ADAPTIVE ARTIFICIAL INTELLIGENCE AGENT FOR 2D PLATFORMER 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

Adaptive Artificial Intelligence Agent for 2D Platformer 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
..... (Mr. Naveed Sultan)	Committee Member
..... (Jaturon Harnsomburana, Ph.D.)	Committee Member

Project Title	Adaptive Artificial Intelligence Agent for 2D Platformer Game Development
Credits	3
Author(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

ในช่วงไม่กี่ปีที่ผ่านมา การประยุกต์ใช้ปัญญาประดิษฐ์ (AI) ในการพัฒนาเกมได้รับความสนใจอย่างมาก โดยเฉพาะในด้านการยกระดับประสบการณ์การเล่นเกมและการทดสอบระบบแบบอัตโนมัติ อย่างไรก็ตาม ระบบ AI ส่วนใหญ่ในปัจจุบันมักถูกออกแบบมาเพื่อใช้กับเกมหรือภารกิจเฉพาะเจาะจง ทำให้ขาดความยืดหยุ่นและไม่สามารถนำกลับมาใช้ซ้ำในโครงการอื่นได้อย่างมีประสิทธิภาพ งานวิจัยนี้นำเสนอการพัฒนาเอเจนต์ AI ที่สามารถปรับตัวได้สำหรับเกมแนวแพลตฟอร์มแบบ 2 มิติ โดยใช้เทคนิคการเรียนรู้เสริม (Reinforcement Learning) โดยเอเจนต์จะถูกฝึกในสภาพแวดล้อมที่พัฒนาบน Unity เพื่อเรียนรู้พฤติกรรมพื้นฐานของผู้เล่น เช่น การเคลื่อนที่ การกระโดด และการหลบหลีกสิ่งกีดขวาง ระบบนี้ใช้โครงข่ายนโยบายแบบ Proximal Policy Optimization (PPO) และออกแบบเชิงโมดูล เพื่อเพิ่มความยืดหยุ่น ขยายขอบเขตการใช้งาน และความสะดวกในการบูรณาการเข้ากับระบบอื่น โดยแยกกระบวนการตัดสินใจออกจากการดำเนินการ ทำให้สามารถต่อยอดเพื่อรองรับพฤติกรรมหรือกลไกเพิ่มเติมได้ง่ายขึ้น โมเดล AI ที่ผ่านการฝึกฝนแล้วจะถูกจัดทำเป็น Unity Asset ที่สามารถนำกลับมาใช้ซ้ำได้ในเกม 2 มิติที่มีลักษณะคล้ายกัน ไม่ว่าจะเป็นในบทบาทของตัวละคร NPC ที่มีความชาญฉลาด หรือเครื่องมือสำหรับการทดสอบเกมแบบอัตโนมัติ ซึ่งแนวทางนี้มีเป้าหมายเพื่อลดระยะเวลาในการพัฒนาและลดอุปสรรคทางเทคนิคสำหรับนักพัฒนาเกมอิสระที่ต้องการผสานพฤติกรรมอัจฉริยะเข้าสู่เกมของตน โดยผลการวิจัยแสดงให้เห็นถึงศักยภาพของการเรียนรู้เสริมในการสร้างเอเจนต์เกมที่สามารถปรับตัวได้ และสนับสนุนการเข้าถึงเครื่องมือ AI อย่างแพร่หลายในวงการพัฒนาเกม

คำสำคัญ: การพัฒนาเกม / ปัญญาประดิษฐ์ / ระบบศัตรูอัจฉริยะ / ทรัพยากรที่นำกลับมาใช้ได้ / ตัวละครที่ไม่ใช่ผู้เล่น / AI ที่ปรับตัวได้ / การเรียนรู้เสริม / การเรียนรู้แบบมีขั้นตอน / การฝึกสอน AI สำหรับเกม / ทรัพยากร AI

ACKNOWLEDGMENTS

This research project, Adaptive Artificial Intelligence Agent in Game Development, was conducted as part of the academic curriculum of the Department of Computer Engineering, King Mongkut's University of Technology Thonburi (KMUTT). The work presented in this report is the result of a collaborative effort between the authors, Mr. Intouch Yuso and Mr. Tanatanee Ponark, who have contributed equally to the research, design, and development of the system.

We would first like to express our deepest gratitude to our advisor, **Assoc. Prof. Natasha Dejdumrong, D. Tech. Sci.,** for her exceptional guidance, continuous support, and insightful feedback throughout the entire project. Her expertise in artificial intelligence and game development served as a critical foundation for our research. Her mentorship not only helped shape the technical scope of the project but also encouraged us to pursue innovative and practical solutions.

We would also like to extend our sincere thanks to our project committee members: **Asst. Prof. Nuttanart Muansuwan, Ph.D.,**, **Asst. Prof. Phond Phunchongharn, Ph.D.,** and **Dr. Jaturon Harnsomburana**, for their thoughtful evaluations, constructive criticisms, and valuable suggestions that greatly improved the overall quality of our work. Their diverse academic perspectives challenged us to think critically and approach the problem with greater depth and precision.

We are grateful to the Department of Computer Engineering at KMUTT for providing the necessary infrastructure, software tools, and academic support that facilitated our experimentation and development processes. The collaborative environment and access to technical resources played a vital role in enabling us to execute our ideas effectively.

Lastly, we would like to thank our families, friends, and classmates who offered their encouragement and moral support throughout the duration of the project. Their patience and motivation helped us stay focused and resilient, especially during difficult phases of development.To all who contributed to the success of this project directly or indirectly we offer our sincerest appreciation.

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, real-time control, and obstacle navigation, have remained a cornerstone of the video game industry. From early classics like Super Mario Bros. to modern indie titles such as Celeste and Hollow Knight, 2D platformers continue to thrive due to their accessible mechanics, expressive level design, and broad audience appeal. These games often serve as entry points for indie developers due to their design simplicity and well-established conventions.

A central component of platformer gameplay lies in the design of intelligent agents. Whether as playable characters or non-player characters (NPCs), these agents must exhibit responsive, believable behavior within dynamic environments. Traditionally, such behavior is implemented using finite state machines or behavior trees, rule-based systems that are straightforward to construct and maintain. However, these methods often produce rigid, predictable behavior that lacks adaptability in complex scenarios.

In contrast, reinforcement learning (RL), a subfield of machine learning, offers a more flexible and scalable approach to agent behavior. By learning through trial-and-error interaction with the environment, RL agents discover policies that maximize long-term cumulative rewards. This paradigm enables AI agents to adapt autonomously to changes in game conditions, mechanics, and objectives. RL has shown great potential in various domains such as robotics, autonomous navigation, and advanced gameplay systems.

Despite these advantages, the adoption of reinforcement learning in commercial game development remains limited. RL-based systems are typically confined to academic research or high-budget productions, primarily due to their steep learning curve, extensive training requirements, and lack of accessible tooling. Small studios and indie developers often lack the machine learning expertise, compute resources, and development time required to implement RL solutions effectively. Furthermore, most existing RL agents are purpose-built for specific games and environments, making them difficult to reuse or extend across projects.

This project aims to address these challenges by developing a modular, reusable AI framework tailored for 2D platformer games. The system is built within the Unity engine and leverages reinforcement learning to enable agents to perform essential platformer actions, such as walking, jumping, dashing, and obstacle avoidance, using a flexible, extensible architecture. The core design supports integration with external mechanics, custom environments, and retraining workflows

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

Artificial intelligence is a critical component of modern video games, influencing enemy behavior, environmental interactivity, and automated testing. Yet for many small-scale and independent game studios, building intelligent, adaptive AI remains a significant hurdle. Traditional methods such as finite state machines and behavior trees dominate game AI implementation due to their ease of use and predictability. However, these approaches offer limited adaptability and scalability, often resulting in scripted, repetitive behaviors that require manual tuning and maintenance.

Reinforcement learning presents a compelling alternative by enabling agents to learn optimal behaviors through experience rather than predefined rules. However, RL implementations are typically developed for

narrow use cases and tightly coupled to specific environments. This specialization limits their transferability, an agent trained for one game cannot easily be applied to another without extensive retraining or reengineering. This lack of generalization undermines RL's practical value for indie developers seeking flexible solutions that can adapt to different mechanics and level designs.

Furthermore, most existing RL toolkits and models are not designed with reusability in mind. They often lack modularity, documentation, or support for customization. Developers must invest considerable effort to integrate RL agents into existing projects, let alone extend them with new mechanics such as double-jumping or combat behavior.

This project seeks to address these limitations by delivering a reinforcement learning-based AI agent for 2D platformers that is not only pretrained but also modular, extensible, and easy to integrate. The agent learns to "solve" a platformer level by performing core gameplay actions, including traversal, object collection, and hazard avoidance, within a training environment constructed in Unity. Once trained, the model is exported as a Unity-compatible asset bundle that includes support components such as perception systems, action modules, and configuration files.

Importantly, the system is built to mimic the way human players generalize mechanical skills across games: a player who learns jumping and timing in one platformer can often apply that knowledge to another with similar mechanics. Likewise, this project aims to enable developers to reuse the pretrained agent across multiple games with comparable design principles, reducing training time, improving reliability, and supporting modular extension.

By lowering the barrier to entry for reinforcement learning in game development, this project provides a practical AI framework that supports not just a single application, but a family of games within a genre. It offers an efficient, scalable alternative to handcrafted AI systems for small teams working under constrained development resources.

1.3 Objectives

The primary goal of this project is to develop reusable, pretrained AI agents for 2D platformer games, delivered as a modular Unity asset that is easy to integrate, customize, and extend. The solution is designed to address the practical needs of indie developers by offering scalable AI functionality without requiring extensive machine learning expertise. The specific objectives are outlined as follows:

1.3.1 Train Adaptive AI Agents for Platformer Mechanics

- Train reinforcement learning (RL) agents capable of performing essential platformer behaviors, such as walking, jumping, dashing, and hazard avoidance, across varied level designs and gameplay scenarios.
- Enable the AI agents to adapt to different environments and tasks without requiring full retraining.

1.3.2 Develop a Modular and Reusable Unity Package

- Package the trained AI models and support systems into a Unity-compatible asset that can be easily imported into other projects.
- Design the architecture to be modular, allowing developers to add or remove functionality without modifying the core agent logic.

1.3.3 Support Customization and Scalability

- Implement an Action Module system that acts as a middleware layer, enabling developers to integrate their own control scripts with the pretrained agent through a standardized interface
- Ensure the system is scalable to support additional mechanics such as double-jumping or ranged attacks by adding modular components.

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

- Include thorough documentation covering setup, customization workflows, and integration steps for different use cases.
- Provide example scenes, tutorials, and retraining guides to support developers with varying levels of experience.

1.4 Scope of Work

This project is focused on the development of a reusable, pretrained AI framework for 2D platformer games using reinforcement learning techniques. The scope is defined by the genre, intended users, tools used, and practical boundaries that shape the project's direction.

1.4.1 Project Focus

The project is limited to the domain of 2D platformer games, chosen for their popularity, accessibility, and relevance to small and indie game development teams. The AI framework developed is intended to be integrated into Unity-based games, where agents must perform core platformer tasks such as movement, jumping, and hazard avoidance.

