 Conclusions, Implications, and Recommendations

5.1 Conclusions

It is possible to have an AI architecture natively implemented into a game engine. An AI

architecture was designed and implemented into GRAVEngine, and a test game was successfully created along with an agent that learned a behavior over time. The modular AI architecture could handle multiple different training algorithms, but only PPO was implemented to show proof that an AI architecture could be natively implemented.

Although the trained behavior resulted in a suboptimal behavior, learning was still present and evident (Figure 19). Tailoring the rewards that an agent receives during training is another area of research that was beyond the scope of this experiment, so showing that learning occurred is acceptable proof of a successful AI architecture.

A key weakness of this experiment was that lots of focus was placed in the development of GRAVEngine and not as much as the AI architecture. Multiple supporting tools and utilities were required to have GRAVEngine work, and not as much work was placed into the AI architecture. As such, there are less quality-of-life features that would have been good in the experiment to have more successful learned behaviors.

Although not as much focus was placed on the AI architecture as GRAVEngine, this experiment was a good proof of concept for a natively implemented AI architecture into a game engine. This experiment showed it is possible to have complex AI inside of a game engine.

5.2 Implications

Because it is possible to have more advanced AI inside of a game engine due to a native AI architecture, games can increase the complexity of agents inside of games. Instead of having to design architecture for an AI inside of a game, game designers can use a native AI architecture and focus on the believability and immersion of an AI. As stated by Szita et. al, a more immersive AI can increase player satisfaction of games.

5.3 Recommendations

Creating a game engine to test an AI architecture is a difficult task. Instead of creating a game engine explicitly, an AI architecture should be developed separately from a game engine and then natively

implemented. If an AI architecture is developed and designed separately, a more robust architecture could be created. As such, it is recommended to develop an AI architecture separately and add a native implementation to a game for an easier development cycle.

6 References

- Bakkes, S., Spronck, P., & Herik, J. v. (2009). Rapid and reliable adaptation of video game AI. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(2), 93-104.
doi:<https://doi.ieeecomputersociety.org/10.1109/TCIAIG.2009.2029084>
- GLFW. (2021). Retrieved from GLFW: <https://www.glfw.org/>
- Khronos Group. (2021). *OpenGL*. Retrieved from OpenGL: <https://www.opengl.org/>
- Nordin, M. (2018, March 19-23). *Deep Learning - Beyond the Hype*. GDCVault. Retrieved November 5, 2020, from <https://www.gdcvault.com/play/1025098/Deep-Learning-Beyond-the>
- ocornut. (2021). *Dear ImGui*. Retrieved from GitHub: <https://github.com/ocornut/imgui>
- OpenGL Mathematics*. (2021). Retrieved from glm: <http://glm.g-truc.net/0.9.9/index.html>
- Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy Optimization Algorithms. doi:1707.06347
- Szita, I., Ponsen, M., & Spronck, P. (2009). Effective and diverse adaptive game AI. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1), 16-27.
doi:<https://doi.ieeecomputersociety.org/10.1109/TCIAIG.2009.2018706>
- Tulip, J., Bekkema, J., & Nesbitt, K. (2006, December 4). Multi-Threaded Game Engine Design. *Proceedings of the 3rd Australasian conference on Interactive entertainment (IE '06)*, 9-14.
doi:10.5555/1231894.1231896