GENERALIZABLE ARTIFICIAL INTELLIGENCE AGENT IN GAME DEVELOPMENT

MR. TANATANEE PONARK
MR. INTOUCH YUSOH

A PROJECT SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR
THE DEGREE OF BACHELOR OF ENGINEERING (COMPUTER ENGINEERING)
FACULTY OF ENGINEERING
KING MONGKUT'S UNIVERSITY OF TECHNOLOGY THONBURI
2024

Generalizable Artificial Intelligence Agent in Game Development

Mr. Tanatanee Ponark
Mr. Intouch Yusoh

A Project Submitted in Partial Fulfillment
of the Requirements for
the Degree of Bachelor of Engineering (Computer Engineering)
Faculty of Engineering
King Mongkut's University of Technology Thonburi
2024

Project Committee

..... (Assoc.Prof. Natasha Dejdumrong, D.Tech.Sci.)	Project Advisor
..... (Asst.Prof. Nuttanart Muansuwan, Ph.D.)	Committee Member
..... (Asst.Prof. Phond Phunchongharn, Ph.D.)	Committee Member
..... (Jaturon Harnsomburana, Ph.D.)	Committee Member

Project Title	Generalizable Artificial Intelligence Agent in Game Development
Credits	3
Member(s)	Mr. Tanatanee Ponark
	Mr. Intouch Yusoh
Project Advisor	Assoc.Prof. Natasha Dejdumrong, D.Tech.Sci.
Program	Bachelor of Engineering
Field of Study	Computer Engineering
Department	Computer Engineering
Faculty	Engineering
Academic Year	2024

Abstract

In recent years, the application of artificial intelligence (AI) in game development has garnered significant attention, particularly in enhancing gameplay and automating testing processes. However, most existing AI systems are tailored for specific games or tasks, limiting their adaptability and reusability across different projects. This study proposes the development of an adaptive AI agent for 2D platformer games using reinforcement learning. The agent is trained within a Unity-based environment to learn essential player behaviors, such as movement, jumping, and obstacle avoidance. Leveraging the Proximal Policy Optimization (PPO) algorithm, the system is designed with modular components to promote flexibility, scalability, and ease of integration. By separating action execution from decision-making, the architecture facilitates the extension of the agent's capabilities to support additional actions or mechanics. The trained AI model is packaged as a reusable Unity asset, intended for deployment in similar 2D games either as an intelligent NPC or an automated game tester. This approach aims to reduce development time and technical barriers for indie developers seeking to integrate intelligent behavior into their games. The results demonstrate the potential of reinforcement learning in constructing adaptable game agents and contribute toward making AI tools more accessible in the domain of game development.

Keywords: Game Development / Artificial Intelligence / Enemy AI / Reusable Assets / Non-Player Characters / Adaptive AI / Reinforcement Learning / Procedural Learning / Game AI Training / AI Assets

หัวข้อปริญญานิพนธ์	ปัญญาประดิษฐ์ที่ปรับตัวได้สำหรับการพัฒนาเกม
หน่วยกิต	3
ผู้เขียน	นายธนธานี โพธิ์นาค
	นายอินทัช ยูโซะ
อาจารย์ที่ปรึกษา	รศ.ดร.ณัฐชา เดชดำรง
หลักสูตร	วิศวกรรมศาสตรบัณฑิต
สาขาวิชา	วิศวกรรมคอมพิวเตอร์
ภาควิชา	วิศวกรรมคอมพิวเตอร์
คณะ	วิศวกรรมศาสตร์
ปีการศึกษา	2567

บทคัดย่อ

Thai translation coming soon

คำสำคัญ: Game Development / Artificial Intelligence / Enemy AI / Reusable Assets / Non-Player Characters / Adaptive AI / Reinforcement Learning / Procedural Learning / Game AI Training / AI Assets

ACKNOWLEDGMENTS

Acknowledge your advisors and thanks your friends here..

CONTENTS

PAGE

LIST OF TABLES

TABLE **PAGE**

LIST OF FIGURES

FIGURE **PAGE**

LIST OF SYMBOLS

SYMBOL		UNIT
α	Test variable	m^2
λ	Interarival rate	jobs/ second
μ	Service rate	jobs/ second

LIST OF TECHNICAL VOCABULARY AND ABBREVATIONS

AI	=	Artificial Intelligence
NPC	=	Non Player Characters
RL	=	Reinforcement Learning

CHAPTER 1 INTRODUCTION

1.1 Background

Platformer games, characterized by side-scrolling environments, obstacle navigation, and real-time player control, have long been a staple of the video game industry. From iconic titles like Super Mario Bros. to modern indie successes like Hollow Knight, 2D platformers remain popular due to their accessible gameplay and design flexibility. However, designing intelligent in-game agents—whether as controllable characters or non-player characters (NPCs)—continues to pose technical challenges, particularly for developers with limited resources.

Artificial intelligence (AI) plays a pivotal role in creating engaging gameplay by simulating intelligent behavior. Traditional AI techniques in games, such as finite state machines or behavior trees, offer predictable and rule-based control systems that are effective but limited in adaptability. These approaches often struggle to scale across complex environments or accommodate emergent gameplay scenarios.

Reinforcement learning (RL), a subfield of machine learning, provides a compelling alternative by enabling agents to learn behaviors through interactions with their environment. Unlike scripted AI, RL agents adapt to different contexts by optimizing their actions to maximize cumulative rewards. This trial-and-error learning paradigm has shown promising results in dynamic environments and is particularly well-suited for games that require exploration, timing, and coordination.

Despite its promise, the use of RL in game development is largely constrained to research settings or high-budget projects. Training and deploying RL models demands expertise in both machine learning and software engineering, as well as significant computational resources. For small studios and independent developers, these requirements often place RL out of reach. Moreover, most existing RL agents are designed for specific game contexts, limiting their reuse across different projects or genres.

This project proposes a reusable and modular AI framework for 2D platformer games using reinforcement learning. The system is implemented within Unity and designed to simulate core player behaviors—such as walking, jumping, dashing, and obstacle avoidance—through a flexible architecture. Its modular structure enables developers to extend the agent's capabilities (e.g., adding special attacks or advanced mechanics) without modifying the core decision-making logic.

Although the focus is on 2D platformers, the framework is built with generalizability in mind. It aims to establish a foundation for developing scalable, adaptable AI agents that can be integrated into a range of game genres. By doing so, it addresses the gap between academic RL research and practical AI deployment in commercial game development, especially within the indie development space.

1.2 Problem Statement

While artificial intelligence has become a critical component in modern game development, many current AI systems suffer from poor adaptability, limited scalability, and a lack of reusability. These shortcomings are especially pronounced in the context of small or independent game studios, where technical expertise and computational resources are often constrained. Traditional game AI architectures—such as finite state machines and behavior trees—tend to produce rigid behaviors that are difficult to maintain or extend across new gameplay mechanics or level designs.

Reinforcement learning offers a powerful alternative by enabling agents to learn from interaction rather than relying on predefined rules. However, RL-based solutions are often developed for narrow use cases and are tightly coupled to specific game mechanics. Consequently, transferring such agents to new games or even

slightly modified environments typically requires retraining, restructuring, or extensive fine-tuning. This process introduces significant overhead, reducing the feasibility of using RL for most developers.
Moreover, the available RL frameworks often lack the modularity and documentation necessary for straightforward integration into existing projects. Pretrained agents, if available, are rarely reusable out-of-the-box and may not support features like modular action sets or customizable input pipelines. This restricts developers' ability to adapt AI systems to their own design needs without substantial reengineering.
To address these limitations, this project focuses on developing a generalizable RL-based AI agent for 2D platformer games. In this context, "solving" a game refers to the agent's ability to complete platformer levels by performing fundamental actions—such as moving, jumping, avoiding hazards, and optionally collecting in-game items. The agent learns these behaviors autonomously through trial-and-error training within a custom-built Unity environment. Once trained, the model is packaged into a Unity-compatible asset that includes integration tools, modular behavior components, and clear extension points for future customization.
By lowering the barrier to entry for reinforcement learning-based AI, this project aims to make intelligent agent development more accessible to indie developers. The outcome is not only a practical AI system for platformer games but also a scalable architectural framework that can be extended to a broader range of game genres and use cases.

1.3 Objectives

The primary objective of this project is to develop reusable, pretrained AI agents for 2D platformer games, packaged as a modular Unity asset that is easy to integrate, extend, and retrain. Specific objectives include:

1.3.1 Train Adaptive AI Agents for Platformer Mechanics

- Train reinforcement learning (RL)-based agents capable of executing core platformer actions—walking, jumping, dashing, and hazard avoidance—within varied level designs.
- Enable both player and enemy AI to generalize across different gameplay scenarios without requiring full retraining.

1.3.2 Develop a Modular and Reusable Unity Package

- Package the trained AI models and supporting systems into a Unity-compatible asset for straightforward integration.
- Create a modular framework that allows developers to adjust AI behavior and insert new mechanics without modifying the core architecture.

1.3.3 Support Customization and Scalability

- Implement an ActionModule system to facilitate the addition of new behaviors (e.g., double jump, special attack) through modular scripts.
- Ensure the system supports scalable AI behavior expansion for varying gameplay needs.

1.3.4 Establish a Generalizable AI Development Framework

- Provide a structured framework that outlines best practices for training, evaluating, and extending AI agents in Unity-based projects.
- Design the architecture for extensibility to future game genres beyond platformers.

1.3.5 Deliver Comprehensive Documentation and Developer Resources

- Provide detailed documentation, including setup instructions, customization workflows, and example use cases.
- Include tutorials, troubleshooting guides, and retraining guidelines for developers of varying experience levels.

1.4 Scope of Work

This project encompasses the design, development, training, and packaging of a reinforcement learning-based AI system for 2D platformer games. The scope includes:

1.4.1 Development of Game Environment and AI Agents

- Create a custom 2D platformer environment in Unity for training and evaluation of RL agents.
- Design and train player and enemy AI agents to effectively navigate, interact with, and respond to dynamic platforming challenges.

1.4.2 Creation of a Modular Unity Asset Package

- Build a Unity package containing pretrained models, core AI scripts, and example integration scenes.
- Ensure compatibility with common Unity project structures to facilitate easy adoption.

1.4.3 Implementation of ActionModule System for Extensibility

- Develop a modular ActionModule interface that supports the addition of new player actions and enemy behaviors.
- Allow developers to extend gameplay mechanics without modifying existing AI logic.

1.4.4 Evaluation and Testing of AI Performance

- Evaluate AI generalization across multiple levels and configurations.
- Conduct automated testing using AI agents to identify game design flaws and assess balance.

1.4.5 Framework and Documentation Delivery

- Provide a structured, reusable framework for AI training, integration, and extension.
- Deliver comprehensive documentation, including step-by-step guides, API references, and example use cases to support both novice and experienced developers.

1.5 Project Schedule

The Gantt chart below (Figure 1.1) illustrates the planned timeline for the project.