1.4.2 Development Scope

The scope includes the design of:

- A reinforcement learning-based AI agent capable of learning and performing platformer mechanics.
- A modular system for integrating trained agents into different Unity projects with minimal adjustments.
- A packaged Unity asset containing pretrained models, example environments, and documentation to support ease of use and further customization.

The project is designed with reusability, adaptability, and developer accessibility in mind, with a particular emphasis on supporting small game teams that may lack machine learning expertise.

1.4.3 Boundaries and Limitations

- The system is developed and tested solely for 2D platformer environments within Unity. Other genres or engines (e.g., 3D games, Unreal Engine) are not included in this scope.
- Only the player agent is trained in this phase. While the system is extensible to enemy agents, adversarial training is reserved for future work.
- The training process focuses on general platformer behavior rather than game-specific strategies, ensuring broad applicability at the expense of specialized behaviors.
- The reinforcement learning approach is based on the Proximal Policy Optimization (PPO) algorithm provided by the Unity ML-Agents toolkit. Alternative algorithms are not explored in this work.

This scope provides a practical and clearly bounded framework for delivering a usable AI solution, while acknowledging limitations that define the context of the project's outcomes.

1.5 Project Schedule

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

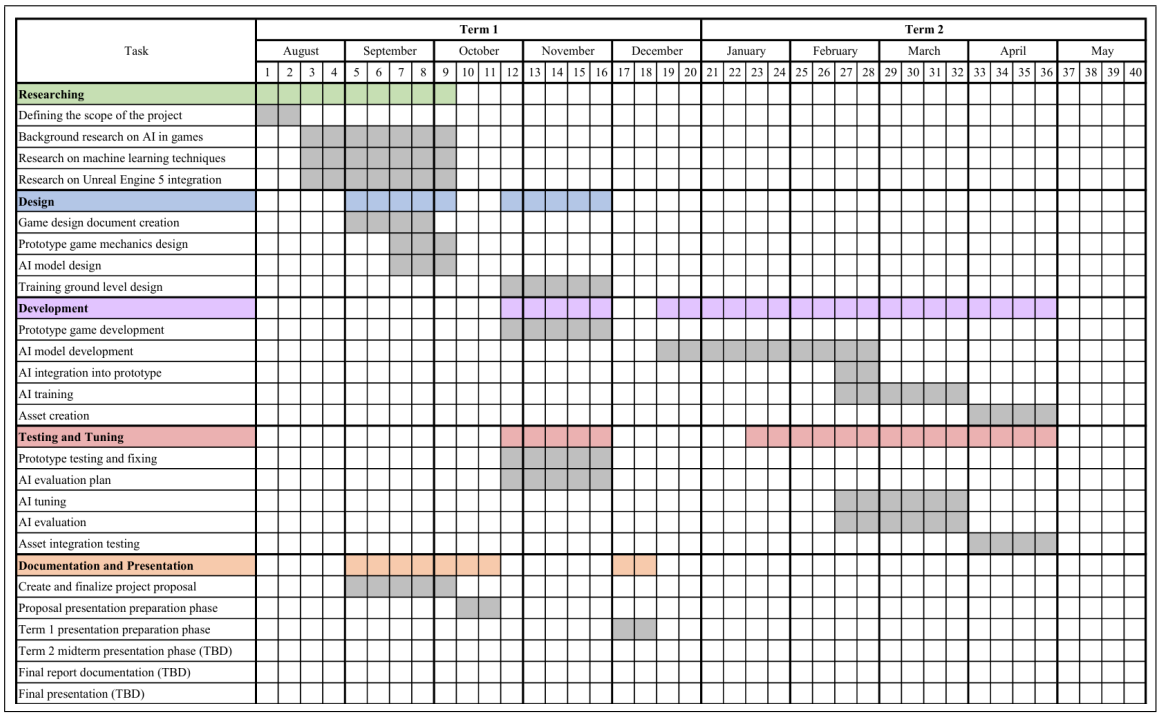


Figure 1.1 Gantt Chart

CHAPTER 2 BACKGROUND THEORY AND RELATED WORK

2.1 Introduction

This chapter presents the foundational theories and background knowledge required to understand the development of adaptive artificial intelligence (AI) agents within game environments. It covers essential machine learning concepts, particularly reinforcement learning, as well as the design principles relevant to interactive and dynamic gameplay. Additionally, the chapter introduces the development tools and frameworks used in this project, followed by a review of related research in the field. By examining existing AI methodologies and comparing them to the proposed system, this chapter highlights the technological context and motivation behind the project's design choices.

2.2 Theories and Core Concepts

2.2.1 Game Development

Game development is a multidisciplinary process involving design, art, programming, and testing to create interactive digital experiences. This project focuses on 2D platformer games, a genre characterized by structured level design, real-time movement, and timing-based mechanics. Understanding how games are developed provides essential context for designing AI agents capable of functioning effectively in such environments.

2.2.1.1 Game Development Lifecycle

The process of developing a game typically consists of several phases, each building upon the previous:

- **Pre-production:** This phase involves defining the game's concept, genre, mechanics, art style, and development scope. It also includes creating design documentation and planning technical requirements.
- **Production:** The core development phase where game assets and systems are built. This includes character controllers, UI, level design, animations, physics, and audio integration.
- **Testing:** Debugging, performance testing, and gameplay balancing. AI behavior, especially if it is not deterministic, must be evaluated for reliability and consistency across different playthroughs.
- **Post-production:** Final refinements, bug fixes, performance optimizations, and user feedback integration. This stage also includes publishing, updates, and maintenance.

Game development often involves teams of varying sizes, from large studios with specialized departments to small indie teams handling multiple roles.

2.2.1.2 Game Design

Game design defines the systems and elements that shape a player's experience. Core areas include:

- **Game Mechanics:** The rules and interactions that define gameplay, movement, physics, progression, scoring, and objectives.
- **Narrative Design:** Story elements that guide player engagement, from minimalistic setups to expansive, character-driven narratives.
- **Level Design:** Crafting environments that balance pacing, challenge, and progression. Good level design introduces new elements incrementally and encourages exploration.

- **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:** Menus, HUDs, buttons, feedback indicators, 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 Game Design in 2D Platformers

2D platformers are structured around character movement through levels composed of platforms, hazards, collectibles, and enemies. These games typically emphasize responsive controls, spatial awareness, timing, and feedback. Their mechanical clarity and visual simplicity make them ideal candidates for applying reinforcement learning techniques. Because platformers focus heavily on movement, navigation, and interaction with environmental objects, they provide a natural environment for testing AI performance in real-time gameplay contexts.

This genre is particularly relevant for this project because it is often chosen by indie developers due to its approachable development scope and strong potential for innovation. Solving AI challenges in this context can yield reusable and broadly applicable solutions.

2.2.1.4 Game Teseting and Automation

Game testing is critical to ensuring gameplay functionality, balance, and quality. It involves systematic evaluation of mechanics, interactions, and edge cases. Testing also plays a central role in identifying bugs, optimizing performance, and assessing the effectiveness of AI systems.

Traditionally, game testing is performed manually by QA testers or designers. This method is time-consuming and may not scale well with game complexity, especially for games with nonlinear progression, emergent behaviors, or multiple difficulty levels.

Automated testing tools have become increasingly useful, particularly for repeated simulation of gameplay scenarios. AI agents and scripted bots can be used to simulate playthroughs, identify unbalanced mechanics, and detect hard-to-find bugs. These agents also enable consistent data collection, such as completion time, failure points, or action frequencies, which supports more informed design decisions.

In recent years, machine learning has opened new possibilities for game testing. Reinforcement learning agents, for instance, can test levels by interacting adaptively with the environment, providing insight into player-like behavior. These methods are particularly valuable in open-ended or highly interactive games where traditional scripting approaches may fall short.

2.2.1.5 Challenges Faced by Developers

Game development, especially in small studios or solo efforts, presents several constraints:

- **Resource Limitations:** Small teams often lack access to advanced hardware, dedicated AI engineers, or large-scale testing capabilities.
- **Complexity of Design:** Creating balanced mechanics and scalable systems becomes increasingly difficult as gameplay depth increases.
- **Time Constraints:** Indie projects frequently operate under tight timelines, requiring efficient workflows and rapid iteration cycles.
- **Market Competition:** High competition in the indie game market makes polish, responsiveness, and innovation essential for visibility and success.

2.2.1.6 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[1]. 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*[2] and *Horizon Zero Dawn*[3] 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 the context of game development refers to the simulation of intelligent behavior in non-player characters (NPCs), enemies, companions, or dynamic game systems[4]. AI systems allow games to respond to player actions, create believable worlds, and maintain a sense of challenge and immersion. While AI in games does not aim for human-level intelligence, it is designed to enhance interactivity and gameplay engagement by imitating decision-making and behavioral patterns.

2.2.2.1 Roles of AI in Games

AI serves various roles depending on the type and genre of the game. Common roles include:

- **Enemy Behavior:** In action and platformer games, AI controls enemy movements, attack patterns, and responses to the player. It creates tension and challenge throughout gameplay.
- **Companions and Allies:** Some AI systems assist the player by following them, healing, or fighting alongside them. These behaviors often involve pathfinding and coordinated actions.
- **Environmental Dynamics:** AI can govern ambient world behaviors, such as wildlife, crowd flow, or weather systems, contributing to environmental immersion.
- **Procedural Content Generation:** AI can be used to dynamically generate terrains, levels, puzzles, or quests based on player progress and interaction.
- **Adaptive Difficulty and Personalization:** AI may adjust game difficulty based on the player's performance or learning curve to maintain balance and engagement.
- **Game Testing and QA:**AI agents can simulate player behaviors to help developers test game mechanics, uncover bugs, or evaluate gameplay balance.

2.2.2.2 Traditional Game AI Techniques

Historically, game AI has relied on rule-based systems due to their predictability, control, and ease of implementation. Common techniques include:

- **Finite State Machines (FSMs):** A widely used architecture where NPCs switch between states (e.g., idle, patrol, chase, attack) based on specific triggers or conditions. FSMs are simple to implement but become increasingly rigid and complex to maintain as behaviors grow.
- **Behavior Trees (BTs):** A hierarchical decision model where nodes represent conditions and actions. BTs offer improved modularity and scalability over FSMs and are common in modern AAA games.
- **Rule-Based Systems:** These use IF-THEN logic to determine actions based on predefined scenarios. Although they are transparent and interpretable, they are difficult to manage in dynamic or large-scale environments.
- **Pathfinding Algorithms:** Algorithms such as A* and Dijkstra's are core to NPC movement, enabling efficient navigation around obstacles in 2D and 3D environments.
- **Scripting and Heuristics:** Custom scripts or scoring systems that allow designers to fine-tune behavior based on game-specific priorities.

2.2.2.3 Limitations of Traditional Game AI

Despite their popularity, traditional AI approaches have several limitations:

- **Lack of Adaptability:** Rule-based agents cannot learn from the player's behavior or from the game world, resulting in repetitive and predictable responses.
- **Poor Scalability:** Expanding FSMs or behavior trees for large games with complex mechanics can lead to brittle and unmanageable codebases.
- **Limited Emergent Behavior:** Pre-scripted systems are unable to develop novel or surprising behaviors, reducing the depth of player-AI interactions.
- **High Manual Effort:** Fine-tuning traditional AI requires extensive design time, playtesting, and ongoing maintenance, especially as content evolves.

