Eksplorium *p*-ISSN 0854-1418
Volume 46 No. 1, January 2025: 1745–1754 *e*-ISSN 2503-426X

Adaptive Multiple AI by Leveraging Reinforcement Learning

Ravisankar Popuri, Gunamani Jena, Shubhashish Jena, Chandra Mouli VSA, P Devabalan

Research scholar, CMJ University, Jorabat Meghalaya, Roland Institute of Technology, BPUT

ravi.shinsou@gmail.com, drgjena@gmail.com

Article Received: 10 Dec 2024, Revised: 16 Jan 2025, Accepted: 26 Jan 2025

Abstract: This paper is about utilizing reinforcement learning (RL) for the development of adaptive AI systems utilizing Unreal Engine 5. This paper focuses on how reinforcement learning can be used to create more efficient artificial intelligence or Non- Playable Characters (NPCs) that dynamically adjust their behavior based on any given in-game context. This provides not only more immersive and challenging gameplay, but also an easy way for AI to mimic human behavior. The research has a detailed analysis of integrating RL algorithm with unreal engine's blueprint and C++ systems. This provides a framework for developers to implement adaptive AI that can evolve over time. The overall performance of this AI system is evaluated through various gameplay scenarios, that demonstrate significant improvements in player engagement. In games now a days, multiplayer experiences continue to grow in complexity and popularity. Game developers are seeking more adaptive and strategically efficient Non-Playable Characters (NPCs) to mimic human-like behavior. This paper showcases an efficient AI system where two opposing teams of agents which are trained using Proximal Policy Optimization (PPO) engage in open-field combat within a game level under Unreal Engine. Each agent aims to maximize individual and collective team rewards by eliminating opponent team members. Through iterative training and carefully designed reward structures, this AI system learns robust combat strategies which leads to an efficient combat AI that exhibits sophisticated coordination behaviors. Results indicate that PPO-driven agents can adapt to various geometrical game levels more effectively compared to conventional scripted NPCs or enemy AI. This paper demonstrates the potential of deep reinforcement learning for complex multi-agent in diverse game environments.

Keywords: create Non- AI Behavior, Playable Characters (NPCs), Proximal Policy Optimization (PPO), RNN-based PPO, Simultaneous multi-RL agents.

I. INTRODUCTION

Game AI has traditionally relied on hard coded decision trees or finite-state machines to drive / control Non-Playable Characters (NPCs). These methods can produce engaging experiences, but they often fail to adapt efficiently to diverse game environments. Reinforcement Learning (RL) presents a proficient alternative by combining deep neural networks. AI agents are trained through repeated interactions in game level environment, RL has a huge potential to implement AI behavior strategies that outperform scripted methods in order to dynamically adapt to changing conditions. This study focuses on designing a multi-agent AI system with two teams, comprised of autonomous agents. These agents are trained using a specific RL algorithm called Proximal Policy Optimization (PPO). All AI agents will face off in a simple open world map created in Unreal Engine 5. Each agent's aim is to shoot its opponent team's agents to contribute to the team's overall success. PPO is mainly selected for this purpose due to its balance between sample efficiency and stable policy updates, attributes. These factors make PPO a strong candidate for multi-agent tasks with continuous or semi-continuous action spaces.

I.I. Contribution

Multi-agent PPO powered AI agents operate as two opposing teams. An in-depth and hands-on implementation of PPO algorithm that accommodates advanced dynamics and complex 3D navigation. Quantitative and qualitative evaluations highlight how RL-driven agents adapt to evolving circumstances. RL agents demonstrate an efficient coordinated team strategy which mimics multiplayer behavior although being single player. Optimizable and extendable framework that can evolve for future needs.

II. LITERATURE REVIEW

PPO was introduced by Schulman et al. (2017), is a policy gradient method designed to stabilize the training of neural-network-based RL agents. PPO modifies the objective function in such a way that large, destabilizing policy updates are penalized further ensuring smoother convergence. In single-agent settings, PPO has been applied successfully to continuous control tasks, arcade-style video games, and simulated robotics. In multi-agent

scenarios, it retains its stability advantages but requires careful design of shared reward structures and training loops.