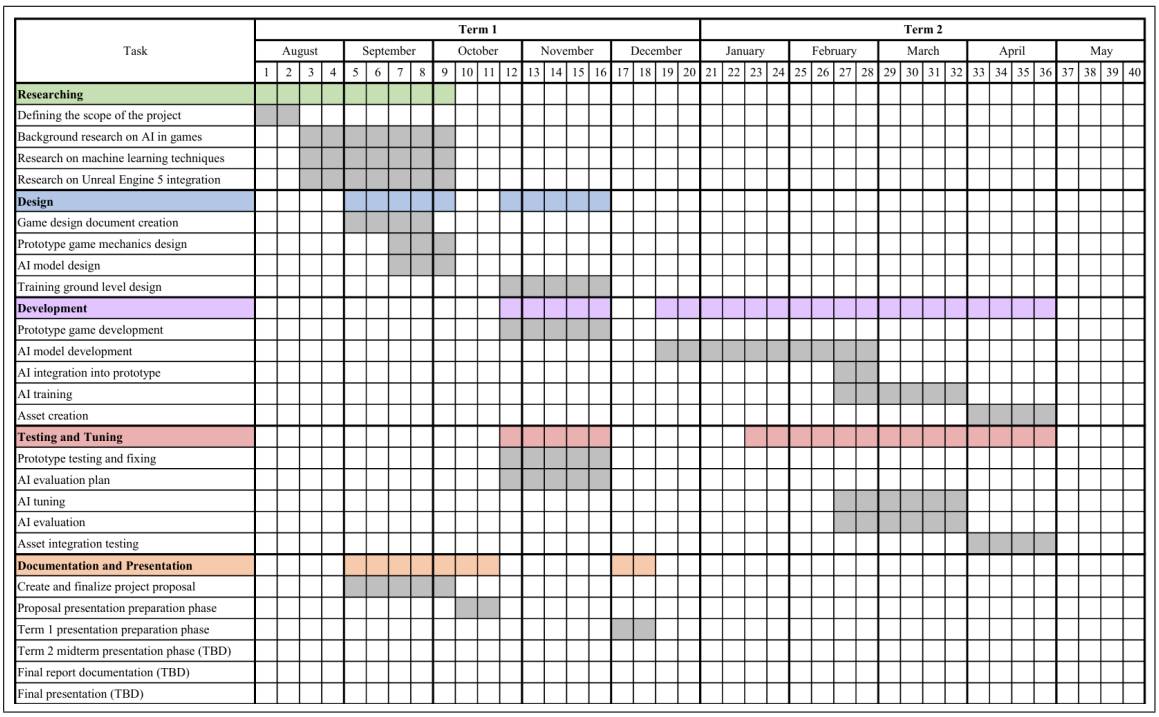

Figure 1.1 Gantt Chart

CHAPTER 2 BACKGROUND THEORY AND RELATED WORK

2.1 Introduction

This chapter provides the essential background knowledge and theoretical foundations necessary to understand the development of adaptive artificial intelligence (AI) in game environments. It covers key concepts, technologies, and programming languages utilized in AI development, with a focus on machine learning techniques such as reinforcement learning and relevant level design principles. Additionally, this chapter will explore related research and existing solutions, offering insights into the current state of AI in game development.

2.2 Theories and Core Concepts

2.2.1 Game Development

2.2.1.1 Game Development Life Cycle

Game development is a complex and multifaceted process that involves various stages, from initial concept to the final product. The main stages include:

- **Pre-production:** Conceptualization, storyboarding, and planning of the game. This stage defines the genre, mechanics, platform, and overall vision for the game.
- **Production:** The actual creation of the game, which includes coding, asset creation (art, sound, and animations), level design, and initial testing.
- **Post-production:** Refining the game through bug fixes, balancing, optimization, and quality assurance. This is also when the game is released to the public, followed by ongoing support such as updates and patches.

Throughout all stages, collaboration among developers, designers, artists, and sound engineers is essential to creating a cohesive experience. Game development today often involves teams of varying sizes, from small indie developers to large studios with hundreds of team members.

2.2.1.2 Game Design

Game design encompasses various elements that contribute to a game's playability, engagement, and overall success:

- **Game Mechanics:** The rules and systems that govern how the game works. These include the physics, player controls, objectives, and win/lose conditions.
- **Story and Narrative:** The storyline or plot that guides the player through the game. This can range from minimal or abstract narratives to fully immersive stories with complex characters and world-building.
- **Level Design:** The creation of game environments and how players progress through them. Effective level design includes balancing challenge, pacing, and exploration, often introducing new elements at each stage to maintain player interest.
- **Assets:** Game assets refer to the components used to build the visual, auditory, and interactive elements of a game. These include:

  - **Visual Assets:** Textures, 3D models, 2D sprites, animations, and visual effects that define the game's appearance.
  - **Audio Assets:** Sound effects, background music, and character voices that create the auditory experience.
  - **Code Assets:** Scripts and algorithms that govern game mechanics, AI behavior, and interactive elements.
  - **UI Assets:** Icons, buttons, menus, and other user interface components.

  Assets are the building blocks of any game, providing the necessary materials to bring concepts to life. Their quality and integration play a significant role in defining the overall aesthetic and functionality of the game.

- **User Interface (UI) and User Experience (UX):** The design of menus, HUDs (heads-up displays), and the interaction flow. A clean and intuitive UI is critical for keeping players immersed in the game.

2.2.1.3 Challenges Faced by Developers

- **Resource Limitations:** Small teams often lack the budget and manpower to implement complex systems like advanced AI, detailed art, or high-end graphics
- **Complexity of Game Design:** The more complex the game, the harder it is to manage. Creating a balanced and fun experience while ensuring that the game performs well across different platforms is a constant challenge.
- **Market Saturation:** With the rise of digital storefronts and indie game platforms, the market is flooded with games, making it hard for any individual title to stand out.
- **Time Constraints:** Many indie developers work on tight deadlines, often needing to finish a game within a limited period to avoid budget overruns or loss of momentum.

2.2.1.4 The Increasing Demand for Adaptive and Dynamic Gameplay

Players today expect more than static, predictable game experiences. They want games to react to their choices in real-time, creating a more dynamic and immersive experience. This demand has led to the integration of more advanced AI systems that can adapt and respond to player behavior in meaningful ways.
Games like *The Witcher 3* and *Horizon Zero Dawn* offer NPCs that respond realistically to the player's actions, creating a sense of a living world. This trend toward dynamic and reactive gameplay is challenging for developers, especially smaller teams with limited resources, as it often requires implementing sophisticated AI and adaptive systems that can learn and evolve.

2.2.2 Artificial Intelligence

Artificial Intelligence (AI) in gaming refers to the simulation of human-like intelligence and decision-making processes in game entities. It plays a vital role in enhancing the interactivity and immersion of games by enabling non-player characters (NPCs) to exhibit responsive and believable behaviors. AI in games can range from simple rule-based systems to sophisticated machine learning models that adapt and evolve over time.

2.2.2.1 Types of AI in Game Development

- **Finite State Machines (FSMs):** FSMs are one of the simplest AI systems used in games. They consist of a finite set of states (e.g., idle, attack, flee) and predefined transitions between these states based on specific conditions. FSMs are easy to implement but may lead to predictable and repetitive behaviors.
- **Behavior Trees:** Behavior trees are hierarchical structures used to manage decision-making processes. They allow for more modular and reusable AI behaviors compared to FSMs, making them popular in modern game development for controlling NPCs.
- **Utility-Based AI:** This system evaluates different actions based on a utility score, selecting the action with the highest score. Utility-based AI allows NPCs to make decisions based on dynamic priorities, leading to more flexible and context-aware behaviors.
- **Pathfinding and Navigation:** AI-controlled entities often need to move through complex environments. Pathfinding algorithms like A* (A-star) are widely used to calculate the shortest or most efficient paths, avoiding obstacles and optimizing movement.
- **Machine Learning in Games:** Machine learning (ML) introduces adaptability and dynamic behaviors that traditional systems cannot achieve. Techniques like reinforcement learning enable AI agents to learn from their interactions with the environment, improving their performance over time. These systems can produce more realistic, challenging, and unpredictable NPCs.

2.2.2.2 Applications of AI in Game Development

- **NPC Behavior and Opponent AI:** AI is commonly used to control NPCs, making them act as enemies, allies, or neutral characters. The goal is to create behaviors that challenge and engage players without being overly predictable or frustrating.
- **Procedural Content Generation (PCG):** AI can generate game assets such as levels, characters, and items dynamically. This enhances replayability and reduces development time by automating repetitive tasks.
- **Game Testing and QA:** Adaptive AI agents can be trained to simulate player behaviors, identifying bugs, balancing gameplay, and stress-testing game mechanics during development.
- **Player Profiling and Personalization** AI systems can analyze player data to adapt the gameplay experience, tailoring difficulty, content, and narratives to individual player preferences.

2.2.2.3 Advantages of AI in Game Development

- **Dynamic and Immersive Gameplay:** Adaptive AI creates experiences that feel more responsive and personalized.
- **Enhanced Development Efficiency:** Automating tasks like testing, content generation, and debugging reduces time and costs.
- **Scalability:** AI systems can be designed to adapt across different game genres and mechanics, enabling reusable assets.s

2.2.2.4 Challenges of AI in Game Development

- **Development Costs:** Advanced AI systems require significant computational resources and expertise, which can be a challenge for indie developers.
- **Complexity in Balancing:** Creating an AI that is both challenging and fair to players requires careful design and testing.
- **Unpredictable Outcomes:** In adaptive systems, emergent behaviors may lead to unintended gameplay consequences.

AI in game development is an ever-evolving field, driven by advances in computational power and machine learning algorithms. It holds tremendous potential for creating smarter, more immersive games while addressing development bottlenecks like scalability and automation.

2.2.3 Machine Learning

Machine learning involves programming computers to optimize a performance criterion based on example data or past experiences. It entails constructing models capable of recognizing patterns or regularities in data, enabling systems to make predictions, adapt, or gain insights. Grounded in statistical theory, machine learning employs efficient algorithms to manage large datasets and perform complex computations. It has widespread applications in fields such as retail, finance, healthcare, and telecommunications, addressing challenges like prediction, optimization, and pattern recognition.
Machine learning algorithms are commonly categorized into three types based on their learning paradigm: supervised learning, unsupervised learning, and reinforcement learning.

2.2.3.1 Supervised Learning

Supervised learning relies on labeled datasets, where the correct outcome is predefined, to train models. These models predict outcomes based on input data and refine predictions by learning from errors. Supervised learning is frequently applied in classification and regression tasks.

- **Naive Bayes:** Naive Bayes is a classification method based on Bayes Theorem, assuming that features are independent of each other. There are three types: Multinomial, Bernoulli, and Gaussian Naive Bayes. It's commonly used in text classification, spam detection, and recommendation systems.
- **K-nearest neighbor:** KNN is a non-parametric algorithm that classifies data points based on proximity, usually using Euclidean distance. It's simple and efficient for small datasets but becomes slower as the dataset grows. KNN is often used in recommendation engines and image recognition.
- **Random forest:** Random forest is a flexible supervised algorithm for classification and regression, combining multiple decision trees to reduce variance and improve accuracy.

2.2.3.2 Unsupervised Learning

Unsupervised learning models analyze unlabeled data to discover hidden patterns or structures. This approach is typically applied in clustering, dimensionality reduction, and anomaly detection.

- **K-means clustering:** K-means clustering groups data points into K clusters based on their distance from the centroids. Larger K values create smaller, more granular groups, while smaller K values result in larger clusters. It's widely used in market segmentation, document clustering, and image compression.

- **Principal component analysis (PCA):** PCA is a dimensionality reduction method that transforms data into new components, maximizing variance while reducing redundancy. Each successive component is uncorrelated and orthogonal to the previous one, capturing the most variance in fewer dimensions.
- **Autoencoders:** Autoencoders use neural networks to compress and reconstruct data, with the hidden layer acting as a bottleneck. The process is split into *encoding* (input to hidden layer) and *decoding* (hidden layer to output).

2.2.3.3 Reinforcement Learning

Reinforcement learning allows an agent to learn by interacting with an environment to achieve a specific goal. Mimicking trial-and-error learning, RL optimizes actions using rewards for goal-oriented behaviors and penalties for undesirable actions. This iterative process helps the agent refine its strategy to maximize cumulative rewards.