These limitations have fueled the growing interest in data-driven and learning-based approaches, such as machine learning and reinforcement learning, which allow AI agents to learn from experience and adapt to dynamic environments. These modern approaches, while more demanding in terms of computational resources and expertise, offer new possibilities for creating intelligent, responsive, and reusable game AI.

2.2.3 Machine Learning

Machine learning (ML) is a subfield of artificial intelligence that enables computer systems to learn patterns and make decisions from data without being explicitly programmed. Instead of using rule-based instructions, machine learning models improve their performance over time by learning from experience or data examples [5]. The core principle behind ML is that given enough data and the right algorithms, a system can learn to predict outcomes, recognize patterns, or take optimal actions.

Machine learning can be broadly categorized into three main types:

- **Supervised Learning:** Involves training a model on a labeled dataset, where the input-output pairs are known. The model learns to map inputs to outputs based on error minimization techniques. Common applications include image classification, sentiment analysis, and spam detection.
- **Unsupervised Learning:** The model is provided with unlabeled data and must identify hidden patterns or groupings. Techniques like clustering and dimensionality reduction fall into this category. Examples include customer segmentation and topic modeling.
- **Reinforcement Learning:** An agent learns to make decisions by interacting with an environment and receiving feedback in the form of rewards or penalties. Unlike supervised learning, the agent is not told what actions to take but must discover strategies that yield maximum cumulative reward through trial and error.

2.2.3.1 Core Components of a Machine Learning System

Across these paradigms, most ML systems share common components:

- **Features and Labels:** Features are input variables used by the model to make predictions. Labels represent the expected output (used in supervised learning).
- **Training and Inference:** Training refers to the phase where the model learns from data. Inference is the process of using the trained model to make predictions on new, unseen inputs.
- **Evaluation Metrics:** Accuracy, precision, recall, and F1-score are common metrics used to evaluate how well a model performs.

In the context of game development, reinforcement learning is particularly relevant because it maps naturally to sequential decision-making and dynamic environments. It enables agents to adapt without the need for hand-coded logic, which is especially valuable in large or interactive game worlds. [6].

2.2.4 Reinforcement Learning

Reinforcement Learning (RL) is a machine learning paradigm where an agent learns to make sequential decisions by interacting with an environment. At each time step, the agent observes the current state of the environment, chooses an action, and receives feedback in the form of a scalar reward. Through repeated interactions, the agent adjusts its behavior based on trial and error to maximize its cumulative reward. Over time, it develops a policy, a strategy for selecting actions based on environmental states, that improves decision-making performance across episodes [6].

Reinforcement learning problems are commonly modeled using a Markov Decision Process (MDP), which consists of the following components [7]:

- **States (S):** The set of all possible configurations of the environment, denoted $s \in S$.
- **Actions (A):** The set of all actions the agent can take, denoted $a \in A$.
- **Transition Probability (P):** A probability distribution $P(s's, a)$ that defines the likelihood of transitioning to a new state s' given the current state s and action a.
- **Reward Function (R):** A function $R(s, a, s')$ or simply $R(s)$, which defines the scalar feedback the agent receives after taking action a in state s and landing in state s'.
- **Policy (π):** A strategy or mapping from states to actions that the agent follows, $\pi(as)$, indicating the probability of taking action a given state s.

- **Value Function (V):** The expected cumulative reward that an agent can achieve starting from a particular state under a given policy.
- **Environment:** The external system or simulator that provides observations, accepts actions from the agent, and returns new states and rewards. It defines the dynamics and constraints of the learning problem.

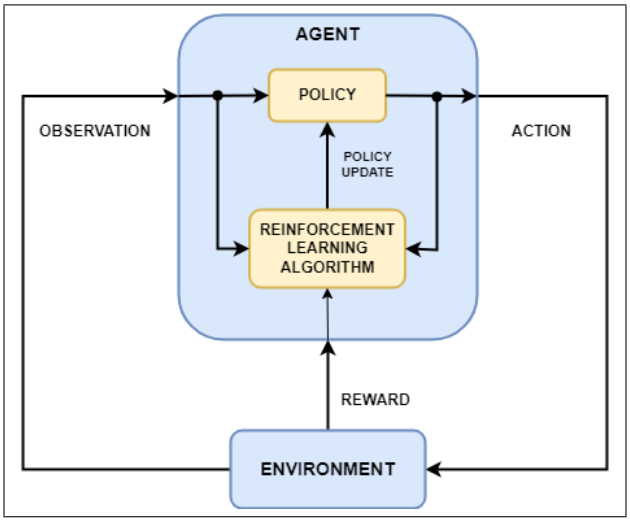


Figure 2.1 RL Process Diagram illustrating the interaction between the Agent and Environment.

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

As shown in Figure 2.1, the environment provides observations and rewards based on the agent's actions, completing the learning loop. The reinforcement learning algorithm updates the policy to favor actions that lead to higher expected rewards.

2.2.4.1 Reinforcement Learning in Video Games

In video games, RL enables agents to learn complex behaviors by trial and error, without the need for manually scripted rules. For instance, in a 2D platformer environment, the agent may learn to walk, jump, dash, avoid enemies, and collect items. The state might include the player's position, velocity, health status, or proximity to obstacles. Actions could involve movement directions or ability triggers, and rewards can be tied to reaching checkpoints, avoiding damage, or completing levels.

2.2.4.2 Temporal Credit Assignment

One unique aspect of RL is handling delayed rewards. Agents must learn to associate long-term outcomes with earlier decisions, a challenge known as the temporal credit assignment problem. For example, a jump executed early in a level may lead to avoiding death several seconds later, and the RL system must discover that causal link through repeated exploration.

2.2.4.3 Exploration vs. Exploitation

A fundamental challenge in RL is the trade-off between exploration (trying new actions to discover rewards) and exploitation (choosing known actions that yield high rewards). Effective training strategies balance these two aspects to ensure the agent continues to discover optimal behaviors without getting stuck in local minima.

2.2.4.4 Episode-Based Learning and Reset Logic

RL training typically occurs across episodes, bounded simulations of gameplay that end upon success, failure, or time limit. After each episode, the environment is reset, allowing the agent to explore new sequences and refine its strategy over time.

These characteristics make RL an ideal candidate for training adaptive game agents that can handle non-deterministic conditions and learn effective strategies without explicit programming.

2.3 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) is a reinforcement learning algorithm introduced by Schulman et al. [8] as a more efficient and practical alternative to earlier policy optimization methods such as Trust Region Policy Optimization (TRPO). PPO strikes a balance between theoretical robustness and implementation simplicity, making it highly suitable for complex and high-dimensional environments such as video games.

2.3.0.1 Policy Gradient Methods and TRPO

In standard policy gradient methods, the agent updates its policy by maximizing the expected return through gradient ascent. The surrogate objective for the policy gradient is given by:

$$\hat{g} = \mathbb{E}_t \left[\nabla_\theta \log \pi_\theta(a_t|s_t) \hat{A}_t \right] \tag{2.1}$$

Where:

- s_t: The state of the environment at time step t.
- a_t: The action taken by the agent at time step t.
- θ: The parameter vector of the policy network (e.g., weights in a neural network).
- $\hat{g}$: The estimated policy gradient. It determines the direction in which the policy parameters should be updated to improve expected rewards.
- $\mathbb{E}t[\cdot]$: Expectation taken over timesteps t. This indicates that the gradient is averaged over a batch of sampled trajectories or steps.
- ∇_θ: Gradient operator with respect to the policy parameters θ.
- $\pi_\theta(a_t|s_t)$: The stochastic policy. It defines the probability of taking action a_t given the current state s_t under the policy parameterized by θ.
- $\log \pi_\theta(a_t|s_t)$: The log-probability of selecting action a_t in state s_t. This is used because its gradient provides an unbiased estimate for the policy update.
- $\hat{A}_t$: The advantage estimate at time t, indicating how much better (or worse) an action is compared to the average performance at that state. A positive advantage encourages repeating the action, while a negative advantage discourages it.

While effective, naive gradient ascent may lead to excessively large updates that destabilize training. To address this, TRPO introduces a constraint based on the Kullback–Leibler (KL) divergence between the old and new policy distributions:

$$\max_\theta \ \mathbb{E}_t \left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \hat{A}_t \right] \quad \text{subject to} \quad \mathbb{E}_t \left[KL \left[\pi_{\theta_{\text{old}}}(\cdot \mid s_t) \,\|\, \pi_\theta(\cdot \mid s_t) \right] \right] \leq \delta \tag{2.2}$$

Where:

- θ: Current parameters of the policy being optimized.
- θ_{old}: Parameters of the policy before the update (used as a baseline for measuring change).
- $\pi_\theta(a_t \mid s_t)$: Probability of taking action a_t in state s_t under the current policy.
- $\pi_{\theta_{\text{old}}}(a_t \mid s_t)$: Probability of taking the same action under the old policy.
- $\hat{A}t$: Estimated advantage function at time step t, which represents how much better or worse the taken action was compared to the average.
- $\frac{\pi\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$: Importance sampling ratio (also known as the probability ratio), which corrects for the fact that data was collected under the old policy.
- $\mathbb{E}t[\cdot]$: Expectation over time steps or sampled trajectories.
- $KL[\cdot, |, \cdot]$: Kullback–Leibler divergence, a measure of how different two probability distributions are.
- $KL\left[\pi\theta_{\text{old}}(\cdot \mid s_t), |, \pi_\theta(\cdot \mid s_t)\right]$: KL divergence between the old policy and the new policy at a given state. This measures how much the new policy has changed relative to the old one.
- δ: A small positive threshold (hyperparameter) that bounds how much the new policy is allowed to deviate from the old policy. It serves as a trust region constraint.

This formulation ensures that the policy improvement step does not shift the agent's behavior too drastically, which helps maintain stable and reliable learning. Although effective, TRPO's reliance on second-order optimization and the need to compute the KL divergence make it relatively difficult to implement and compute-intensive.

2.3.0.2 Clipped Surrogate Objective

PPO simplifies the TRPO constraint by introducing a clipped surrogate objective that approximates trust region behavior without explicitly computing second-order derivatives. The clipped objective is defined as:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta)\hat{A}_t,\ \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t \right) \right] \tag{2.3}$$

Where:

- $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$: Probability ratio between the new and old policy.
- ϵ: Clipping threshold (typically 0.1 to 0.2) controlling how much the policy is allowed to change per update.

This objective penalizes updates that cause large policy shifts, effectively constraining updates within a small neighborhood of the old policy, thereby maintaining stability while encouraging learning progress.