Traditional RL algorithms like Q-learning approximate the value of actions (Q-values) and select actions that maximize expected returns. In contrast, policy gradient methods directly learn the probability distribution of actions given states, typically represented by a parameterized function $\pi\theta(a \mid s)$, where θ are the parameters (e.g., weights in a neural network). The agent's objective is to maximize the expected return, often formalized as:

$J(\theta) = E\tau \sim \pi\theta\ [t = 0 \sum \gamma tRt]$,

where τ is a trajectory (sequence of states, actions, and rewards), T is the episode length,

γ is the discount factor, and Rt is the reward at time t.

One challenge with policy gradient methods is the risk of destructive large updates, which can push the policy far from previously successful behaviors and destabilize training. PPO addresses this by introducing a clipped objective that penalizes deviations from the old policy beyond a certain threshold. This ensures iterative, proximal updates, striking a balance between exploring new strategies and consolidating previous gains.

Steps involved in PPO algorithm:

- *Data Collection:* The current policy $\pi\theta$ is used to run multiple episodes or mini episodes, generating a batch of trajectories.
- *Advantage Estimation:* An advantage function At is calculated for each timestep t, indicating how much better taking action at was compared to the agent's baseline expectation.
- *Policy Update:* The ratio $rt(\theta)$ between the new policy and old policy probabilities for each action is calculated: $rt(\theta) = \pi\theta\ old\ (at \mid st)\pi\theta(at \mid st)$.
- Then, the clipped objective:
  $LCLIP(\theta) = Et[\min(rt(\theta)At, clip(rt(\theta), 1 - \epsilon, 1 + \epsilon)At)]$, is optimized, where ϵ (commonly 0.1 or 0.2) controls the clipping range.
- *Value Function Update:* PPO often also learns a value function $V\phi(st)$ to aid in advantage estimation, updating ϕ via a mean-squared error (MSE) loss.
- *Repeat*: New experiences are collected with the updated policy, and the steps repeat until convergence or for a set number of iterations.

By constraining each policy update, PPO yields a learning process characterized by greater stability and fewer catastrophic policy shifts. This characteristic is especially advantageous in multi-agent scenarios where each agent's policy update can affect the entire environment dynamics.

[Mastering the game of Go with deep neural networks and tree search] demonstrates a unique approach to playing board game Go at very high superhuman level by combining deep neural networks with a Monte Carlo Tree Search (MCTS) framework. A policy network is trained to predict human strategies. while a value network evaluates board positions. The system adapts its strategy through self-play. This sets a new precedent for the application of reinforcement learning in game environments.

[Grandmaster level in StarCraft II using multi-agent reinforcement learning] explores a multi-agent reinforcement learning approach for the real-time strategy game StarCraft II. This utilizes a combination of deep learning architectures, distributed training and centralized value functions to coordinate multiple in-game units. The findings illustrate the scalability of RL methods and their ability to handle the complexity of strategy games.

III. PROPOSED APPROACH

III.I. Game AI Using Unreal Engine

Unreal Engine is a very popular game engine that is widely adopted in both academic research and commercial game development. It has high-fidelity graphics, physics simulations, and flexible scripting environments. Game developers and researchers are using it for studying emergent behaviors in AI-driven agents, pathfinding in

complex terrains, and player adaptation. Unreal engine can be regarded as an IDE where the entire system can be implemented without depending on any other external tools.

In the proposed system two teams governed by reinforced learning agents will be trained to shoot multiple targets by utilizing RL agent interactor, RL agent Trainer and RL agent Manager with a 3D game character. Interactor serves as the hands and eyes of the RL agents. The trainer handles the core learning algorithm, storing data, updating models, and providing the policy decisions. Finally, the Manager orchestrates the entire simulation by instantiating agents.

III.II. Architecture of PPO

The PPO Architecture diagram depicted in Fig.1 showcases the major components of the multi-agent PPO system in Unreal Engine. The diagram illustrates the data flow between the modules involved along with their interaction during training and inference.

Component Explanation

i. **Parameter Server:** Parameter Server is a central coordinator which maintains a centralized repository of the shared policy ($\pi\theta$) and value function networks which serves as the "authority" on which weights each agent should use at any given point of time.