Key Components of RL:

- **Agent:** The learner or decision-maker.
- **Environment:** The system with which the agent interacts.
- **Actions:** Choices available to the agent.
- **Rewards:** Feedback to guide the agent's learning process.
- **Policy:** The strategy determining the agent's actions.

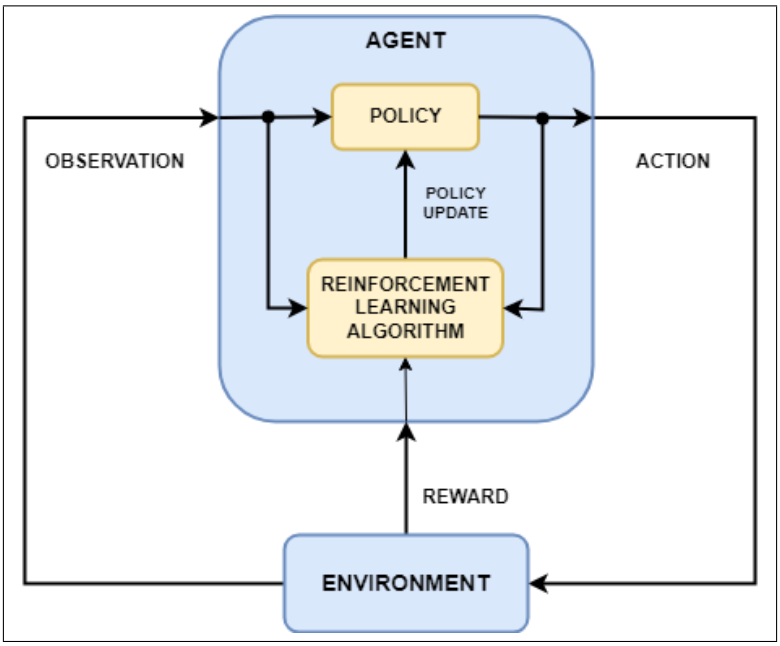

Figure 2.1 Reinforce Learning Model Diagram

Source: https://www.mathworks.com/help/reinforcement-learning/ug/what-is-reinforcement-learning.html

In game development, RL is particularly useful for training adaptive AI that can learn and evolve based on player interactions. This enables the creation of NPCs with more dynamic and challenging behaviors, as well as automated game testing agents that evaluate game mechanics and balance.

2.2.4 Reinforcement Learning

Reinforcement learning (RL) focuses on training agents to make sequences of decisions by interacting with an environment. The agent learns to optimize a long-term reward by observing states, taking actions, and receiving feedback in the form of rewards or penalties. This section explores key RL algorithms relevant to game development and their applications.

2.2.4.1 Q-Learning

Q-Learning is one of the simplest RL algorithms that builds a Q-value table to represent the expected future rewards for every state-action pair.

Mathematical Foundation

The Bellman Equation updates the Q-value for a given state-action pair iteratively

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma max_a(Q(s',a')) - Q(s,a)) \tag{2.1}$$

- $Q(s,a)$: Current Q-value for state s and action a.
- s: Current state
- a: Current action
- r: Reward received after taking action a in state s
- a': Next action
- s': Next state
- α: Learning rate, controlling the size of the update step.
- γ: Discount factor, determining the importance of future rewards.

Key Features

- Off-policy: It learns the optimal policy independently of the agent's actions.
- Suitable for discrete and small state-action spaces.

Use Case in Games

Training AI for grid-based games like Snake or Pac-Man.

Psuedocode

Algorithm 1 Q-Learning Algorithm

```
Initialize Q-table with zeros
for each episode do
    Initialize state s
    while not done do
        Choose action a using epsilon-greedy policy
        Take action a, observe reward r and next state s'
        Update Q-value:
        Q(s,a) ← Q(s,a) + α * (r + γ * max(Q(s',a')) − Q(s,a))
        s = s'
    end while
end for
```

2.2.4.2 Deep Q-Learning (DQN)

DQN scales Q-Learning to environments with continuous or high-dimensional state spaces by approximating the Q-function using a deep neural network.

Enhancements over Q-Learning:

- **Experience Replay:** Stores past transitions (s, a, r, s') in a replay buffer, sampling mini-batches for training to reduce correlation.
- **Target Network:** A separate network for computing target Q-values, updated less frequently to stabilize training.

Key Equations

Update the neural network weights θ by minimizing the loss:

$$L(\theta) = \mathbb{E}_{(s,a,r,s')}[(r + \gamma max_a Q(s', a; \theta^-) - Q(s, a; \theta))^2] \tag{2.2}$$

- $L(\theta)$: The loss function, a measure of how far the model's predictions are from the true or desired values.
- $Q(s, a; \theta)$: $Q(s, a)$ approximate using neural network θ.
- θ: Parameters of a neural network used to approximate a function.
- $\mathbb{E}$: The average value of the loss across sampled data points.

Use Case in Games

Training agents in platformers, complex strategy games, or visual navigation tasks.

Psuedocode

Algorithm 2 Deep Q-Learning Algorithm

```
Initialize replay buffer and Q-network
for each episode do
    Initialize state s
    while not done do
        Choose action a using epsilon-greedy policy
        Take action a, observe reward r and next state s'
        Store transition (s, a, r, s') in replay buffer
        Sample mini-batch from replay buffer
        Compute target: y = r + γmax(Q_target(s', a'))
        Update Q-network using gradient descent
        Periodically update target network
    end while
end for
```

2.2.4.3 Policy Gradient Methods

Policy gradient methods optimize the policy directly by maximizing the expected reward.

Policy Representation

The policy is parameterized as $\pi_\theta(a|s)$, where θ are the weights of the model.

Objective Function

$$J(\theta) = \mathbb{E}_{\pi_\theta}[\sum_t \gamma^t r_t] \tag{2.3}$$

Gradient Update:

Using the policy gradient theorem:

$$\nabla_\theta J(\theta) = \mathbb{E}[\nabla_\theta log \pi_\theta(a|s) R] \tag{2.4}$$

where R is the cumulative reward.

Use Case in Games

Training agents in games requiring smooth control, such as racing or flight simulators.

Psuedocode

Algorithm 3 Policy Gradient Methods

```
Initialize policy network
for each episode do
    Collect trajectory of states, actions, and rewards
    Compute discounted rewards for each state
    Update policy network to maximize log-probability of actions weighted by rewards
end for
```

2.2.4.4 Proximal Policy Optimization (PPO)

PPO improves policy gradient methods by introducing a clipped objective to ensure stable updates.

Objective Function

$$L^{CLIP}(\theta) = \mathbb{E}[\min(r_t(\theta)A_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t] \tag{2.5}$$

where $r_t(\theta)$ is the probability ratio, A_t is the advantage estimate, and ϵ is a hyperparameter for clipping.

Key Features

- Prevents large policy updates.
- Balances performance and training stability.

Use Case in Games

Robust AI for large-scale multi-agent systems or highly variable environments.

Psuedocode

Algorithm 4 Proximal Policy Optimization

```
for each iteration do
    Collect trajectories using current policy
    Compute advantages using value function
    Update policy network using clipped surrogate objective
    Update value network to reduce value estimation error
end for
```

2.2.4.5 Actor-Critic Methods

Actor-Critic combines policy optimization (actor) with value function estimation (critic).

Advantages

- Prevents large policy updates.
- Balances performance and training stability.

Variants

- **A2C:** Uses synchronous updates with multiple workers.
- **A3C:** Parallelizes training by running multiple agents asynchronously.

Use Case in Games

Real-time strategy games requiring fast adaptation.

Psuedocode

Algorithm 5 Actor-Critic Methods

```
for each iteration do
    Collect trajectories from actor network
    Compute value targets using critic network
    Update actor to maximize policy objective
    Update critic to minimize value estimation error
end for
```

2.2.4.6 Multi-Agent Reinforcement Learning (MARL)

In MARL, multiple agents learn and interact in a shared environment, adapting strategies collaboratively or competitively.

Key Challenges

- Coordination among agents.
- Balancing exploration and exploitation in dynamic environments.

Applications in Games

- Multiplayer strategy games.
- Cooperative tasks in simulation-based training.

Table 2.1 Comparison of RL Algorithms

Algorithm	Advantages	Limitations	Suitable Games
Q-Learning	Simple, effective for small problems.	Struggles with large state spaces.	Grid-based games, Puzzles
Deep Q-Learning (DQN)	Handles complex state spaces.	Computationally intensive.	Platformers, Visual games
Policy Gradient	Works well in continuous action spaces.	Susceptible to instability.	Racing, Flight simulators
PPO	Stable, efficient in diverse scenarios.	Trade-off between stability and speed.	Scalable multi-agent games
Actor-Critic	Fast convergence, suitable for real-time games.	Requires careful tuning.	Real-time strategy games, Simulators
MARL	Enables collaborative/ competitive AI.	Complexity increases with agent count.	Multiplayer games, Co-op challenges

2.3 Development Tools

2.3.1 Unity

Unity is a versatile and widely-used game development engine renowned for its ease of use, flexibility, and strong support for both 2D and 3D projects. It was chosen for this project due to its compatibility with machine learning tools, efficient workflow, and robust asset pipeline. Unity provides a range of tools for AI development, including the Unity ML-Agents Toolkit, NavMesh systems, and animation controllers, which enable the creation of intelligent and adaptive behaviors. These tools allow AI agents to interact with their environment, learn from rewards, and optimize performance effectively.

- **ML-Agents Capabilities** Unity's ML-Agents[1] Toolkit is a robust feature for integrating machine learning into game development. It supports training AI agents using techniques like reinforcement learning (RL), imitation learning, and other advanced approaches. Key features include:
  - **Reinforcement Learning:** ML-Agents supports deep reinforcement learning (DRL), enabling agents to learn through interactions with the environment by maximizing cumulative rewards. This is particularly suitable for developing adaptive AI behaviors in platformer games.
  - **Training Environments:** It simplifies the creation of training environments, allowing agents to interact with dynamic game worlds, respond to stimuli (e.g., obstacles or goals), and improve their performance iteratively.
  - **Training with Multiple Agents:** The toolkit supports simultaneous training of multiple agents in the same environment, ideal for creating coordinated enemy behaviors or competitive agents.
  - **Sensor Integration:** ML-Agents accommodates various sensors (e.g., visual, ray-casting, and custom inputs), enabling agents to gather feedback from the game world for informed decision-making.
  - **Model Export:** Trained models can be exported in the ONNX (Open Neural Network Exchange) format for seamless deployment in Unity, integrating advanced AI behavior into gameplay.
- **2D Game Support:** Unity's native 2D tools, including Tilemaps, physics, and animation systems, streamline the creation of platformer environments. This aligns perfectly with the project's goals by facilitating a smooth workflow for designing levels and mechanics.

2.3.2 C#

C# is Unity's primary programming language, offering an optimal balance of performance, ease of use, and flexibility. It is instrumental in implementing game mechanics and AI behaviors.

- **Performance Optimization:** C# enables efficient scripting for real-time calculations, game events, and AI logic. Its clean syntax and extensive libraries ensure that resource-intensive tasks, such as agent movement and obstacle detection, run efficiently.
- **Custom Gameplay Features:** Using C#, developers can create custom gameplay mechanics, such as pathfinding, player interactions, and dynamic environmental elements. Unity's scripting API provides precise control over agents and game objects, offering flexibility that extends beyond built-in tools.