2.3.0.3 Combined Objective and Additional Terms

The final PPO objective combines the clipped policy loss with additional components: a value function loss for accurate reward estimation and an entropy bonus to encourage exploration. The complete loss function is:

$$L^{\text{PPO}} = \mathbb{E}_t \left[L_t^{\text{CLIP}}(\theta) - c_1 L_t^{\text{VF}}(\theta) + c_2 S[\pi_\theta](s_t) \right] \tag{2.4}$$

Where:

- $L_t^{\text{CLIP}}(\theta)$: Clipped surrogate policy loss at time step t.
- $L_t^{\text{VF}}(\theta)$: Squared error between predicted and target value estimates (value loss).
- $S\pi_\theta$: Entropy of the policy at state s_t to promote exploration.
- c_1, c_2: Coefficients that control the weighting of the value loss and entropy bonus, respectively.

2.3.1 Advantages and Suitability for Game Environments

PPO has demonstrated high empirical performance in both discrete and continuous control problems, including benchmark environments like Atari and MuJoCo. Key advantages include:

- Simplicity of implementation using standard deep learning frameworks.
- High training stability and robustness across different hyperparameters.
- Suitability for environments with high-dimensional or stochastic observations.

In the context of this project, PPO is used to train adaptive agents for 2D platformer games, where actions such as jumping, dashing, and avoiding hazards require both temporal coordination and spatial awareness. PPO's clipping mechanism ensures that learning is stable and reliable, even in complex environments with diverse platforming mechanics and variable reward structures.

2.4 Development Tools

2.4.1 Unity

Unity is a widely-used game engine known for its support of both 2D and 3D game development, offering an extensive suite of tools for scene design, animation, physics, and cross-platform deployment. It provides a modular component-based architecture, enabling rapid prototyping and flexible gameplay development. Unity's support for custom scripting, asset import pipelines, and a rich ecosystem of packages makes it particularly well-suited for game AI experimentation and deployment. The Unity Editor also includes built-in visualization, profiler, and debugging tools that facilitate game iteration and performance tuning.

2.4.2 ML-Agents Toolkit

The Unity ML-Agents Toolkit is an open-source machine learning library designed to bridge Unity and modern reinforcement learning algorithms. It allows game developers to embed intelligent behavior into their games using machine learning, particularly reinforcement learning (RL). The toolkit includes:

- A Python training backend that integrates with libraries such as PyTorch for model training.
- A set of Unity-side C# components to define agents, behaviors, and sensors.
- Support for both visual and vector-based observations, discrete and continuous action spaces, and reward signal configuration.
- Export capabilities to convert trained models into ONNX format for inference inside Unity runtime.

The toolkit is highly extensible and supports training with multiple agents simultaneously in a shared or parallelized environment.

2.4.3 Python

Python is a high-level programming language widely used in scientific computing, machine learning, and automation. In this context, Python serves as the main language for configuring, executing, and managing the training process of reinforcement learning agents. Key reasons for its use include:

- Compatibility with ML-Agents' training pipeline.
- Integration with machine learning libraries such as PyTorch for neural network optimization.
- Access to data processing tools like NumPy and Pandas, which are useful for monitoring and evaluating model performance.
- TensorBoard support for visualizing training metrics, including rewards, episode length, and loss trends.

2.4.4 C#

C# is the primary programming language used within Unity for developing game logic, agent control scripts, UI systems, and custom components. Its integration into Unity's scripting API allows developers to:

- Control gameplay mechanics and interactions between objects.
- Define behavior parameters for ML agents, including state tracking, sensor input, and action execution.
- Implement physics-based movement, animation controllers, and in-game event handling.
- Interface with Unity's MonoBehaviour system to manage game loop events such as initialization, updates, and collisions.

C# provides a balance of performance and expressiveness, making it well-suited for real-time game development.

2.4.5 ONNX Runtime

ONNX (Open Neural Network Exchange) is an open standard format for representing machine learning models. After training reinforcement learning agents using the ML-Agents toolkit, the resulting models are exported in ONNX format. Unity supports ONNX model inference through the Barracuda inference engine, enabling:

- Embedding trained models directly into game builds for real-time AI decision-making.
- Hardware-agnostic execution across different platforms (Windows, macOS, WebGL, etc.).
- Seamless deployment without requiring Python or ML libraries at runtime.

ONNX provides a robust and portable way to integrate learning-based AI into production Unity games.

2.4.6 Development Environment

The development workflow combines multiple software tools to streamline programming, training, and testing:

- **Visual Studio:** Used for writing and debugging C# scripts in Unity.
- **Python Environment:** Managed through Anaconda or virtual environments to isolate dependencies required for ML-Agents.

- **Git:** Used for version control, backup, and collaborative development.
- **Unity Editor Tools:** Includes physics simulators, asset organization tools, and visual debugging aids.

These tools contribute to a structured and maintainable development pipeline from prototyping to deployment.

2.5 Related research

2.5.1 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)[9] 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.

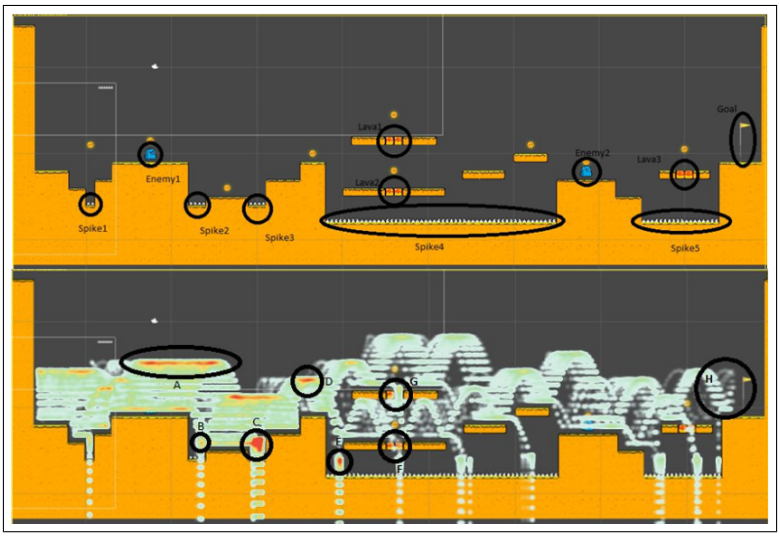


Figure 2.2 Unseen level and Unseen level with the agent heat map

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.

2.5.2 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) [10] 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.

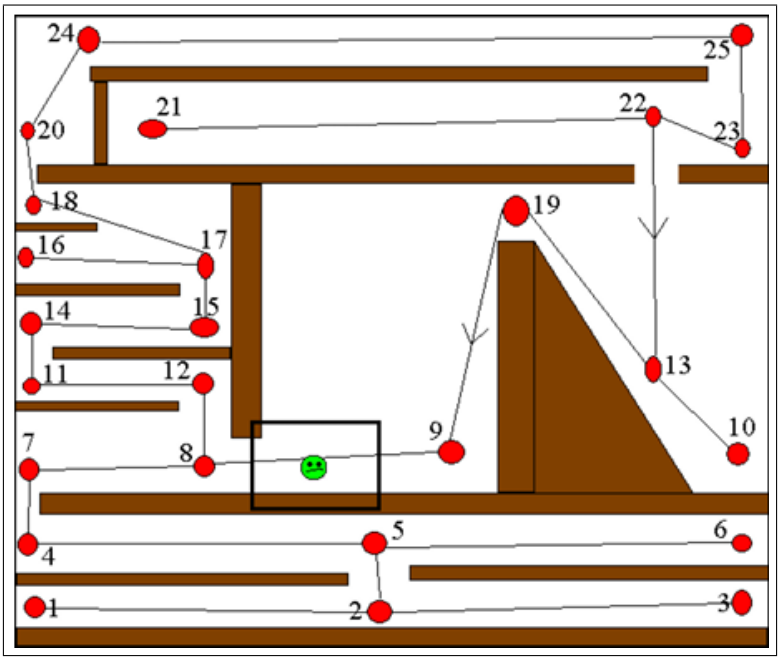


Figure 2.3 In a 2D platform game, pathfinding nodes are shown as ellipse with corresponding indexes and connections on the map.

Source: https://www.diva-portal.org/smash/get/diva2%3A4762/FULLTEXT01.pdf?utm

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.

2.5.3 Pathfinding and Navigation in Platformer AI

One main issue within platformer AI is making an intelligent movement deemed natural and responsive. Smith (2021)[11] 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.

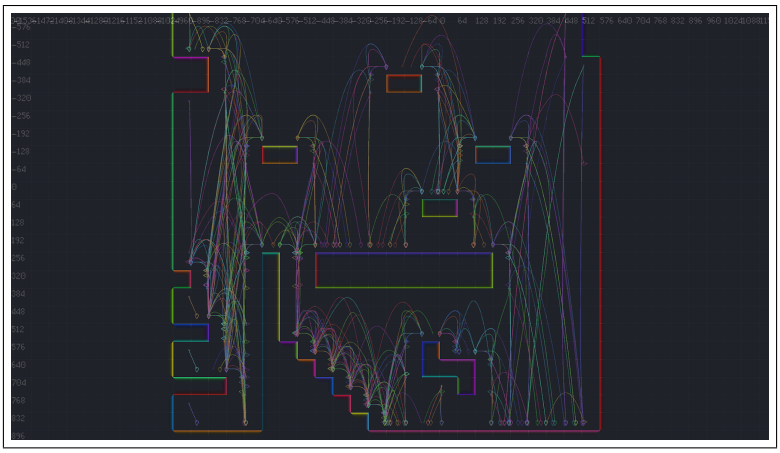


Figure 2.4 Using A* search to find path through the graph

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.

2.6 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.

2.7 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 the reinforcement learning algorithm.
- **Game Environment Design:** A description of the custom 2D platformer environment developed for training and evaluating AI behavior.
- **AI Agents Development:** The structure and modular design of the player agent, including its perception setup, action abstraction, and behavior controls.
- **AI Algorithms and Models:** The selection and application of Proximal Policy Optimization (PPO) as the learning algorithm.
- **Training Process and Optimization:** Training setup, reward shaping strategies, curriculum learning stages, and monitoring tools.
- **AI Integration and Packaging:** The deployment of the trained model into Unity for inference, along with steps for packaging and delivering it as a reusable Unity asset.

By following this methodology, the project aims to deliver a robust AI development framework that supports intelligent agent behavior, simplifies the adoption of reinforcement learning in indie projects, and lays the groundwork for future scalability and customization.

3.2 System Architecture

The architecture of this project is designed to support modular, trainable reinforcement learning agents within a Unity-based 2D platformer game. It integrates Unity ML-Agents, a dual-camera perception system, custom action middleware, and a structured feedback loop for learning. The primary goal is to provide a scalable and adaptable framework that can be reused across various platformer game scenarios. Figure 3.1 illustrates the complete learning system.