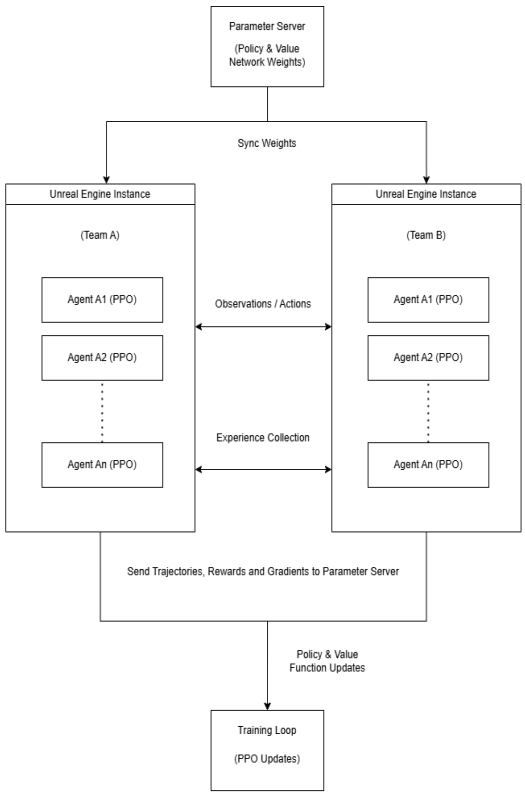
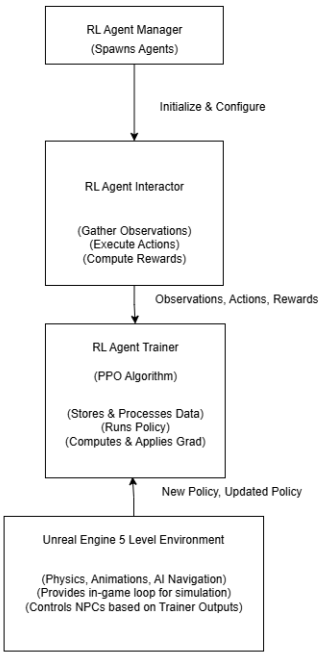


Fig. 1(a) PPO Architecture

Fig. 1(b) Unreal Engine RL Architecture

ii. **Data Flow:** Sync Weights periodically distributes the latest network parameters (θ for the policy network and ϕ for the value network) to each Team instance.

iii. **Policy & Value Function Updates:** After receiving experience data from the agents like states, actions, rewards and next states, it performs PPO training steps like policy gradient updates, clipping, advantage calculation. Then the updates weights are synchronized back to the agents.



PPO Agents

Local Decision Making: Every agent A1, B1...An, Bn observes the game states like health, positions, line of sight to decide on actions like movement, firing and reposition in accordance with the opponents on the field using the local copy of the policy network.

i. **Experience Collection:** At each timestep the agent logs a tuple $(st, at, rt, st + 1)$. These are assembled into trajectories and then sent back to the Parameter Server for training.
ii. **Actions & Observations:** Agents receive data like positions of enemies, team status from the Unreal Engine environment and output discrete or continuous actions like move forward, fire weapon.
iii. **Experience Collection:** Agents gather rewards like kill bonuses like added score, death penalty and store them in memory until they reach a designated batch size of the end of an episode.
iv. **Training Loop:** When an episode or mini- batch is completed (e.g. iterations / steps), the local agents send their collected trajectories to the parameter server.
v. **Policy & Value Network Optimization:** The server applies PPO updates to compute the advantage function using the value network, calculate the clipped objective to ensure stable updates and finally adjust network parameters θ (policy) and ϕ (value function) via gradient descent.
vi. **Updated Policy Broadcast:** After each update, mew network weights are broadcast to the agents so that they continue acting with the latest policy.

Communication Protocol

Will be realized with a client-server model over sockets or Remote Procedural Call (RPC), ensuring data from multiple Unreal Engine instances flows to a single Parameter Server.

Synchronization Frequency is determined by the training protocol where agents might sync after every episode, every N steps or after a fixed time interval.

Implementation in Unreal Engine

Environment Design

An open-field map is designed in Unreal Engine, populated by two teams. Each team is composed of multiple RL agents. Key considerations include:

- **Terrain Layout:** Mostly flat with some scattered obstacles or cover points, ensuring line-of-sight constraints and possibilities for strategic maneuvering.
- **Spawn Points:** Both teams spawn at opposite ends, forcing them to traverse the field to encounter enemies.
- **Collision and Damage Systems:** Leveraging Unreal Engine's physics and collision detection for bullet projectiles, hit registration, and health calculations.

State, Action, and Reward Spaces

State Representation

Each agent's observation is encoded in vectors that capture:

- **Self-Information:** Current health, ammunition count, cooldown timers, movement speed.
- **Teammate Data:** Relative positions, health status of nearby teammates.
- **Enemy Data:** Approximate direction, distance, and known positions of visible enemies.
- **Environment Context:** Spatial position, presence of cover, and navigability of terrain. This data is periodically sampled and fed into a neural network representing $\pi\theta$.