2.3.3 Python

Python is a widely-used programming language in machine learning and AI development, known for its simplicity, flexibility, and extensive ecosystem of libraries. In this project, Python plays a critical role in developing and training AI models.

- **Machine Learning Development:** Python, in combination with PyTorch, is used to implement reinforcement learning algorithms (e.g., Deep Q-Networks, Proximal Policy Optimization) for training AI agents. Its high-level syntax accelerates experimentation and prototyping, making it ideal for AI development.
- **Data Processing:** Libraries like NumPy and Pandas are used for pre-processing and analyzing training data, enabling efficient monitoring of agent performance and model behavior.
- **Integration with Unity:** Trained models developed in Python can be exported (e.g., using ONNX) and integrated into Unity, where they control agent behavior within the game environment.

2.3.4 PyTorch

PyTorch is an open-source machine learning library widely used for deep learning and reinforcement learning. It plays a crucial role in this project for developing, training, and fine-tuning AI models to operate within the Unity environment.

- **Reinforcement Learning Compatibility:** PyTorch supports key reinforcement learning algorithms such as Deep Q-Networks (DQN) and Proximal Policy Optimization (PPO). These algorithms are integral to training AI agents, enabling them to interact with and learn from the game environment by adapting strategies based on feedback and rewards.
- **GPU Acceleration:** PyTorch's GPU acceleration significantly reduces training time for neural networks. This allows for faster iteration and experimentation, facilitating efficient model development and parameter tuning.
- **Integration with Unity:** Although PyTorch is Python-based, trained models can be exported in ONNX or other compatible formats for integration into Unity. This ensures that machine learning models seamlessly control in-game agents during gameplay, bridging the gap between AI development and deployment.

2.3.5 GitHub

GitHub is a version control and collaboration platform that facilitates the management of source code and assets throughout the project's development.

- **Version Control:** GitHub enables efficient version tracking, allowing developers to monitor changes to the codebase and revert to previous versions when necessary. This ensures stability and helps maintain control over the evolving project.
- **Collaboration:** GitHub supports team collaboration by allowing multiple developers to work simultaneously on different parts of the project. The platform's branching and pull request features ensure an organized and efficient development workflow, enabling seamless merging of contributions.
- **Backup and Deployment:** The repository serves as a secure backup for all project files and assets. Additionally, GitHub Actions can automate tasks such as deployment and testing, streamlining development and ensuring smooth continuous integration.

2.4 Related research

2.4.1 Automated Playtesting in Game Development

Playtesting is an important part of game development since it ensures that the challenges are balanced, systems work properly, and the player experience remains enjoyable. Traditionally, this process has relied on manual testers, which may be time-consuming and costly. In response to these issues, Sriram (2019)[2] researches the use of deep reinforcement learning (DRL) and curriculum learning to automate playtesting in 2D platformer games. His research introduces an Automated Playtesting (APT) program that trains AI agents to explore game levels, discovering design defects and gameplay imbalances without the need for operator input. By gradually increasing level complexity, these AI agents learn to adapt and manage a wide range of in-game events, making them useful for assessing new levels.

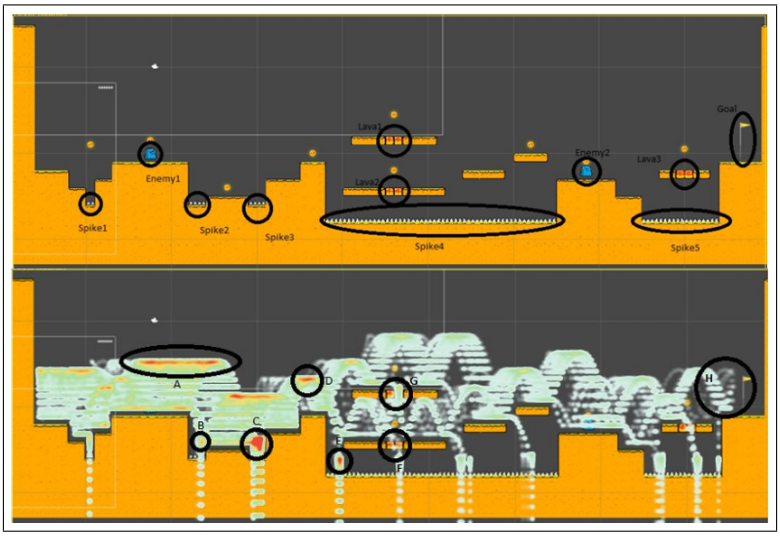

Figure 2.2 Unseen level and Unseen level with the agent heat map

Source: https://repository.library.northeastern.edu/files/neu:m0455c95d/fulltext.pdf

Sriram's methodology provides a data-driven strategy for playtesting, but it is not the only option. Scripted AI agents are a more traditional approach, as they follow specific rules and behaviors rather than learning flexibly. These bots are easier to implement and use far less computational resources. However, they lack flexibility, which means they cannot generalize well across game levels or detect unexpected gameplay problems. In comparison, DRL-based playtesting is a versatile and scalable method that is especially helpful for games with automated generation or advanced level designs.

2.4.2 AI Techniques in 2D Platformer Games

AI is an important tool for creating interesting yet challenging gameplay, particularly in platform games. In this area, AI behavior greatly influences player interest and difficulty. To improve AI behavior, Persson (2005)[3] studies three AI methods: pathfinding, image recognition, and line of sight. Their research focuses on AI movement and perception for more realistic interactions. The detection of line of sight ensures that the opponents will never see the player unless there are no obstacles in their path, preventing their irrational behavior. If they see changes in their environment with the aid of image recognition, then the AI will make appropriate changes to their actions. Finally, pathfinding algorithms enable AI to traverse complex levels, allowing for accurate movement while following or avoiding players.

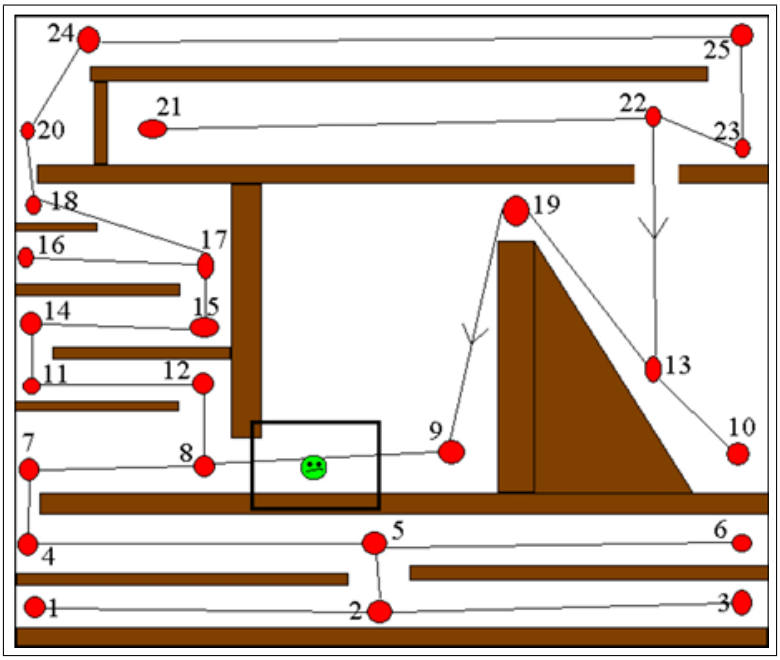

Figure 2.3 In a 2D platform game, pathfinding nodes are shown as ellipse with corresponding indexes and connections on the map.

Source: https://www.diva-portal.org/smash/get/diva2%3A4762/FULLTEXT01.pdf?utm_source=chatgpt.com

Behavior trees and FSMs make for an alternative pseudo-scientific contrast, they would be a fit in the context of game AI. FSMs provide a fixed, predictable AI behavior strategy, tending to have an extraordinarily rigid structure that requires much tuning to reproduce complex interactions. Behavior trees are about modularity and scalability, but still, they lack dynamically generated structures for decision making. Persson's solution offers a system that is more dynamic in character than pure FSM-based behavior systems but suffers from a downslope in computation.

2.4.3 Pathfinding and Navigation in Platformer AI

One main issue within platformer AI is making an intelligent movement deemed natural and responsive. Smith (2021)[4] has proposed a physics-based pathfinding system that moves AI-controlled characters with a real-life dynamic principle for level navigation. The construction of a platform graph is established, with surfaces as nodes and possible movement trajectories as edges. AI agents use the A* pathfinding algorithm to determine optimal routes between platforms following the physics enforced by the game. This is not pure AI control since Smith's system translates movement decisions into simulated player inputs to guard against jerkiness and make motions feel natural.

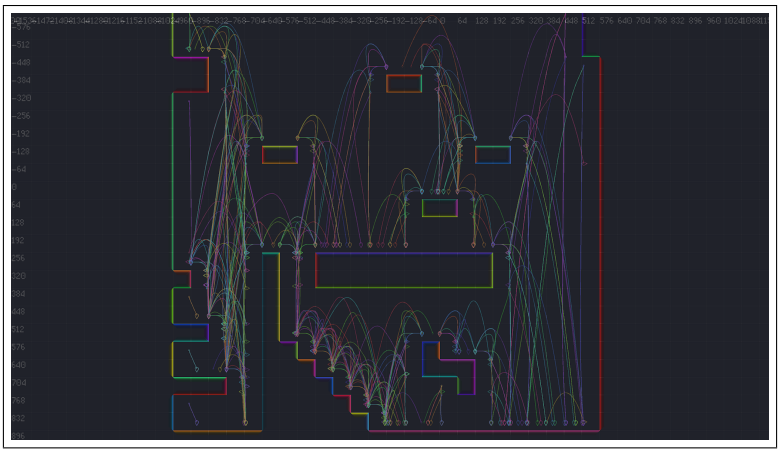

Figure 2.4 Using A* search to find path through the graph

Source: https://devlog.levi.dev/2021/09/building-platformer-ai-from-low-level.html?utm_source=chatgpt.com

Smith's algorithmic approach, in contrast to machine learning-based navigation systems, allows for a greater degree of control and responsiveness. However, it is still limited. Deep reinforcement learning (DRL) would be an alternative approach to AI navigation, where agents would learn movement strategies through trial and error. On the one hand, the advantage of DRL-based AI is its adaptation to a dynamically changing environment; this, of course, comes alongside their requirement of extensive training and computation resources. Smith's method, conversely, is fast and reliable during runtime, for their physics-based paths are pre-calculated, yet they lack the adaptability of AI solutions based on DRL to tackle the unforeseen changes of levels.

2.5 Gap Analysis

While artificial intelligence has made notable strides in 2D platformer games, several recurring limitations remain evident across the literature. For instance, in Sriram's (2019) project on automated playtesting using reinforcement learning, the AI agent was trained to navigate platformer levels for the purpose of validating level design. Although the study effectively showcased how deep reinforcement learning (DRL) can be applied in testing environments, the focus remained on single-agent behavior in relatively static and simplified levels. There was no interaction with dynamic enemies or evolving objectives, and the system lacked the flexibility to handle more diverse or unpredictable challenges—conditions often present in real gameplay.Similarly, Persson (2005) presented a foundational overview of traditional AI techniques in 2D platformers, including pathfinding, line-of-sight mechanics, and behavior trees. These techniques were—and still are—widely used due to their efficiency and ease of implementation. However, they rely on rule-based logic, which tends to produce predictable and rigid behavior. These systems often fail to generalize well when faced with new level structures or gameplay rules, limiting their usefulness in adaptive game environments or procedural content systems.Smith (2021) explored physics-based pathfinding using A* algorithms and platform graphs, providing a practical method for navigating complex 2D spaces. His work contributes valuable insights into deterministic path planning, especially in tightly constrained levels. However, this approach assumes a largely fixed environment and doesn't incorporate learning or adaptation over time. As a result, agents guided solely by static pathfinding algorithms struggle when unexpected gameplay changes occur—such as moving platforms, adversarial agents, or real-time hazards.Taken together, these studies highlight a clear opportunity for improvement: while each offers effective strategies for solving specific problems, they often fall short in handling dynamic gameplay, adapting to new challenges, or interacting with other agents. This is where our work seeks to contribute—by creating a reinforcement learning-driven framework that not only learns from experience but also adapts to a variety of gameplay elements through a modular, generalizable design.