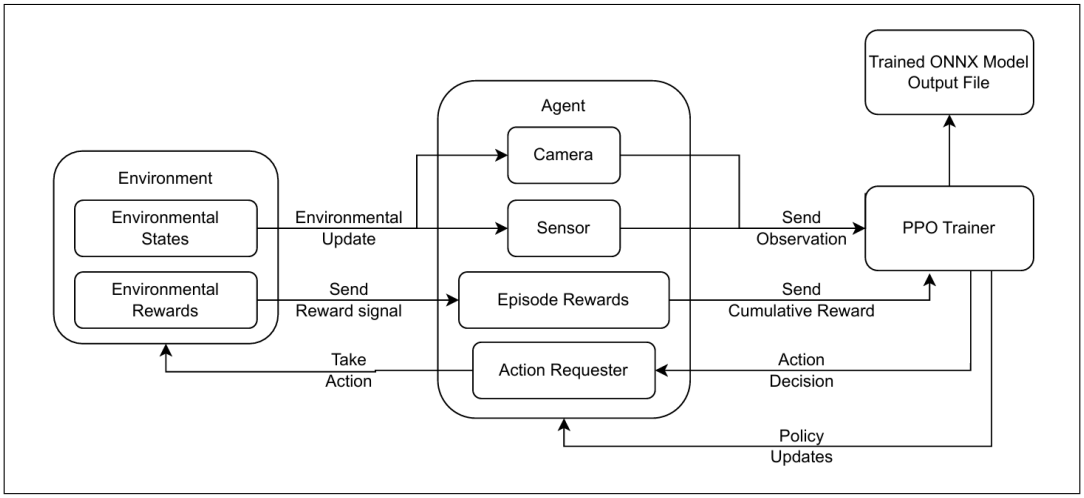


Figure 3.1 System Architecture and Reinforcement Learning Training Loop

3.2.1 Component Overview

- **Environment:** The game world is created using Unity's 2D tools, featuring platformer elements such as platforms, hazards, coins, and checkpoints. The environment continuously updates based on the agent's actions and physics rules.
- **Agent:** The agent acts as the decision-making entity. It receives environmental states or camera observations and interacts with the world through a policy network trained using PPO. Its action requests are routed through a modular Action Requester layer.
- **Sensor and Camera:** The agent receives observations via attached sensors. In visual training scenarios, a secondary camera renders a simplified, color-coded version of the environment exclusively for the agent. This visual feed is passed as input to the model to guide decision-making.
- **Action Requester:** This middleware abstracts the communication between the model's outputs and the actual in-game behaviors. It converts action indices into commands (e.g., jump, dash) and delegates them to appropriate character controller components. The character controller then executes the received commands, updating the agent's position, animation, and interaction within the game environment.
- **Reward System:** A reward handler observes events like reaching goals, collecting items, taking damage, or idling. These trigger scalar feedback signals that are sent back to the model to guide future learning.
- **Episode Feedback Loop:** The ML-Agents framework calculates the cumulative reward per episode. At the end of each episode, the agent's performance is logged, and training resets begin from a new randomized state.
- **PPO Trainer:** Training occurs on an external Python environment using Unity ML-Agents. The PPO trainer optimizes the policy parameters to improve the agent's decision-making over time.
- **ONNX Model Export:** Once trained, the resulting neural policy is exported in ONNX format and deployed back into Unity for inference. This makes the model reusable across different platformer environments without additional training.

3.2.2 System Highlights

This architecture emphasizes:

- Decoupled design for modularity and extensibility
- Real-time reward feedback for adaptive learning
- Dual-camera support for efficient agent perception
- Seamless transition between training and inference modes

The structured pipeline facilitates experimentation, debugging, and deployment, ensuring that the agent can be reused and adapted in other projects by indie developers or game designers without machine learning expertise.

3.3 Game Environment Design

The game environment constitutes the primary interactive space in which reinforcement learning (RL) agents are trained and evaluated. A well-structured environment is critical to effective agent learning, as it defines the sensory inputs, action possibilities, feedback mechanisms, and contextual challenges that guide behavioral adaptation. This section elaborates on the structure, components, and mechanics of the custom 2D platformer environment used for this study, with emphasis on level design, agent interaction capabilities, physics implementation, and system validation.

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.

3.3.1.1 Platforming and Terrain

The terrain system integrates a variety of platform types that influence agent decision-making and navigational strategies. Each terrain element contributes uniquely to the richness of the training experience:

- **Wall Surfaces:** Walls allow for interactions such as wall jumps or wall slides, affecting navigation strategies.
- **Solid Platforms:** Standard ground elements on which the agent can walk and jump.

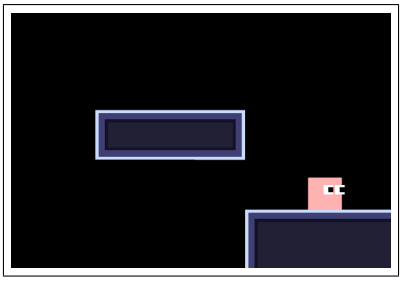


Figure 3.2 The image shows solid wall and platform.

- **One-Way Platforms:** Platforms that can be landed on from above but allow the agent to jump through from below.

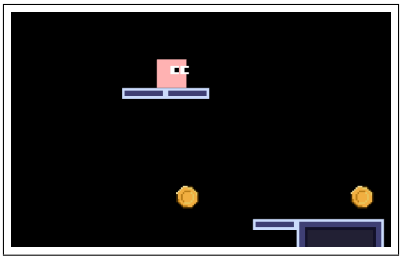


Figure 3.3 The image highlights one-way platforms.

- **Moving Platforms:** Dynamic platforms that require precise timing and positioning.

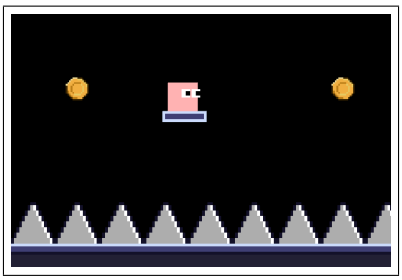


Figure 3.4 The image highlights moving platforms over obstacle.

3.3.1.2 Interactive Elements

To enrich the environment's interactivity and encourage exploration, a range of functional objects are distributed throughout the level:

- **Goals:** Objects such as coins or checkpoints that act as positive reinforcement signals for the agent.

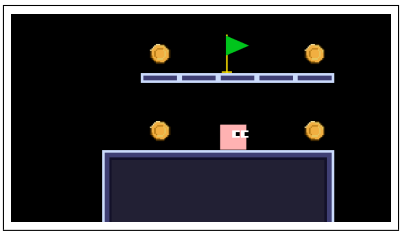


Figure 3.5 The image highlights coins and checkpoint as goals.

- **Hazards:** Spikes, traps, or pits that impose penalties and help the agent learn avoidance behaviors.

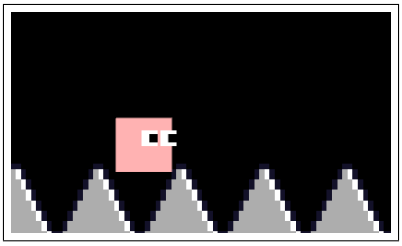


Figure 3.6 The image highlights player falling into hazard.

- **Enemies:** Opponents with basic AI that introduce combat-related decisions and temporal awareness.

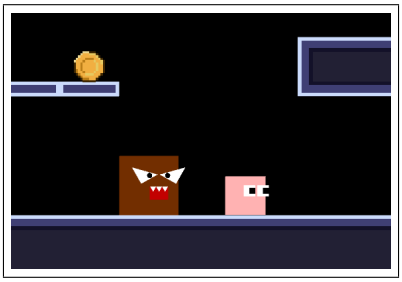


Figure 3.7 The image highlights enemy chasing player.

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

Figure 3.8 Environment Training Ground for the AI

3.3.2 Agent Actions

Table **??** 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 movement and interaction mechanics of the game are governed by Unity's built-in 2D physics engine, facilitated by the Rigidbody2D component and customized through script-based motion logic. This configuration enables realistic yet controllable movement responses, ensuring that the AI agent operates under consistent physical constraints. These constraints are crucial for promoting transferable behavior and narrowing the simulation-to-reality gap in reinforcement learning.

3.3.3.1 Movement Mechanics

Movement in the platformer environment is handled through a centralized input-processing system defined in the `HandleInput()` method of the `BasicPlayerMovement.cs` script. This function maps user or agent decisions into stateful input variables that drive the player's actions across update cycles. Input is only processed when `inputEnabled` is set to true, ensuring synchronization between AI control and gameplay flow.

```
1  private void HandleInput()
2  {
3    if (!inputEnabled) return;
4
5    moveInput = new Vector2(Input.GetAxisRaw("Horizontal"), Input.GetAxisRaw("Vertical")
         );
6    moveDir = moveInput.x;
7    dropInput = moveInput.y;
8    jumpInput = Input.GetKeyDown(KeyCode.Space);
9    dashInput = Input.GetKeyDown(KeyCode.LeftShift);
10
11   if (jumpInput)
12   {
13     Jump();
14   }
15
16   if (dashInput)
17   {
18     Dash();
19   }
20 }
```

This method collects directional input using Unity's built-in `Input.GetAxisRaw()` system and stores it in the `moveDir` and `dropInput` variables. Action-based inputs such as jumping and dashing are mapped to boolean flags, `jumpInput` and `dashInput`, respectively. These are then passed to their corresponding methods (`Jump()` and `Dash()`) to initiate action events. This separation of input and behavior logic allows the system to be easily extended for reinforcement learning agents, which can override these inputs through scripted or neural decisions.

- **Walking and Running:** Horizontal movement is applied using an acceleration-based force model. As shown in the `HandleMovement()` method which takes in one argument, `direction`, the lateral velocity is updated using player input and smoothed via velocity power with the following code snippet.

```
private void HandleMovement(float direction)
{
  if (IsDashing) return;

  if (playerAttack == null || !playerAttack.isAttacking)
  {
    if (direction > 0 && !IsFacingRight)
      Flip();
    else if (direction < 0 && IsFacingRight)
      Flip();
  }

  float targetSpeed = direction * moveSpeed;
  float speedDifference = targetSpeed - rb.linearVelocity.x;
  float accelRate = (Mathf.Abs(targetSpeed) > 0.01f) ? acceleration :
      decceleration;
  float movement = Mathf.Pow(Mathf.Abs(speedDifference) * accelRate, velPower) *
      Mathf.Sign(speedDifference);

  if (!IsWallJumping)
  {
    rb.AddForce(movement * Vector2.right);
  }
  else
  {
    rb.linearVelocity = Vector2.Lerp(rb.linearVelocity, (new Vector2(direction *
        moveSpeed, rb.linearVelocity.y)), wallJumpLerp * Time.deltaTime);
  }
}
```

- **Jumping:** The `PerformJump()` and `HandleJump()` methods handle vertical jumping, including coyote time and jump buffering. When a jump is triggered, vertical velocity is reset and overridden:

```
private void HandleJump()
{
  canJump = CanJump();
  canWallJump = CanWallJump();

  if (Time.time - lastJumpPressedTime <= jumpBufferTime)
  {
    if (canJump && !hasJumped)
    {
      PerformJump();
      hasJumped = true;
    }
    else if (canWallJump && !hasWallJumped)
    {
      wallJumpDir = (IsOnRightWall) ? -1 : 1;
      PerformWallJump(wallJumpDir);
```

```
      hasWallJumped = true;
    }
  }
}
```

```
private void PerformJump()
{
  lastGroundedTime = 0;
  lastJumpPressedTime = 0;
  lastJumpTime = Time.time;
  rb.linearVelocity = new Vector2(rb.linearVelocity.x, 0f);
  rb.linearVelocity = new Vector2(rb.linearVelocity.x, jumpForce);
  isJumping = true;
}
```

This guarantees consistent jump height and reduces variance in vertical behavior, which is beneficial for stable learning outcomes.