Action Space

Actions are discretized into commands such as:

- **Movement**: Move forward, backward, strafe left/right, sprint. Rotation: Look left/right, adjust pitch for aiming.
- **Combat**: Fire weapon, reload, relocate.
- In some variants, these actions can be combined or subdivided for finer control, potentially moving toward a continuous action space if required.

Reward Structure

We designed a reward function to promote strategic team combat:

- **Kill Reward (+r1):** Positive reward for successfully eliminating an opponent. Team Support Bonus (+r2): Additional reward for assisting a teammate in a kill (e.g., damage contribution). Death Penalty (-r3): Negative reward if the agent is killed. Friendly Fire Penalty (-r4): Discourages shooting teammates.
- **Survival Incentive (+r5):** Minor positive reward for staying alive each timestep to emphasize survival as a contributing factor to victory. The magnitudes of these rewards (r1, r2, r3, r4, r5) are crucial to shaping the emergent policy and are subject to iterative tuning.

IV. RESULTS

Quantitative Analysis

Team Victory Rate dictates the percentage of matches won by each team over 100 evaluation games. Over training time, both teams improved, but the team with better hyperparameter tuning or more stable training outperformed the other.

Mean Episode Return dictates the average cumulative reward per agent per episode. This metric helps gauge individual agent effectiveness.

Kill/Death Ratio demonstrates the offensive prowess and survivability of the agents.

Coordination Score is a custom metric based on proximity and timing of attacks, measuring how often agents worked in unison. Over approximately 3000 training iterations, the PPO agents showed a substantial increase in coordinated behaviors. Many matches culminated in agents successfully flanking enemy positions, suggesting a learned tactical approach rather than random engagements.

IQualitative Observations

Adaptive Positioning where agents began using the terrain strategically, seeking cover in small dips in the ground or behind scattered props. Over time, the friendly-fire penalty effectively diminished instances of teammates firing in each other's line-of-sight.

In post-evaluation tests, the PPO agents demonstrated a challenging skill ceiling, sometimes catching opponents off-guard with concentrated fire.

Result and Discussion

Fig.2 showcases all agents for both teams getting instantiated / spawned by the RL Agent Manager.

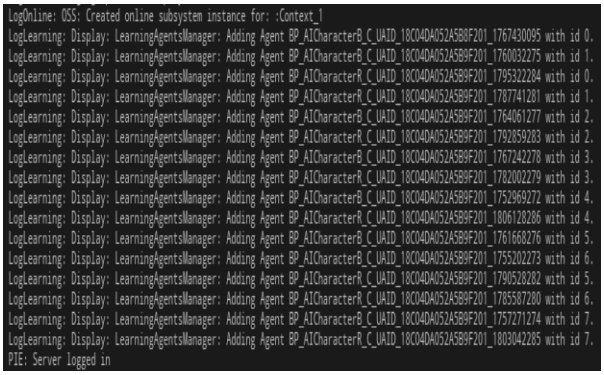


Fig.2 Instance starting

Fig.3 and Fig.4 Shows the initialization of the training process.

Fig.3 Training Process initialization

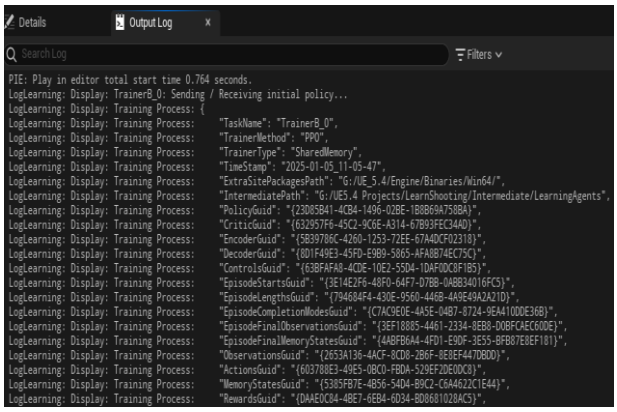


Fig.4 Initialization of policy, episode, critical, encoder, decoder and control guids

Fig.5, 6 & 7 showcases the starting iteration, second iteration and 480th iteration of the training process.