2.6 How our work differs

This project builds upon the knowledge gained from the previous research while addressing the shortcomings of that research through a somewhat novel mixture of reinforcement learning, adversarial agent design, and modular asset creation. Sriram's (2019) work was specific to the use of a single agent to automatically validate levels; this work instead looks at having two agents interact within an environment—both the player and the enemy agent are being trained together. Such an adversarial setup makes things more complicated and realistic from the perspective of the agent and thus allows for a richer framework for gameplay-testing and behavior emergence. Our approach stands several other feet taller than Persson's (2005) rule-based ones as it puts Proximal Policy Optimization (PPO) to work to allow agent training from scratch using feedback from the environment while being more adaptive and functioning under some measures of uncertainty. This design choice of switching from hard-coded behavior to trainable policies vastly improves an agent's ability to function in procedurally varied or evolving levels.While Smith (2021) deals with deterministic navigation, what we have here is an integration of physics-aware decision-making into a learning framework, wherein an agent can compute a path but is also able to revise its traversal policy depending on the outcomes and depending on interactions with moving obstacles or enemies. Another major innovation is a modular and reusable asset design. In contrast to many prior approaches that custom-build AI systems for specific games or sets of use cases, this platformer AI is designed to be a stand-alone Unity asset, which can be plugged into all kinds of 2D games with very few modifications. This is, thus, a goldmine for indie developers and researchers.

CHAPTER 3 METHODOLOGY AND DESIGN

3.1 Introduction

This chapter outlines the methodology and design principles employed in the development of an adaptive artificial intelligence (AI) system tailored for 2D platformer games. The project integrates reinforcement learning into a modular AI framework, aiming to provide a reusable and easily extendable solution for game developers.

Building on the theoretical foundations established in the previous chapter, this section focuses on the practical aspects of system implementation. The goal is to bridge academic research in reinforcement learning with real-world game development tools and workflows. To achieve this, a structured, modular design was adopted to ensure flexibility, reusability, and compatibility across different 2D platformer projects.

This chapter will outline the key components of our methodology, including:

- **System Architecture:** An overview of the core system components, including Unity, ML-Agents, and the communication pipeline between the training environment and learning algorithm.
- **Game Environment Design:** A description of the custom 2D platformer environment developed for training and evaluating AI behavior.
- **AI Algorithms and Models:** The selection and implementation of reinforcement learning techniques, specifically Proximal Policy Optimization (PPO).
- **Training Process and Optimization:** Details on training configuration, reward structure, curriculum learning considerations, and agent evaluation metrics.
- **AI Integration and Packaging:** The process of embedding the trained AI into a Unity-compatible asset and enabling modular extensibility through custom action modules.

By following this methodology, the project aims to deliver a robust AI development framework that not only supports player and enemy agent training but also provides an accessible entry point for indie developers to integrate, extend, and retrain AI agents for their own games.

3.2 System Architecture

The system architecture developed for this project facilitates the training, evaluation, and deployment of adaptive AI agents within a Unity-based 2D platformer environment. It is designed to support modularity, enabling seamless integration with external reinforcement learning frameworks and reusability across multiple projects.

The architecture consists of six core components:

- A Unity-based training environment for simulating gameplay interactions.
- AI agents (player and enemy) that perceive, decide, and act in response to environmental stimuli.
- Behavior modules that define the agent's input-output processing pipeline.
- Environment parameters, which enable dynamic adjustment of gameplay variables during training.
- A Python-Unity communication interface (Communicator) for transmitting observations and actions.
- A Python-based trainer implementing the Proximal Policy Optimization (PPO) algorithm.

Figure 3.1 visualizes the architecture and interconnections between these components.

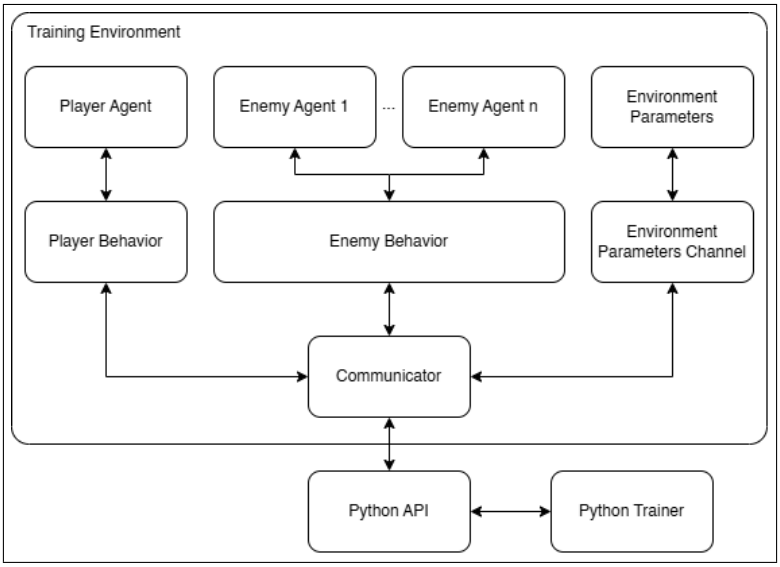

Figure 3.1 System Architecture Diagram, drawn on draw.io

3.2.1 Training Environment

The training environment is a controlled simulation where AI agents interact, learn, and refine their decision-making abilities through reinforcement learning. It is structured within a Unity scene that replicates typical platformer game scenarios, including various obstacles, enemies, and interactive elements. By defining key training parameters—such as agent actions, observations, and reward structures—the system ensures an effective learning process, guiding the agents toward optimal gameplay strategies.

3.2.1.1 Agents

Agents are entities in the environment that actively interact with their surroundings. Each agent observes the current state of the game environment using provided sensors, make decisions based on the information, and perform actions accordingly. These actions influence the environment and the agent's progression toward its assigned goals.

Agents are also subject to a reward system, where positive or negative feedback is assigned based on the outcomes of their actions. This feedback mechanism enables agents to learn and adapt their behavior over time, aligning their actions with the objectives of the game.

In this project, agents are categorized into two types:

- **Player Agent:** The primary agent designed to complete platformer levels by efficiently navigating the environment. It learns to overcome obstacles, avoid hazards, collect objectives, and reach predefined goals. The player agent undergoes reinforcement learning to optimize movement patterns, timing of actions, and strategic decision-making.
- **Enemy Agents:** Secondary agents that introduce dynamic challenges within the game. These agents exhibit various behaviors, such as patrolling, chasing, or attacking the player agent. Enemy agents contribute to the complexity of the game environment by enforcing strategic decision-making on the part of the player agent, thereby enhancing the training process.

Both types of agents undergo reinforcement learning, where their interactions with the environment influence their long-term behavior. The iterative training cycle ensures that agents develop effective strategies tailored to their specific roles.

3.2.1.2 Behaviors

Behaviors define how an agent interprets its surroundings and responds to different stimuli within the game environment. A behavior module processes input observations, evaluates potential outcomes, and determines appropriate actions to achieve a predefined objective. In reinforcement learning, behaviors are continuously refined based on the received rewards, enabling the agent to develop adaptive and optimized responses over time. Each behavior module consists of the following key components:

- **Input Observations:** The agent perceives relevant environmental data, such as the position of obstacles, enemy movements, collectible items, and the agent's current state (e.g., velocity, health, and platform contact).
- **Decision Processing:** The AI model evaluates possible actions based on the received observations and selects the most optimal response according to the learned policy.
- **Action Execution:** The agent performs an action within the game environment, influencing its surroundings and affecting future observations.

By iteratively refining behaviors through reinforcement learning, the AI system ensures that agents develop intelligent, context-aware decision-making processes.

3.2.1.3 Environment Parameters

Environment Parameters are configurable settings within the game that can be adjusted during runtime. These parameters can control elements like the difficulty level, enemy behavior, or platform properties. Through Side Channels, Python can send updates to Unity to modify these parameters, while Unity can also send data back to Python for analysis. This two-way communication enables dynamic adjustments, precise control over agent training, and scalability for creating varied scenarios.

3.2.1.4 Communicator

The Communicator component is responsible for enabling seamless data exchange between Unity and external Python-based reinforcement learning processes. It establishes real-time communication channels for transmitting agent observations, receiving policy updates, and executing AI-driven decisions within the game environment.

- **Sending Observations to Python:** Unity collects environmental data and transmits it to the reinforcement learning algorithm.
- **Processing Actions in Python:** The AI model processes the received observations, updates its policy, and determines the optimal action based on learned strategies.
- **Returning Actions to Unity:** The chosen action is sent back to Unity, where it is executed by the corresponding agent.

This interaction is fundamental to reinforcement learning, as it allows agents to iteratively refine their behaviors based on real-time feedback from the game environment

3.2.2 Python API and Trainer

The Python API serves as the interface between Unity and the external machine learning framework responsible for training the AI models. It enables the execution of reinforcement learning algorithms, processes training data, and updates policy models based on collected experience.

The Python Trainer is responsible for implementing the reinforcement learning algorithm used in this project. The selected approach is **Proximal Policy Optimization (PPO)**, a policy-gradient method that balances exploration and exploitation to optimize agent performance. The training process consists of the following stages:

- **Data Collection:** The AI agent interacts with the game environment, accumulating experience in the form of state-action-reward sequences.
- **Policy Optimization:** The collected data is used to update the agent's decision-making policy, refining its action-selection strategy.
- **Evaluation and Refinement:** The updated policy is tested, and further adjustments are made to improve performance.

Through iterative training cycles, the AI model progressively enhances its decision-making capabilities, leading to more intelligent and efficient gameplay strategies. The integration of the Python API and Trainer ensures a seamless connection between Unity and reinforcement learning frameworks, enabling scalable and efficient AI development.

3.3 Game Environment Design

The game environment serves as the primary medium through which the AI agent interacts with and learns from the world. Designing an effective environment is crucial for reinforcement learning (RL), as it defines the agent's possible actions, objectives, challenges, and reward mechanisms. This section outlines the structural and functional aspects of the game world, detailing level composition, physics interactions, agent perception, and the reinforcement learning framework.

3.3.1 Level Structure and Composition

The environment is designed as a 2D platformer world, incorporating fundamental elements commonly found in platformer games, such as platforms, obstacles, hazards, collectibles, and enemies. The level is structured to balance exploration, platforming challenges, and enemy encounters, ensuring that the agent experiences a diverse range of situations during training. Key elements included:

3.3.1.1 Platforming and Terrain

The game world consists of various terrain types that influence agent movement and decision-making:

- **Solid Platforms:** Standard ground elements on which the agent can walk and jump.
- **One-Way Platforms:** Platforms that can be landed on from above but allow the agent to jump through from below.
- **Moving Platforms:** Dynamic platforms that require precise timing and positioning.
- **Wall Surfaces:** Walls allow for interactions such as wall jumps or wall slides, affecting navigation strategies.

3.3.1.2 Interactive Elements

- **Goals (Coins, Checkpoints):** Objects that incentivize exploration and progression.
- **Hazards (Spikes, Traps, Pits):** Objects that penalize incorrect movement decisions and reinforce safe navigation.