- **Dashing:** Dash behavior is implemented via a coroutine in `PerformDash()`, where gravity is temporarily disabled and horizontal velocity is set explicitly which is handled by `HandleDash()`:

```
private void HandleDash()
{
  if (CanDash() && lastDashPressedTime > lastDashTime)
  {
    StartCoroutine(PerformDash());
  }
}
```

```
private IEnumerator PerformDash()
{
  isDashing = true;
  lastDashPressedTime = Time.time;
  dashDirection = IsFacingRight ? Vector2.right : Vector2.left;
  rb.gravityScale = 0;
  rb.linearVelocity = dashDirection * dashSpeed;
  yield return new WaitForSeconds(dashDuration);
  rb.gravityScale = gravityScale;
  isDashing = false;
  lastDashTime = Time.time;
}
```

This mechanic enables burst-based directional movement, adding timing complexity and escape strategies to the agent's behavior model.

- **Wall Interactions:** Wall detection and sliding are handled in `HandleWallDetection()` and `HandleWallSlide()`. When the agent is airborne and collides with a vertical surface, its vertical descent is controlled:

```
private void HandleWallSlide()
{
  if (IsOnWall && !IsGrounded && rb.linearVelocity.y < 0)
  {
    rb.linearVelocity = new Vector2(rb.linearVelocity.x, -wallSlideSpeed);
  }
}
```

In addition, wall jumping is enabled through the `PerformWallJump()` method. When the agent is near a wall and a jump input is received, a directional impulse is applied, launching the agent away from the wall. The direction is based on wall contact (left or right) and is calculated as follows:

```
private void PerformWallJump(int dir)
{
  if (isWallJumping || hasWallJumped) return;

  lastGroundedTime = 0;
  lastJumpPressedTime = 0;
  rb.linearVelocity = Vector2.zero;
  Vector2 force = new Vector2(wallJumpForce.x, wallJumpForce.y);
  force.x *= dir;

  if (Mathf.Sign(rb.linearVelocity.x) != Mathf.Sign(force.x))
    force.x -= rb.linearVelocity.x;

  if (rb.linearVelocity.y < 0)
  {
    force.y -= rb.linearVelocity.y;
  }

  rb.AddForce(force, ForceMode2D.Impulse);
  isWallJumping = true;
  hasWallJumped = true;
}
```

This mechanic increases vertical navigation flexibility and enables the agent to escape corners or ascend narrow shafts, enriching both player and AI-controlled movement strategies.

3.3.3.2 Collision and Interaction System

Unity's physics engine and collider components are used to detect and manage interactions between the agent and the environment. These interactions are monitored continuously and affect the agent's internal state and eligibility to perform actions.

- **Ground and Wall Detection:** Performed via a box collider in `HandleGrounded()` and `HandleWallDetection`, these checks are used to determine whether the agent can initiate jumps, wall-jumps, or dash. The logic employs `Physics2D.OverlapBox()` and updates state variables accordingly:

```
private void HandleGrounded()
{
  isGrounded = Physics2D.OverlapBox(_groundCheckPoint.position, _groundCheckSize,
       0, _groundLayer) != null;

  if (isGrounded && !wasGrounded)
  {
    hasJumped = false;
    hasWallJumped = false;
    isJumping = false;
    isWallJumping = false;
  }

  if (isGrounded && rb.linearVelocityY <= 0)
  {
    lastGroundedTime = Time.time;
  }

  wasGrounded = isGrounded;
}
```

```
private void HandleWallDetection()
{
  if (IsGrounded)
```

```
    {
      isOnWall = isOnLeftWall = isOnRightWall = false;
      return;
    }

    Transform leftCheck = IsFacingRight ? _wallCheckPointLeft :
        _wallCheckPointRight;
    Transform rightCheck = IsFacingRight ? _wallCheckPointRight :
        _wallCheckPointLeft;

    isOnLeftWall = Physics2D.OverlapBox(leftCheck.position, _wallCheckSize, 0,
        _wallLayer) != null;
    isOnRightWall = Physics2D.OverlapBox(rightCheck.position, _wallCheckSize, 0,
        _wallLayer) != null;

    isOnWall = IsOnLeftWall || IsOnRightWall;
    isWallJumping = false;
    hasWallJumped = false;
}
```

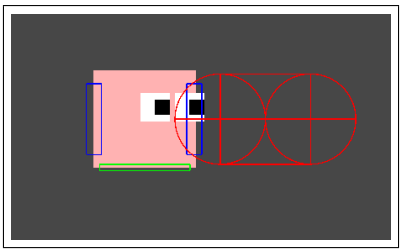


Figure 3.9 Collision detection and attack zone used in the player movement system.

- **Hazard Detection:** hazard detection is structured to operate via layer-based triggers and collisions (e.g., spikes or traps). These interactions would typically result in episode termination or reward penalties, and can be expanded through `OnCollisionEnter2D()` or `OnTriggerEnter2D()` events.

```
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.CompareTag("Hazard"))
    {
  Debug.Log("Player hit the spike (trigger)!");
  TakeDamage(maxHealth);
    }
}
```

- **Enemy Interaction:** Enemy detection and combat interactions are facilitated through the `PlayerAttack` script. These interactions allow for additional agent decision-making complexity, such as whether to evade or engage an enemy.

To further refine physical realism and training consistency, additional systems manage vertical acceleration and friction:

- **Custom Gravity:** The `ApplyCustomGravity()` method applies varying gravity multipliers depending on the jump state and vertical velocity. This produces more nuanced motion:

```
rb.gravityScale = gravityScale * fallMultiplier;
```

- **Friction Handling:** Friction is manually applied when movement input ceases, reducing velocity over time using impulses:

```
rb.AddForce(Vector2.right * -amount, ForceMode2D.Impulse);
```

Through the combination of Unity physics components and finely tuned motion scripting, the agent operates within a physically constrained yet expressive movement model. This model serves as the foundation for learning transferable locomotion policies across varied level designs.

3.3.4 Curriculum Learning Stages

To improve training efficiency and agent generalization, a curriculum learning strategy is implemented. The idea is to start with simpler tasks and environments, gradually increasing complexity as the agent gains proficiency. This approach enables the reinforcement learning agent to learn fundamental behaviors incrementally rather than attempting to master all mechanics at once. The curriculum consists of four distinct stages, each introducing new gameplay concepts while building on previously acquired skills.

3.3.4.1 Stage 1 – Basic Movement and Item Collection

The initial stage introduces flat, single-layer environments designed to teach basic movement and reward-seeking behavior. The agent learns to:

- Move horizontally toward visible goals.
- Collect coins or other positive-reward objects.
- Avoid hazards such as spikes (present since this stage).

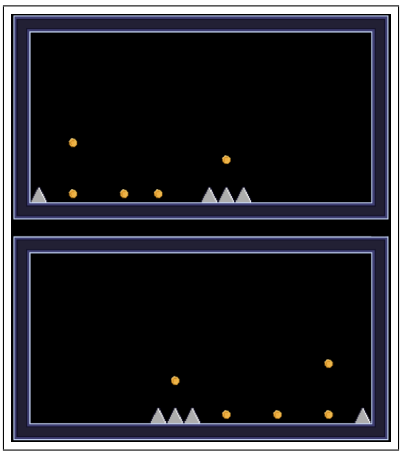


Figure 3.10 Stage 1 – Basic Movement and Item Collection

3.3.4.2 Stage 2 – Advanced Navigation (Multi-layer Platforms)

Once the agent consistently performs basic movement, multi-level environments are introduced. These levels feature vertical traversal requiring jumping. The agent learns to:

- Jump to elevated platforms and move vertically through the level.
- Optimize paths through layered terrain to reach rewards.

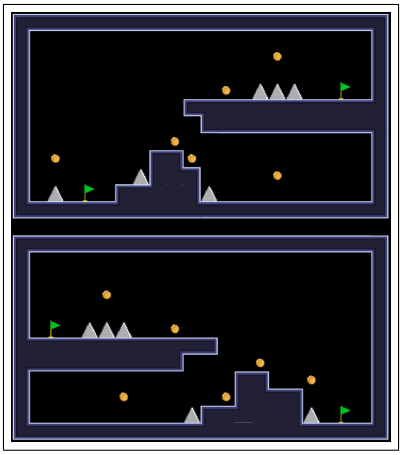


Figure 3.11 Stage 2 – Advanced Navigation with Multi-layered Platforms

3.3.4.3 Stage 3 – One-Way Platform Introduction

One-way platforms are added to introduce a new navigational rule. These platforms can be jumped onto from below and dropped through from above. The agent must learn to:

- Recognize one-way platforms as passable terrain.
- Drop down through platforms when needed to reach goals or avoid hazards.
- Adapt navigation strategy dynamically based on vertical positioning.

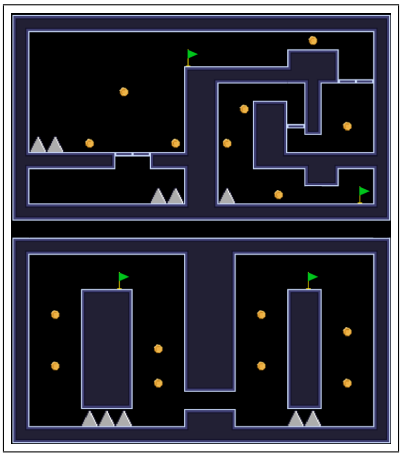


Figure 3.12 Stage 3 – Introduction of One-Way Platforms

3.3.4.4 Stage 4 – Enemy Introduction and Combat Awareness

In the final curriculum stage, basic enemy characters are introduced. These enemies follow simple state machine behavior (e.g., patrolling), and the agent must engage with or avoid them. New learning objectives include:

- Attacking enemies using melee actions.
- Avoiding enemy attacks and contact.
- Prioritizing combat versus avoidance depending on context.

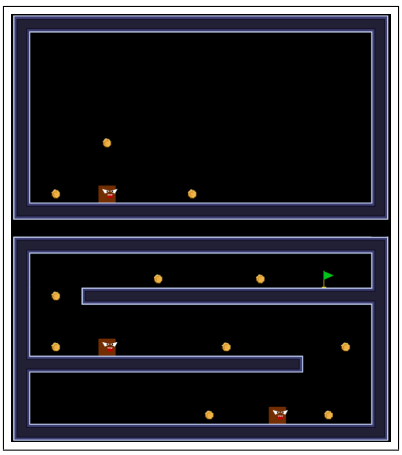


Figure 3.13 Stage 4 – Enemy Introduction and Combat Interaction

3.3.4.5 Full-Scale Platformer Environment

After completing all curriculum stages, the agent is transitioned to a large, hand-crafted platformer level designed to emulate real-world gameplay. This map includes all previously introduced mechanics—multi-layered platforms, one-way platforms, collectibles, hazards, and enemies—distributed throughout a complex and varied layout. This final training/testing stage serves to:

- Evaluate the agent's ability to generalize behaviors learned from smaller curriculum stages.
- Assess performance in realistic, high-difficulty scenarios involving dynamic decision-making.
- Train on longer episodes with diverse routes, reward opportunities, and hazards.