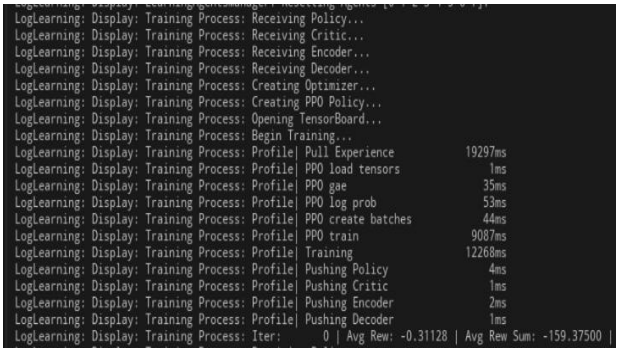


Fig.5 First Iteration

Fig.6 Second Iteration

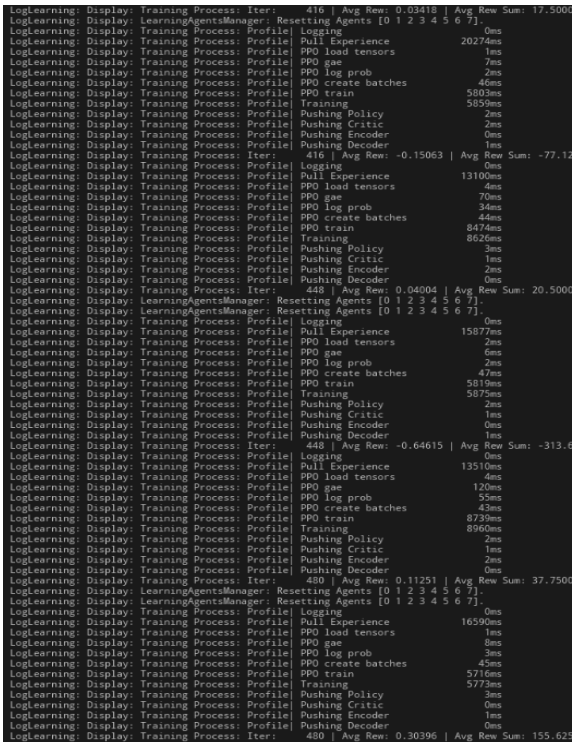


Fig.7 480th iteration

Fig. 8 (a, b, c & d) showcases the in-game scoreboard for both the teams. This clearly showcases how both teams have won the games throughout the training process instead of only one team getting the upper hand in the score.

p-ISSN 0854-1418
e-ISSN 2503-426X

Fig.8 (a) Team A and Team B with scoreboard

Fig. 8 (b) Team B with more points

Fig. 8 (c) Team A with more points

Fig. 8 (d) Team A winning

Fig.9 Asset Validation after training process

```
LogSavePackage: Moving output files for package: /Game/Learn/Red/DA_NeuralNetworkPoli
LogSavePackage: Moving '../../../../UE5.4 Projects/LearnShooting/Saved/DA_NeuralNetwo
LogFileHelpers: InternalPromptForCheckoutAndSave took 134.384 ms
LogContentValidation: Display: Starting to validate 8 assets
LogContentValidation: Enabled validators:
LogContentValidation:     /Script/DataValidation.DirtyFilesChangelistValidator
LogContentValidation:     /Script/DataValidation.EditorValidator_Localization
LogContentValidation:     /Script/DataValidation.WorldPartitionChangelistValidator
LogContentValidation: Display: Validating asset /Game/Learn/Blue/DA_NeuralNetworkEnco
AssetCheck: /Game/Learn/Blue/DA_NeuralNetworkEncoderB Validating asset
LogContentValidation: Display: Validating asset /Game/Learn/Blue/DA_NeuralNetworkDeco
AssetCheck: /Game/Learn/Blue/DA_NeuralNetworkDecoderB Validating asset
LogContentValidation: Display: Validating asset /Game/Learn/Blue/DA_NeuralNetworkCrit
AssetCheck: /Game/Learn/Blue/DA_NeuralNetworkCriticB Validating asset
LogContentValidation: Display: Validating asset /Game/Learn/Blue/DA_NeuralNetworkPoli
AssetCheck: /Game/Learn/Blue/DA_NeuralNetworkPolicyB Validating asset
LogContentValidation: Display: Validating asset /Game/Learn/Red/DA_NeuralNetworkEncod
AssetCheck: /Game/Learn/Red/DA_NeuralNetworkEncoderR Validating asset
LogContentValidation: Display: Validating asset /Game/Learn/Red/DA_NeuralNetworkDecod
AssetCheck: /Game/Learn/Red/DA_NeuralNetworkDecoderR Validating asset
LogContentValidation: Display: Validating asset /Game/Learn/Red/DA_NeuralNetworkCriti
AssetCheck: /Game/Learn/Red/DA_NeuralNetworkCriticR Validating asset
LogContentValidation: Display: Validating asset /Game/Learn/Red/DA_NeuralNetworkPolic
AssetCheck: /Game/Learn/Red/DA_NeuralNetworkPolicyR Validating asset
```