- **Enemies:** AI-controlled obstacles that introduce combat-based decision-making for the agent.

These elements are strategically placed to guide the agent's learning, providing a structured environment that encourages skill acquisition through reinforcement learning.

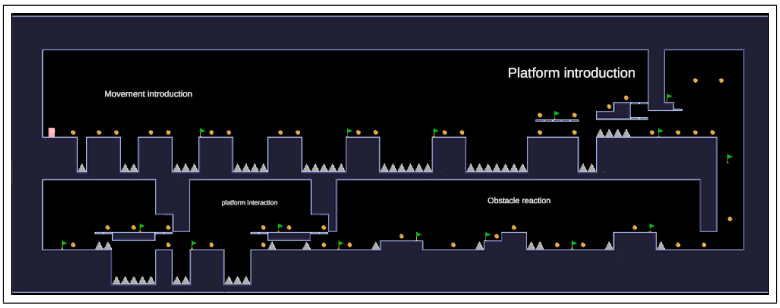

Figure 3.2 Environment training ground for the Ai

3.3.2 Agent Actions

Table 3.1 outlines the available actions for Player and Enemy agents. These actions simulate the gameplay mechanics and ensure that the AI can engage with the environment and other agents meaningfully.

Table 3.1 Agent Actions Table

	Player Agent	**Enemy Agent**
Actions	Walk and run Jump Attack (Melee) Drop Down Dash	Walk and run Jump Attack player (Melee)

3.3.2.1 Player Agent Actions

- **Walk and run:** Allows navigation of horizontal spaces.
- **Jump:** Essential for crossing gaps and reaching elevated platforms.
- **Attack (Melee):** Simulates combat mechanics where the player can engage with enemies or break objects.
- **Drop Down:** Enables traversal through one way platform accessing other places.
- **Dash:** Enables moving horizontally at a higher speed avoiding obstacles.

3.3.2.2 Enemy Agent

- **Walk and run:** Supports patrol or chase behaviors.
- **Jump:** Ensures enemies can traverse complex terrains.
- **Attack (Melee):** Adds combat mechanics, making the environment more dynamic.

3.3.3 Physics and Movement Constraints

The game environment adheres to a realistic 2D physics system that governs movement dynamics. The agent's movement system follows industry-standard platformer mechanics, ensuring consistency with human gameplay expectations.

3.3.3.1 Movement Mechanics

- **Walking and Running:** The agent moves laterally within predefined speed limits.
- **Jumping:** The agent can jump with variable force depending on input duration.
- **Dashing:** A short burst movement in a direction, used for evasion or precise navigation.
- **Wall Interactions:** The agent can cling to or slide down walls, enabling wall jumps for vertical traversal.

3.3.3.2 Collision and Interaction System

The physics engine handles collisions and interactions between the agent and the environment using Unity's built-in 2D physics system.

- **Ground Detection:** Determines whether the agent is standing on a platform, affecting movement choices.
- **Hazard Detection:** Triggers damage events when colliding with harmful objects.
- **Enemy Interaction:** Enables combat-related decisions such as attacking or avoiding enemies.

By enforcing a consistent movement and interaction model, the environment ensures that the agent learns within realistic constraints, improving the generalizability of the trained AI.

3.3.4 Environment Testing

Before integrating the reinforcement learning model, rigorous environment testing is conducted to ensure the platformer world is functionally sound, scalable, and optimized for training.

3.3.4.1 Functional Testing

Functional tests verify that core game mechanics operate as intended:

- **Movement Tests:** Ensuring the agent can walk, jump, dash, and interact with platforms correctly.
- **Collision Tests:** Verifying that the agent properly detects and responds to ground, walls, hazards, and enemies.
- **Camera Tests:** Confirming that the ML sprite system correctly renders simplified perception inputs.

3.3.4.2 Performance Optimization

Since reinforcement learning requires high-frequency environment interaction, performance optimizations are applied:

- **Physics Simplification:** Reducing unnecessary physics calculations to improve simulation speed.
- **Memory Management:** Ensuring that ML sprite rendering and collision detection do not create excessive overhead.
- **Frame Rate Stability:** Maintaining consistent simulation speed to ensure uniform training conditions.

3.3.4.3 Scalability Testing

To ensure the environment remains adaptable for future projects, scalability tests evaluate:

- **Level Size Variability:** Testing different map layouts to verify adaptability.
- **Enemy and Obstacle Density:** Measuring the impact of increased AI-controlled elements on performance.
- **Generalization Potential:** Ensuring that the AI can learn effectively in varied level designs without overfitting to a specific layout.

This testing process ensured that the game environment aligned with the project's objective of creating robust and reusable AI assets for game development.

3.4 AI Algorithms and Models

The artificial intelligence (AI) system in this project is designed to function as a reinforcement learning (RL) agent, capable of autonomously learning and executing platforming mechanics. The primary objective is to develop an adaptive AI agent that improves its performance through iterative learning. This agent is implemented using Unity ML-Agents, and training is conducted within the custom-built platformer game environment.

3.4.1 Action Space and Decision-Making

The action space defines the possible decisions the AI agent can take at any given time.
The following discrete actions are implemented in the platformer environment:

Table 3.2 Action Space Table

Action	Description
Move Left	Moves the agent left along the x-axis.
Move Right	Moves the agent right along the x-axis.
Jump	Initiates a jump if the agent is on solid ground.
Drop Down	Allows the agent to descend through one-way platforms.
Dash	Performs a short burst movement in the selected direction.
Attack	Executes a melee attack when enemies are in range.

The decision-making process is handled by the AI model, which outputs a probability distribution over possible actions at each time step. The agent selects an action based on learned policies, with an emphasis on maximizing future rewards.

3.4.2 Reward and Penalty System

The reinforcement learning model requires a reward function to guide learning. This function assigns positive rewards for desirable actions and penalties for suboptimal behavior.

3.4.2.1 Player Agent Rewards and Penalties

The Player Agent is primarily focused on completing levels and maximizing performance. The rewards and penalties assigned to the Player Agent are designed to incentivize behaviors that contribute to level progression, combat efficiency, and overall success.

Table 3.3 Player Agent Reward Table

Action	Rewards
Level Completion	+1000
Enemy Defeated	+100
Collecting Objective Items	+50
Interact with Objectives	+20
Exploring	+1 per new point reached
Falling or Hitting Hazards	-50
Taking Damage from Enemy	-200
Eliminated	-1000

3.4.2.2 Enemy Agent Rewards and Penalties

The Enemy Agents are designed to create challenges for the Player Agent. Their behavior is shaped by rewards and penalties that encourage actions which counter the Player Agent's progress.

Table 3.4 Enemy Agent Reward Table

Action	Rewards
Player Damaged	+100
Chasing Player	+5 for every period of time chasing player
Survival Time	+1 for every period of time survived
Falling or Hitting Hazards	-50
Eliminated	-200

This reward system is fine-tuned iteratively to ensure that the AI develops efficient and strategic movement patterns without exploiting rewards through unintended behaviors.

3.4.3 Agent Perception and Observations

To facilitate intelligent decision-making, the AI agent perceives the environment through a structured observation space, eliminating reliance on raycasting and instead leveraging a dedicated agent camera with simplified ML sprites. This approach mirrors human vision-based gameplay and enhances the interpretability of learned behaviors.

3.4.3.1 Observation Space and State Representation

The agent receives structured inputs that define its state and surroundings:

Positional and Movement Awareness

- **Current position** within the game world.
- **Grounded status** `IsGrounded` determines if the agent is standing on solid ground.
- **Jumping status** `IsJumping` determines if the agent is jumping or in mid air from the ground.
- **Dashing status** `IsDashing` determines if the agent is in the dashing state.
- **Dropping status** `IsDropping` determines if the agent is dropping from a one way platform.
- **On-wall status** `IsOnWall` determines if the agent is on the wall, available for wall related actions.
- **Wall-jumping status** `IsWallJumping` determines if the agent is jumping off from a wall.
- **Facing direction** `IsFacingRight` determines if the agent is facing right or left.

Environmental Awareness

- Proximity to enemies, hazards, or collectibles.
- **Checkpoint status:** tracks progress within the level.
- **Health and damage state:** monitors the agent's survival status.

3.4.3.2 Camera-Based Perception and ML Sprite System

A unique aspect of this project is the decision to replace traditional raycasting techniques with a dedicated agent camera. This approach mimics human player perception, where decision-making is based on visible elements rather than invisible proximity checks or abstract sensor inputs. To implement this, the game utilizes a dual-layer rendering system:

- **Primary Camera (Player Perspective):** The main camera, which is responsible for rendering the actual game world as seen by the player. This includes fully detailed textures, lighting effects, UI elements, and all in-game objects categorized into specific layers such as Ground, Enemy, Prize, Checkpoint, etc.

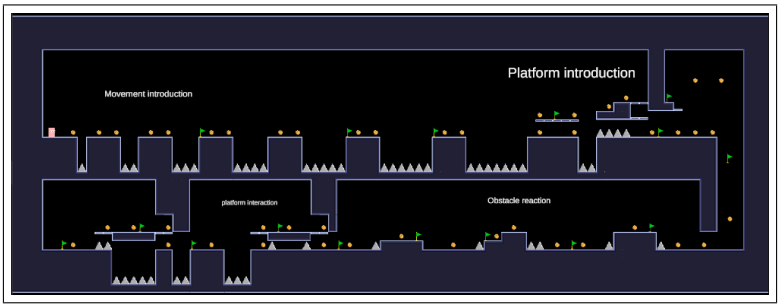

Figure 3.3 Player perspective

- **Agent Camera (AI Perspective):** A separate camera dedicated to the AI agent, rendering a simplified version of the game world. This camera ignores visual effects and detailed textures, instead utilizing a distinct ML sprite system composed of minimalistic geometric representations of in-game objects.

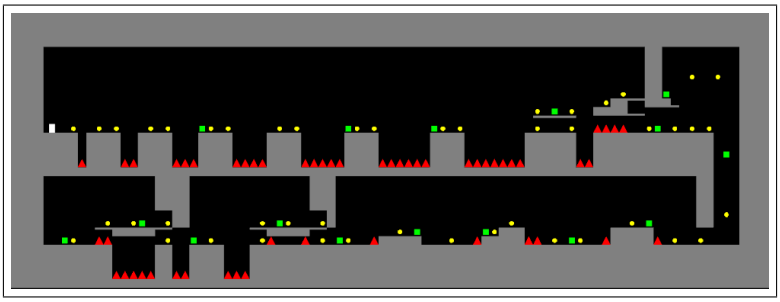

Figure 3.4 AI perspective

ML Sprite Classification System

Each object in the environment is assigned a secondary ML sprite, which is rendered exclusively for the agent camera. These ML sprites are categorized into distinct layer masks and follow a predefined color-coded classification system to provide the AI with structured, interpretable visual data.

Table 3.5 Sprite Color Code Table

Color	Meaning	Hex Code
White	Player	#FFFFFF
Red	Hazards (traps, spikes)	#FF0000
Magenta	Enemies (attackable entities)	#FF00FF
Green	Goals (checkpoints, exits)	#00FF00
Yellow	Collectibles (coins)	#FFFF00
Blue	Attacking hitboxes	#0000FF
Gray	Ground (solid platforms)	#808080

The classification scheme is as follows:
Through this system, the agent perceives objects in an abstract yet structured manner, focusing only on relevant gameplay elements rather than unnecessary graphical details. This enables a highly extensible framework where additional layers or object categories can be introduced seamlessly by developers integrating this AI into their own projects.