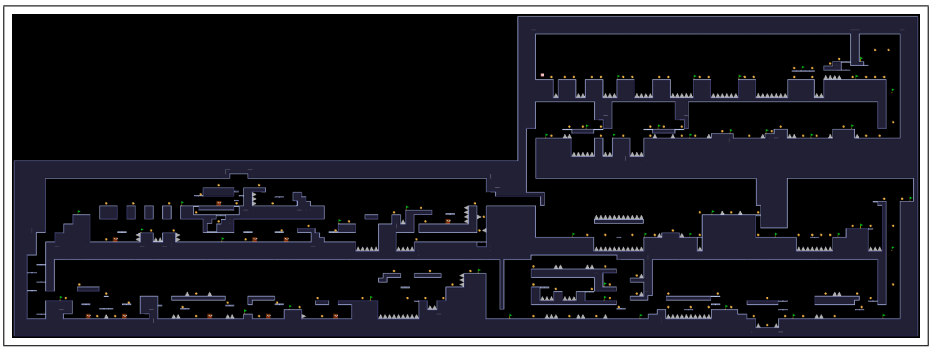


Figure 3.14 Full-Scale Environment for Final Training and Evaluation

This full environment is used for both extended training (with reward shaping) and testing the transferability of the agent's skills. The layout simulates the kind of level design seen in commercial 2D platformer games, making it ideal for benchmarking the agent's practical capabilities.

3.3.5 Environment Testing

Prior to integrating the reinforcement learning model, the platformer environment underwent systematic testing to ensure that it met the functional, performance, and scalability requirements necessary for stable agent training. These tests were essential for validating gameplay mechanics, maintaining frame rate stability, and ensuring compatibility with machine learning pipelines.

3.3.5.1 Functional Testing

Functional testing focused on verifying the correctness of the core gameplay systems. Using Unity's play mode, scene inspector tools, and in-editor debugging outputs, the following aspects were evaluated:

- **Movement Tests:** Confirmed that the agent could perform essential actions such as walking, jumping, dashing, and dropping through platforms, using the logic defined in the `BasicPlayerMovement` script.
- **Collision Tests:** Verified accurate collision detection with ground surfaces, hazards, enemies, and walls using Unity's `Physics2D.OverlapBox()`.
- **Camera Tests:** Ensured the secondary ML-rendering camera captured a simplified, texture-free version of the environment to serve as the agent's observation input.

3.3.5.2 Scalability Testing

To ensure the environment's reusability across future projects and training scenarios, a series of scalability tests were conducted:

- **Level Size Variability:** Different map sizes and segment arrangements were tested to validate agent adaptability across layouts.
- **Enemy and Obstacle Density:** The performance and behavior of agents were evaluated under varying densities of enemies and environmental hazards.
- **Generalization Potential:** Agents were tested in unseen level variants to assess their ability to transfer learned behaviors beyond a single map configuration.

This environment not only provides a stable foundation for reinforcement learning but also supports our broader goal of creating reusable AI agents capable of generalizing across platformer-level variants.

3.4 AI Agents Development

The artificial intelligence system developed for this project leverages reinforcement learning (RL) to train agents capable of navigating and interacting with a 2D platformer environment. Implemented using Unity ML-Agents, the AI model learns through trial-and-error to execute gameplay behaviors such as jumping, attacking, and collecting objectives. This section details the core aspects of the AI system, including the action space, observation model, reward system, and the algorithm used to drive learning.

3.4.1 Action Space and Decision-Making

The action space defines the set of discrete and continuous decisions the agent can make at any given time. These actions are interpreted and executed through the `PlayerActionModules` system, which acts as a modular interface between the AI model and the character controller components. The decision-making is handled inside `AgentController.cs` via the `OnActionReceived()` method:

```
public override void OnActionReceived(ActionBuffers actions)
{
  float moveX = Mathf.Clamp(actions.ContinuousActions[0], -1f, 1f);
  bool jumpAction = actions.DiscreteActions[0] == 1;
  bool dashAction = actions.DiscreteActions[1] == 1;
  bool attackAction = actions.DiscreteActions[2] == 1;
  bool dropAction = actions.DiscreteActions[3] == 1;

  playerActionModules.Move(moveX);
  if (jumpAction)
```

```
    {
      AddReward(rewardConfigSO.jumpPenalty);
      playerActionModules.Jump();
      jumpCount++;
    }
    if (dashAction)
    {
      AddReward(rewardConfigSO.dashPenalty);
      playerActionModules.Dash();
      dashCount++;
    }
    if (attackAction) playerActionModules.Attack();
    if (dropAction) playerActionModules.Drop();

    EvaluateRewards();
}
```

The following action set is available to the Player Agent: These actions are abstracted through `PlayerActionModules.cs`,

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's canJump state is `true`.
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.

allowing the AI to activate specific movement or combat features through calls like `Jump()`, `Dash()`, and `Attack()`.

3.4.2 Modular Action Handling via Action Modules

In reinforcement learning systems integrated into video games, the agent's decision-making logic must ultimately be translated into real-time gameplay actions. To maintain modularity and scalability, this project introduces an abstraction layer called the Action Module, implemented in the `PlayerActionModules.cs` script. As shown in figure 3.15 this component acts as a bridge between the `AgentController` and the character's mechanical systems, such as movement and combat.

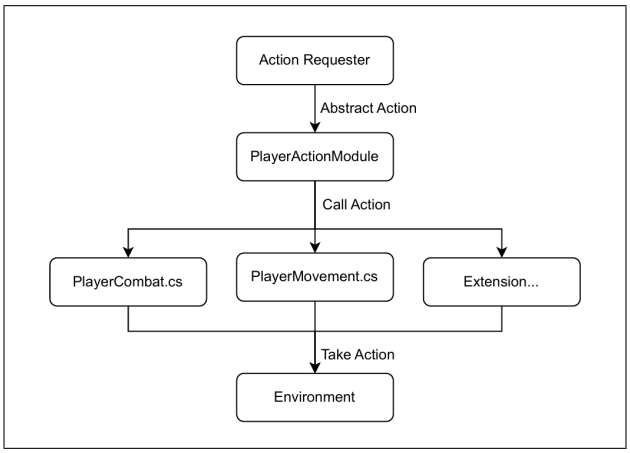


Figure 3.15 ActionModule acts as a middleware linking AgentController decisions to gameplay subsystems such as movement and combat.

3.4.2.1 Purpose and Motivation

Rather than embedding low-level movement or attack logic within the agent class itself, `PlayerActionModules` exposes high-level behavior functions like `Move()`, `Jump()`, `Dash()`, `Drop()`, and `Attack()`. These functions internally route commands to specialized scripts like `BasicPlayerMovement` and `PlayerAttack`. This separation offers several advantages:

- **Modularity:** The AI logic remains independent of how each action is technically executed, enabling clean separation of concerns.
- **Reusability:** The Action Module system can be reused across different characters or AI types with minimal code duplication.
- **Scalability:** Additional abilities, such as ranged attacks, double jumps, or new movement types, can be integrated by extending this module without modifying the AgentController or retraining from scratch.

3.4.2.2 Implementation Structure

Each public method in `PlayerActionModules` is responsible for delegating a discrete action:

```
public void Move(float direction)
{
  if (basicPlayerMovement != null)
    basicPlayerMovement.moveDir = direction;
}

public void Jump()
{
  if (basicPlayerMovement != null)
    basicPlayerMovement.Jump();
}
```

Likewise, the Dash, Attack, and Drop methods call respective functions in their subsystems, maintaining a centralized and standardized interface for decision execution.
In `AgentController.cs`, these methods are invoked after the action decision is received from the RL model:

```
playerActionModules.Move(moveX);
if (jumpAction)
{
  AddReward(rewardConfigSO.jumpPenalty);
  playerActionModules.Jump();
  jumpCount++;
}
```

This design ensures that all behavior can be routed through a consistent middleware, simplifying debugging, control testing, and future development.

3.4.2.3 Extensibility for New Actions

By adhering to this modular interface pattern, developers can create and register new actions by:

- Adding a new method to PlayerActionModules (e.g., `PerformSpecialAttack()`)
- Updating the AgentController to include a new discrete action channel
- Binding the action within ML-Agents via the Action Spec definition

This method reduces integration complexity and supports the project's broader goal: producing a reusable, extensible AI system for platformer games.

3.4.3 Agent Perception and Observations

The agent receives a series of structured numerical inputs via the CollectObservations() method. These observations are used by the neural network to infer context and predict optimal actions.

3.4.3.1 State Observations

To enable effective decision-making, the reinforcement learning agent relies on a structured observation vector that captures both its internal status and external context within the environment. These observations are collected in the `CollectObservations()` method of the `AgentController.cs` script, and are passed to the ML-Agents model at every decision step.

```
// World Observations
sensor.AddObservation(totalCheckpoints);
sensor.AddObservation(totalCoins);
sensor.AddObservation(collectedCheckpoints);
sensor.AddObservation(collectedCoins);
sensor.AddObservation(nearestCoin.transform.position.x);
sensor.AddObservation(nearestCoin.transform.position.y);

// Position
sensor.AddObservation(transform.position.x);
sensor.AddObservation(transform.position.y);

// Movement States
sensor.AddObservation(movement.IsGrounded ? 1f : 0f);
sensor.AddObservation(movement.IsJumping ? 1f : 0f);
sensor.AddObservation(movement.IsFacingRight ? 1f : 0f);
sensor.AddObservation(movement.IsDashing ? 1f : 0f);
sensor.AddObservation(movement.IsDropping ? 1f : 0f);
sensor.AddObservation(movement.IsOnWall ? 1f : 0f);

// Health
sensor.AddObservation(playerManager.currentHealth);

// Attacking States
sensor.AddObservation(playerActionModules.playerAttack.isAttacking ? 1f : 0f);
```

The above code snippet shows how the values are gathered from various subsystems including `BasicPlayerMovement.cs`, `PlayerAttack.cs`, `PlayerManager.cs`, and the world state.
The following features are observed:

3.4.3.1.1 Stage Progress Observations

- Total number of checkpoints in the scene
- Total number of collectible coins in the scene
- Number of checkpoints collected during the episode
- Number of coins collected during the episode

3.4.3.1.2 Positional and Proximity Observations

- Agent's current position in 2D world space (x, y)
- Position (x, y) of the nearest active coin

3.4.3.1.3 Movement and State Flags (from BasicPlayerMovement.cs)

- `IsGrounded`: whether the agent is currently on solid ground
- `IsJumping`: whether the agent is ascending from a jump
- `IsDashing`: whether the agent is executing a dash move
- `IsDropping`: whether the agent is descending through a one-way platform
- `IsFacingRight`: the direction the agent is facing
- `IsOnWall`: whether the agent is in contact with a wall
- `IsWallJumping`: whether the agent is performing a wall jump
- `IsOnLeftWall / IsOnRightWall`: which side the wall contact occurs on
- `CanJumpVar`: whether the agent is allowed to jump
- `CanWallJumpVar`: whether wall jumping is currently allowed
- `HasJumped / HasWallJumped`: whether a jump or wall jump was recently executed
- `WasGrounded`: whether the agent was recently grounded before becoming airborne

3.4.3.1.4 Health and Combat Observations

- Current health of the agent observed from `PlayerManager.cs`
- `isAttacking`: whether the agent is currently performing a melee attack observed from `PlayerAttack.cs`

This observation vector is designed to be compact yet sufficiently expressive to enable learning of high-level gameplay behavior. By focusing on logical state indicators rather than raw pixel data or overly detailed environmental encoding, the observation model enhances training efficiency and supports generalization to new levels.