Fig.9 depicts the asset validation at the end of the training process.

Discussion

Key Benefits of PPO Architecture

The RL Agent Manager may periodically instruct the system to reset episodes, change the conditions for domain randomization. It collects global stats and manages training schedules by stopping once performance targets are met.

i. **Scalability:** Multiple Unreal Engine instances can be distributed across several machines, each running a subset of agents which will effectively parallelize collected experience.
ii. **Stable Training:** PPO's clipped updates in a centralized server will prevent catastrophic forgetting thus ensuring more robust learning in a competitive multi-agent setting.
iii. **Modularity:** The separation between Unreal Engine environment and the Parameter Server learning provides flexible substitution of RL algorithm and parameter tuning scripts.
iv. **Sample Efficiency:** Collected trajectories can be reused multiple times within a single update epoch which makes PPO effectively extract information from each sample thus resulting in accelerating learning in high-dimensional multi-agent environments. Each agent independently uses the learned policy which results in fostering scalability and robustness to partial observability or network latency.

Limitations

When running multiple Unreal Engine instances, each simulating multiple PPO agents, is a huge resource intensive process. Choices such as the clipping parameter ϵ, learning rate, and reward weights can substantially affect training outcomes due to hyper parameter sensitivity. Agents cannot always have a comprehensive view of the battlefield due to lack of deeper research into recurrence (RNN-based PPO) or attention mechanisms.

V. CONCLUSION

This paper about Adaptive multiple AI gave a hands-on approach on how Proximal Policy Optimization (PPO) can powerfully drive a multi-agent combat system in Unreal Engine, with two opposing teams striving to eliminate each other. By carefully designing state representations, action spaces, and reward structures, agents can learn coordinated behaviors and complex tactics. Experimental results show that PPO-trained agents offer a more engaging and dynamic challenge compared to traditional rule-based hard coded NPCs, indicating the potential for broader applications of RL in team-based adversarial game environments. Although resource demands and hyperparameter tuning remain substantial challenges, ongoing work in hierarchical methods and advanced sensor models may further elevate the sophistication of AI combat behavior. As the field evolves, RL-driven multi-agent systems will become more prevalent in both research and commercial gaming which offers players rich, adaptive experiences that clearly mimic real human behavior.

Future Directions

Splitting the combat tasks into higher-level strategies and lower-level combat skills will boost learning efficiency. New approaches that allow policies to generalize quickly to new map layouts or novel enemy strategies. Introducing new objectives such as resource collection or control point captures to blend cooperative and competitive dynamics. Using Leveraging advanced sensor models to identify and track enemies over long distances or through obstructed lines of sight.

References

[1] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov (2017). “Proximal Policy Optimization Algorithms.”
[2] David Silver, Aja Huang., (2016). “Mastering the game of Go with deep neural networks and tree search.”
[3] O. Vinyals, T. Ewalds. (2019). “Grandmaster level in StarCraft II using multi-agent reinforcement learning.”
[4] Remi Niel “Hierarchical Reinforcement Learning for Real-Time Strategy Games”
[5] E. Lockhart and Y. K Kwok, (2021). “Resource Management Using Proximal Policy Optimization in Tower Defense Games.”
[6] L. Zhang, A. A. Aziz, and M. G. Rashed, (2022). “Multi-Agent Deep Reinforcement Learning for Dynamic Team Tactics in MOBA Games.”
[7] K. Kim, S. Im, J. Park, (2022) “Applying Resinforcement Learning in FPS Environments for Realistic Non-Player Characters.”
[8] K. Shao, Z. Tang (2019). “A Survey of Deep Reinforcement Learning in Video Games.”
[9] D. Kreffer, (2022). “Reinforcement learning for non-player characters in the video game industry.” Epic Games. Unreal Engine 5