3.5 Training Process and Optimization

The training process of the AI agent follows a structured pipeline designed to optimize learning efficiency and ensure convergence towards an optimal policy. This process consists of data collection, policy network updates, curriculum learning strategies, and performance evaluation metrics. By leveraging reinforcement learning (RL) principles, particularly policy optimization methods, the AI progressively refines its ability to navigate the platformer environment.

3.5.1 Data Collection and Experience Buffer

During training, the AI agent interacts with the environment and records state-action-reward sequences. These experiences are stored in a replay buffer, which helps the agent learn by analyzing past interactions.

3.5.1.1 Experience Replay and Data Storage

Unlike traditional online learning, where updates occur immediately after each action, experience replay (as employed in off-policy learning techniques) allows the AI agent to learn from a diverse set of past interactions. This enhances sample efficiency and reduces training instability by preventing excessive bias toward recent experiences.
The data collection process consists of:

- **State Observations:** Capturing spatial and contextual information from the environment at each timestep.
- **Action Selection:** The policy network predicts the most suitable action based on past experiences.
- **Reward Assignment:** The agent receives feedback (reward or penalty) based on its action.
- **Storage in Replay Buffer:** The tuple (state, action, reward, next state) is stored in memory.
- **Batch Sampling for Training:** A mini-batch of past experiences is selected for policy updates.

3.5.2 Policy Network Training

he AI agent employs a neural network-based policy model, where policy parameters are updated iteratively using gradient-based optimization techniques. The training process consists of several key components:

- **Collecting gameplay experiences:** The AI agent performs actions in the environment.
- **Evaluating rewards:** The system calculates cumulative rewards for different strategies.
- **Updating the policy:** The AI refines its decision-making using policy gradient updates.

3.5.3 Curriculum Learning Approach

To facilitate structured learning, a curriculum-based training strategy is employed, wherein training difficulty is gradually increased as the AI model improves. This approach enables more stable learning and prevents the agent from being overwhelmed by complex tasks in the early stages.
The curriculum is structured into four phases:

- **Phase 1: Basic Movement Training:** The agent learns fundamental mechanics such as walking, jumping, and dashing in an obstacle-free environment.
- **Phase 2: Platforming Challenges:** The environment introduces gaps, moving platforms, and elevation changes to refine traversal skills.
- **Phase 3: Combat Scenarios:** The agent encounters dynamic enemies, learning attack patterns and defensive maneuvers.
- **Phase 4: Full-Level Training:** The AI is trained in fully designed levels, requiring mastery of all mechanics to complete objectives.

3.5.4 Performance Metrics and Convergence

The efficiency of training is evaluated using key performance indicators (KPIs) that measure learning progress and policy effectiveness.

3.5.4.1 Cumulative Reward Trend

- Tracks the total reward accumulated per episode.
- Indicates whether the agent is improving over time.
- A consistently increasing reward trend suggests successful learning.

3.5.4.2 Episode Length

- Measures the duration of each playthrough.
- Shorter episodes in earlier training stages may indicate suboptimal policies (e.g., frequent deaths).
- As training progresses, longer episode lengths suggest improved survival and decision-making.

3.5.4.3 Success Rate

- Defined as the percentage of episodes in which the agent successfully reaches level objectives.
- A threshold success rate (e.g., 95%) can be set to determine when training should be finalized.

3.5.4.4 Training Convergence

Training is considered complete once performance metrics stabilize, indicating that additional training no longer significantly improves the policy.
Convergence is monitored using:

- **Policy Entropy:** Ensuring the model does not become overconfident in suboptimal actions.
- **Variance in Success Rate:** Stability in success rate across multiple test runs.
- **Generalization Testing:** Evaluating the AI on unseen levels to verify adaptability.

3.6 AI Implementation and Packaging

The successful deployment of the trained AI model within a game development environment necessitates a structured approach to integration, optimization, and packaging. This process ensures that the AI agent functions efficiently within the Unity-based platformer while also being modular and reusable for future projects. Additionally, the AI system is designed to be packaged as a standalone asset that developers can easily integrate into similar games without requiring extensive modifications.
This section outlines the methodology for embedding the AI model into the game engine, optimizing its execution, setting up a dual-rendering system for enhanced perception, and packaging it as a shareable Unity asset.

3.6.1 Integration of the Trained AI Model

After the reinforcement learning model has been trained and validated, it is integrated into the Unity project for real-time inference. The AI must be able to process environmental inputs and execute actions effectively while maintaining computational efficiency.

3.6.1.1 Loading and Utilizing the Trained Model

The trained model is exported as a `.onnx` file, a format compatible with Unity's ML-Agents inference system. The steps for incorporating the model into the game are as follows:

- **Model Importion**
  - The `.onnx` model file is placed into the Unity project's Assets directory.
  - Unity's Behavior Parameters component is configured to reference the model.
- **Agent Configuration**
  - The AI agent is modified to switch from training mode to inference mode.
  - The model receives observations from the environment and produces action outputs in real time.
- **Testing and Validation**
  - The AI's performance is validated within a controlled game scenario.
  - Debugging tools are used to monitor decision-making processes and adjust parameters as needed.

3.6.2 Dual-Rendering System for Agent Perception

To enhance the AI's perception of the game environment, a dual-rendering system is implemented, allowing the agent to interpret its surroundings using a dedicated perception layer distinct from the player's visual representation. This approach improves object recognition and state representation, optimizing learning efficiency and decision-making accuracy.

3.6.2.1 Design and Implementation

- **Primary Camera (Player View)**
  - This camera renders the standard game environment for the player.
  - Traditional visual element layers, including textures, lighting, and UI elements, are rendered with this camera.
- **Secondary Camera (Agent Perception View)**
  - A separate, hidden camera renders a simplified version of the environment specifically for the AI.
  - Objects are color-coded as provided in the table 3.5 instead of textured, reducing unnecessary complexity.
  - Dynamic elements such as enemies and hazards are highlighted distinctly.
- **Integration with ML-Agents**
  - The AI model receives input from the perception camera rather than the full game scene.
  - Observation preprocessing converts the camera feed into a structured tensor representation.

3.6.2.2 Advantages of the Dual-Rendering System

- **Reduces Unnecessary Complexity:** The AI perceives only relevant elements, improving training efficiency.
- **Standardizes Input Data:** Ensures consistency across different game levels or environments.
- **Optimizes Learning Speed:** Simplifies object recognition, allowing the model to generalize more effectively.

3.6.3 Modular AI System for Reusability

To ensure adaptability across multiple projects, the AI system follows a modular design, enabling easy integration with different platformer environments. The key components include:

- **ActionModule:** Centralizes AI interactions with the game, allowing for customization of available actions (e.g., jumping, attacking).
- **Customizable Observations:** The AI's input space can be modified based on environmental conditions.
- **Configurable Reward System:** Developers can fine-tune rewards to optimize learning outcomes.
- **Flexible Behavior Parameters:** AI response time, exploration rate, and decision frequency can be adjusted dynamically.

3.6.4 Packaging the AI as a Reusable Unity Asset

To facilitate widespread adoption, the AI system is packaged as a Unity asset bundle, allowing developers to easily import and configure the AI within their own projects. The packaging process follows structured guidelines to ensure compatibility and ease of use.

3.6.4.1 Asset Bundling and Exportation

- The AI scripts, model files, and configuration settings are structured into a self-contained Unity package.
- Dependencies such as ML-Agents are documented to streamline integration.
- A standardized folder structure is maintained to enhance clarity.

3.6.4.2 Documentation and User Guide

- A `README.md` file provides quick-start instructions.
- A detailed PDF manual explains AI setup, customization, and troubleshooting.
- Code examples illustrate how to modify AI behaviors for specific game mechanics.

3.6.4.3 Version Control and Distribution

- The AI package is versioned, with changelogs detailing improvements.
- Distribution options include GitHub, Unity Asset Store, and private repositories.
- Developers are encouraged to contribute enhancements to ensure long-term maintainability.

CHAPTER 4 RESULTS AND CURRENT AI CAPABILITIES

This chapter presents the current state of the AI system in the wake of training and implementation. In-depth consideration is given to the performance of the AI within the established environment in terms of ability to navigate, interact with the game elements, and make decisions through the reinforcement learning framework. The results portray the present-day functionality of the system without speculation regarding future developments or improvements.

4.1 Overview of Accomplishments

4.1.1 Movement and Navigation

After carefully training for an extensive period using the PPO reinforcement learning algorithm, the AI agent has become capable of fully controlling itself in the platformer's environment. With a reliably functioning walking, jumping, and descent mechanism, the agent can drop straight from one-way platforms. As a result of iterative reinforcement learning, the model knows the optimal paths of movement by which he could traverse efficiently without much delay, avoid an obstruction, and come to certain destinations within the level. The AIs keep both exploration and efficient across-the-board movement and minimizing downside-avoiding turns while remaining adaptive to different terrains.

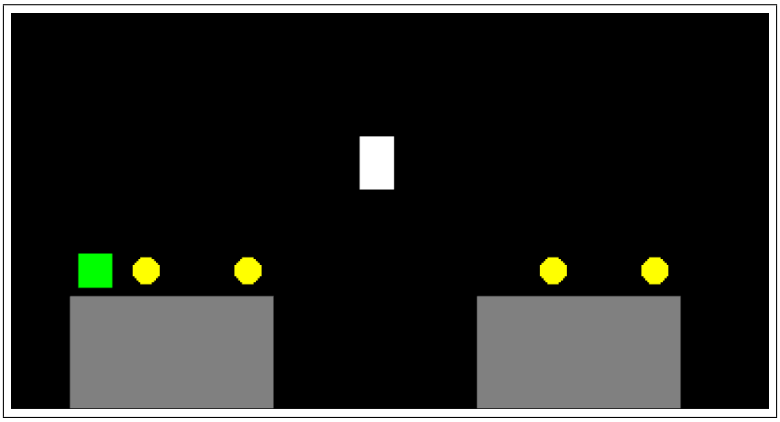

Figure 4.1 AI bot hopping from ground to another.

4.1.2 Interaction with Collectibles and Obstacles

The AI agent acquired knowledge in the interaction of collecting objects and environmental obstacles. It has also learned to identify and prioritize collecting objects, such as coins, adjust its movement to them, and optimize the reward received from them. The improvement of avoidance by hazardous events has demonstrated the ability to recognize penalty-inducing-threshold objects and determine trajectory modification. The ability to run these behaviors accurately signifies that the AI can consistently complete test runs in the environment without failing in excess.

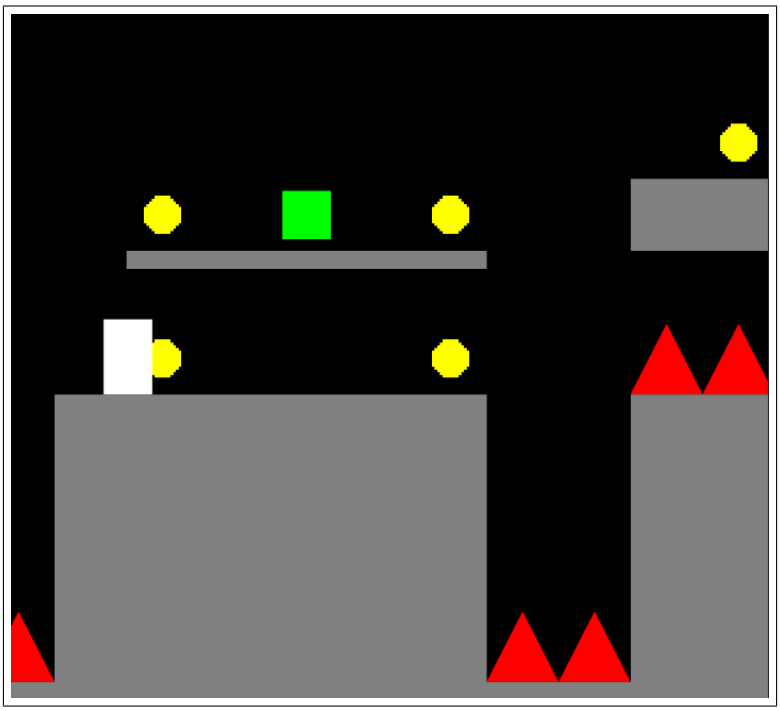

Figure 4.2 AI bot interacting collectible and obstacles.

4.1.3 Combat and Adaptive Behavior

As a consequence, the developmental capacities of AI now depend on movement and interaction with other worlds; combat mechanics have not been integrated into the reinforcement learning architecture as yet, and AI does not know how to perform attack sequences or enemy encounters. As a result, the present section does not contain any results for adversarial behavior or combat-related decision-making. The existing implementation continues to be focused on traversal and interactions with the environment as currently foundation lines for future use extensions in terms of functionalities.

4.2 Training Performance Metrics

4.2.1 The Reward Functions Performing

The Reinforcement learning algorithms have exhibited good and stable trends of reward acquisition from one training cycle to the next. History shows improvements in increments of reward as occurrences of suboptimal or penalty behavior decline. Thus, the agent has managed to utilize positive reinforcement, indicating the successful design of the reward function to facilitate movement and interaction optimally.

4.2.2 Convergence and Stability

Evidence of training convergence has been through stabilization in performance over many episodes. The first few phases of training revealed very high variance in agent behavior performance in making decisions and severe penalties. However, the learning model, after training over a sufficient number of epochs, has reduced the performance fluctuation, indicating stable learning and lesser dependence on exploratory randomness. The convergence of the policy indicates a successful imprinting of an optimal decision-making structure within the agent by the training process.

4.2.3 Evaluation of Training Efficiency

The evaluation of training efficiency was through policy improvement rate and reduction of superfluous movements. The learning curve of the AI clearly indicates the improvement as it refrained from making unnecessary movements early on before improving the strategy. Other training efficiencies have been enhanced by tuning hyperparameters and distributing rewards, making learning focused and effective in computational use.

4.3 AI Behavior in Environment training ground

4.3.1 Behavior Under Different Level Designs

Varieties of platform layouts offered opportunity to review generalization properties of the AI model. Test results illustrate the versatility of the trained policy across structurally similar environments. High navigation and interaction performances are sustained. The AI can adapt to different setups without retraining as long as the key level design principles stay within the threshold defined by the training environment.

4.3.2 Response to Dynamic Obstacles

The AI shows a degree of flexibility in interacting with moving obstacles or hazards in the environment, whereby learned avoidance allows the agent to detect and circumvent moving threats with adjustments to time and trajectory accordingly. The steady successful performance on avoidance would suggest that reinforcement learning has encoded a decisional process for hazard avoidance.

4.3.3 Efficacy in Collectible Acquisition

In tasks where collectible objects were tested in the environment, the AI was observed to optimize paths when seeking maximal reward. This reliance was mostly without detours as the agent preferred the path of least resistance toward collectible objects, following the decision patterns as defined by the reinforcement structure. Thus, the AI's consistent behavior in achieving successful item collection while avoiding hazards proves its risky behavior and reward calculations in a game.

4.4 Implementation and Integration of AI

4.4.1 AI Integration in the Unity Environment

The integrated AI model works fully in the Unity platformer game in real time. Transitioning from simulated training environments to real-time execution appeared completely uneventful, with the AI behaving exactly as it was supposed to, following the learned patterns from training data. This has been made possible by synchronization using the ML-Agents package in Unity so that the AI decision-making can continuously interact with the physics of the game and the game world.

4.4.2 AI Perception and a Dual-Render Environment Setup

The dual-render environment setup allows the AI perception system to distinguish between the in-game elements. It enables the agent to structure information on the environment so that the decision-making mechanism correctly identifies relevant objects, such as the platform, obstacles, and collectibles. A rendering-based system increases the spatial awareness of the AI and hence the contextually relevant actions that can be taken.

4.4.3 Modularity and Reusability in Implementation

The AI system is designed as a modular system to promote reusability across projects. Movement controllers, reward functions, and decision- making modules are key components that can be adjusted to new environments in the implementation without requiring extensive reconfiguration. Thus, this modular approach enhances the AI's suitability to even more varied platformer game projects, which are intended to become an AI reusable asset.

4.5 Summary of Current AI Capabilities

The AI system is now competent in the movement, navigation, and interaction of a platformer game. It has shown convergence in learning which is stable, effective awareness of its environment, and generalized behavior across different level designs. Although combat is not yet in use, the current implementation offers a firm groundwork for extended AI development in platformer games. The reinforcement-learning framework has fostered agent behavior that is conducive to the free navigation and interaction of game elements in a structured and goal-directed manner.

CHAPTER 5 CONCLUSIONS

This work presents a method encompassing the modeling of adaptive artificial intelligence (AI) agents for 2D platformers under a reinforcement learning paradigm, specifically PPO. By designing a flexible and extensible game environment in Unity, we allowed for the creation and training of Player and Enemy agents that behave intelligently and dynamically in complex scenarios.Through iterative training methods, the Player Agent was able to learn to traverse various levels, avoid hazards, and achieve goals. At the same time, the Enemy Agent developed methods of aggression to best or challenge the Player Agent. Our results illustrate that PPO is a viable option for training agents in environments with ever-changing discrete actions and delayed rewards. A further contribution of the work is reusable AI assets usable for disparate games or situations, presenting a reduction in time and effort for indie developers. Through the integration of machine learning tools with Unity's game engine environment and the establishment of modularity in environment design, we have given a start to scalable AI solutions for general application in game development.

5.1 Problems and Solutions

State your problems and how you fixed them.

5.2 Future Works

What could be done in the future to make your projects better.

REFERENCES

1. Unity Technologies, 2024, “ML-Agents Toolkit Documentation,” .
2. Vignesh Sriram, 2019, “Automated Playtesting in 2D Platformers Using Deep Reinforcement Learning and Curriculum Learning,” https://github.com/sriramr/AutoPlayRL.
3. Andreas Persson, 2005, “AI Techniques in 2D Platformers: Pathfinding, Image Recognition, and Line of Sight,” https://www.gamedev.net/tutorials/programming/artificial-intelligence/ai-techniques-for-2d-platformers-r3002/.
4. Thomas Smith, 2021, “Physics-Based Pathfinding in Platformer AI: Using A* and Platform Graphs for Navigation,” https://towardsdatascience.com/physics-based-pathfinding-in-2d-platformer-games-c5c6ecaa7f4f.

APPENDIX A

FIRST APPENDIX TITLE

Put appropriate topic here

This is where you put hardware circuit diagrams, detailed experimental data in tables or source codes, etc..

Figure A.1 This is the figure x11 https://www.google.com

This appendix describes two static allocation methods for fGn (or fBm) traffic. Here, λ and C are respectively the traffic arrival rate and the service rate per dimensionless time step. Their unit are converted to a physical time unit by multiplying the step size Δ. For a fBm self-similar traffic source, Norros [**?**] provides its EB as

$$C = \lambda + (\kappa(H)\sqrt{-2\ln\epsilon})^{1/H} a^{1/(2H)} x^{-(1-H)/H} \lambda^{1/(2H)} \tag{A.1}$$

where $\kappa(H) = H^H(1-H)^{(1-H)}$. Simplicity in the calculation is the attractive feature of (A.1).
The MVA technique developed in [**?**] so far provides the most accurate estimation of the loss probability compared to previous bandwidth allocation techniques according to simulation results. Consider a discrete-time queueing system with constant service rate C and input process λ_n with $\mathbb{E}\{\lambda_n\} = \lambda$ and $\mathrm{Var}\{\lambda_n\} = \sigma^2$. Define $X_n \equiv \sum_{k=1}^{n} \lambda_k - Cn$. The loss probability due to the MVA approach is given by

$$\varepsilon \approx \alpha e^{-m_x/2} \tag{A.2}$$

where

$$m_x = \min_{n \geq 0} \frac{((C-\lambda)n + B)^2}{\mathrm{Var}\{X_n\}} = \frac{((C-\lambda)n^* + B)^2}{\mathrm{Var}\{X_{n^*}\}} \tag{A.3}$$

and

$$\alpha = \frac{1}{\lambda\sqrt{2\pi\sigma^2}} \exp\left(\frac{(C-\lambda)^2}{2\sigma^2}\right) \int_C^\infty (r-C) \exp\left(\frac{(r-\lambda)^2}{2\sigma^2}\right) dr \tag{A.4}$$

For a given ε, we numerically solve for C that satisfies (A.2). Any search algorithm can be used to do the task. Here, the bisection method is used.
Next, we show how $\mathrm{Var}\{X_n\}$ can be determined. Let $C_\lambda(l)$ be the autocovariance function of λ_n. The MVA technique basically approximates the input process λ_n with a Gaussian process, which allows $\mathrm{Var}\{X_n\}$ to be represented by the autocovariance function. In particular, the variance of X_n can be expressed in terms of $C_\lambda(l)$ as

$$\mathrm{Var}\{X_n\} = nC_\lambda(0) + 2\sum_{l=1}^{n-1}(n-l)C_\lambda(l) \tag{A.5}$$

Therefore, $C_\lambda(l)$ must be known in the MVA technique, either by assuming specific traffic models or by off-line analysis in case of traces. In most practical situations, $C_\lambda(l)$ will not be known in advance, and an on-line measurement algorithm developed in [**?**] is required to jointly determine both n^* and m_x. For fGn traffic, $\mathrm{Var}\{X_n\}$ is equal to $\sigma^2 n^{2H}$, where $\sigma^2 = \mathrm{Var}\{\lambda_n\}$, and we can find the n^* that minimizes (A.3) directly. Although λ can be easily measured, it is not the case for σ^2 and H. Consequently, the MVA technique suffers from the need of prior knowledge traffic parameters.

APPENDIX B

SECOND APPENDIX TITLE

Put appropriate topic here

Figure B.1 This is the figure x11 https://www.google.com

Next, we show how $\mathrm{Var}\{X_n\}$ can be determined. Let $C_\lambda(l)$ be the autocovariance function of λ_n. The MVA technique basically approximates the input process λ_n with a Gaussian process, which allows $\mathrm{Var}\{X_n\}$ to be represented by the autocovariance function. In particular, the variance of X_n can be expressed in terms of $C_\lambda(l)$ as

$$\mathrm{Var}\{X_n\} = nC_\lambda(0) + 2\sum_{l=1}^{n-1}(n-l)C_\lambda(l) \tag{B.1}$$

Add more topic as you need

Therefore, $C_\lambda(l)$ must be known in the MVA technique, either by assuming specific traffic models or by off-line analysis in case of traces. In most practical situations, $C_\lambda(l)$ will not be known in advance, and an on-line measurement algorithm developed in [**?**] is required to jointly determine both n^* and m_x. For fGn traffic, $\mathrm{Var}\{X_n\}$ is equal to $\sigma^2 n^{2H}$, where $\sigma^2 = \mathrm{Var}\{\lambda_n\}$, and we can find the n^* that minimizes (A.3) directly. Although λ can be easily measured, it is not the case for σ^2 and H. Consequently, the MVA technique suffers from the need of prior knowledge traffic parameters.