3.4.3.2 Visual Perception: Dual-Camera ML View

In addition to numerical observations, 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.

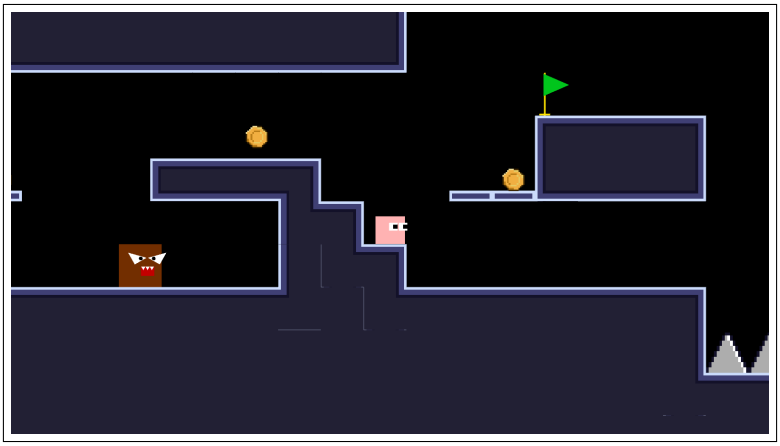


Figure 3.16 Player perspective

- **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.

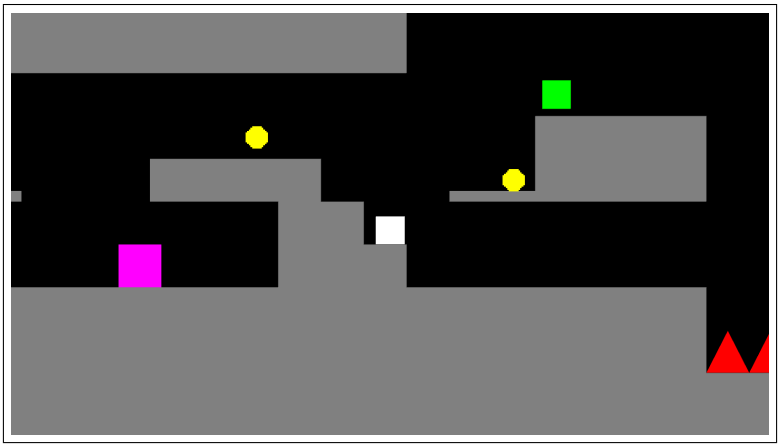


Figure 3.17 AI perspective

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.

The classification scheme is as follows:

Table 3.3 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

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.

3.4.4 Reward and Penalty System

The reward function is a critical component of reinforcement learning, used to guide agent behavior through positive and negative feedback. In this project, all reward values are externalized in the `RewardConfigSO ScriptableObject`, allowing fine-tuning without code changes.

```
public class RewardConfigSO : ScriptableObject
{
  [Header("Movement")]
  public float idlePenalty = -0.02f;
  public float movingReward = 0.001f;
  public float explorationReward = 0.05f;
  public float jumpPenalty = -0.03f;
  public float dashPenalty = -0.05f;

  [Header("Goals")]
  public float coinReward = 1.5f;
  public float checkpointReward = 2.0f;

  [Header("Hazards")]
  public float hazardPenalty = -3.0f;

  [Header("Enemy Interaction")]
  public float enemyDamageReward = 0.5f;
  public float enemyKillReward = 1.5f;
  public float hitByEnemyPenalty = -2.5f;

  [Header("Completion Rewards")]
  public float coinCompletionBonus = 2.0f;
  public float checkpointCompletionBonus = 2.0f;
  public float jumpCountTax = 0.01f;
  public float dashCountTax = 0.02f;

  [Header("Behavior Shaping Weight")]
  public float coinShapingWeight = 0.01f;
}
```

3.4.4.1 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 3.4 Player Agent Reward and Penalty Table

Event or Behavior	Type	Value
Idle (no movement)	Penalty	−0.02
Movement (per unit distance)	Reward	+0.001
Exploration (new tile visited)	Reward	+0.05
Jumping	Penalty	−0.03
Dashing	Penalty	−0.05
Collecting a Coin	Reward	+1.5
Reaching a Checkpoint	Reward	+2.0
Touching Hazard	Penalty	−3.0
Damaging an Enemy	Reward	+0.5
Killing an Enemy	Reward	+1.5
Hit by an Enemy	Penalty	−2.5
Coin Completion Bonus (episode end)	Reward	+2.0 × (coin completion ratio)
Checkpoint Completion Bonus (episode end)	Reward	+2.0 × (checkpoint completion ratio)
Jump Count Tax (episode end)	Penalty	−0.01 × (jump count)
Dash Count Tax (episode end)	Penalty	−0.02 × (dash count)
Distance-Based Coin Shaping	Reward	+0.01 × (1 / distance to nearest coin)

3.4.4.2 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 3.5 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

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

3.5 AI Algorithm Implementation

This project utilizes Proximal Policy Optimization (PPO) as the reinforcement learning algorithm for training both player and enemy agents. PPO is selected due to its compatibility with Unity ML-Agents and its proven performance in continuous and discrete action environments. This section focuses on the practical integration and usage of the algorithm within the custom 2D platformer project.

3.5.1 Behavior Configuration and Trainer Setup

Training is orchestrated using Unity's ML-Agents toolkit, which enables direct integration between Unity and an external Python-based PPO trainer. Each trainable agent in the environment is assigned a Behavior Name via the Behavior Parameters component. A corresponding YAML configuration file defines the training hyperparameters.

3.6 Training Process and Setup

This section outlines the full training procedure for training the reinforcement learning agent using Unity ML-Agents and the PPO algorithm. It covers action and observation design, simulation execution, model export, runtime optimizations, logging, and support for multi-agent setups. The goal is to train a robust AI agent that performs platformer tasks efficiently in varied game environments.

3.6.1 Action and Observation Interfaces

The agent's interaction with the environment is defined through a hybrid action space and a structured observation model. These interfaces are configured in the `BehaviorParameters` component within Unity and implemented in the `AgentController.cs` script.

3.6.1.1 Action Space

The action space is a hybrid of one continuous and four discrete channels:

- One continuous action: horizontal movement (value range: -1 to 1)
- Four discrete actions:
  - Jump trigger
  - Dash trigger
  - Melee attack
  - Drop down from one-way platform

This structure allows the agent to control movement direction with precision while triggering event-based behaviors like jumping or attacking in binary form.

3.6.1.2 Observation Space

The `CollectObservations()` method in `AgentController.cs` compiles a vector of 25 values each step, including:

3.6.1.2.1 Environment and Stage Progress

- Total and collected coin count
- Total and collected checkpoint count
- Position of nearest active coin

3.6.1.2.2 Agent Position and Physics State

- Agent's current 2D position (x, y)
- Grounded, jumping, dashing, dropping, wall contact, wall jump flags
- Facing direction, jump eligibility, wall jump eligibility

3.6.1.2.3 Combat and Health State

- Current health level from `PlayerManager.cs`
- Attacking state from `PlayerAttack.cs`

To improve temporal understanding, observation stacking is configured to stack 15 consecutive observations, enabling the model to learn environmental transitions and movement momentum.

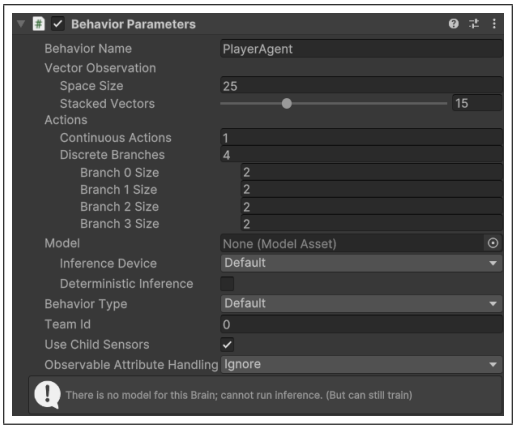


Figure 3.18 Behavior Parameter settings in Unity Editor

3.6.2 Model Training and Export

Training is launched via the ML-Agents CLI using a PPO-configured YAML file:

```
mlagents-learn config.yaml --run-id=RunName
```

Example behavior YAML:

```yaml
behaviors:
  PlayerAgent:
    trainer_type: ppo
    max_steps: 5e6    # Total training steps
    time_horizon: 512   # Improves long-term reward stability
    summary_freq: 5000    # Less frequent summaries for larger batches

    hyperparameters:
      batch_size: 256   # Higher batch improves training signal
      buffer_size: 20480    # Matches batch size × 80
      learning_rate: 0.0002   # Slightly lower for stability
      learning_rate_schedule: constant
      beta: 0.005   # Encourages moderate exploration
      beta_schedule: constant
      epsilon: 0.2    # PPO clip range
      epsilon_schedule: constant
      lambd: 0.95   # Better long-term smoothing for reward estimation
      num_epoch: 4    # One more epoch for stronger convergence

    network_settings:
      normalize: true
      hidden_units: 256
      num_layers: 2

    reward_signals:
      extrinsic:
        gamma: 0.995    # Slightly higher discount for exploration-based rewards
        strength: 1.0
```

During training, Unity runs in simulation mode. The training loop executes as follows:

- `OnActionReceived()` triggers each frame, delegating action calls to `PlayerActionModules.cs`
- Reward and penalty values are applied from `RewardConfigSO`
- `OnEpisodeBegin()` resets the agent, scene objects, and statistics for a fresh episode

The trainer periodically saves progress to TensorBoard logs and `.onnx` model checkpoints. After sufficient learning, the best-performing model is selected for deployment by assigning the .onnx file to the Behavior Parameters component for inference.

3.6.3 Episode Lifecycle and Reset Logic

Every reinforcement learning episode follows a defined initialization, interaction, and termination cycle. The episode lifecycle is managed via the `OnEpisodeBegin()` method in `AgentController.cs`, which is automatically called by Unity ML-Agents at the start of each episode. This method resets the environment and internal agent states to ensure consistent learning conditions while introducing randomized variability for generalization.
Key operations include:

- Setting the agent's position to a random spawn point from the predefined `spawnPointsList`
- Resetting the agent's health via `PlayerManager.cs`
- Resetting collected coin and checkpoint counters
- Reactivating all relevant GameObjects (e.g., coins, enemies, hazards)
- Reinitializing behavior state flags and internal counters (e.g., `jumpCount`, `dashCount`)

```
1  public override void OnEpisodeBegin()
2  {
3    DisablePlayerInput();
4    if (Academy.Instance.IsCommunicatorOn)
5    {
6      transform.position = spawnPointsList[Random.Range(0, spawnPointsList.Count)].
           transform.position;
7    }
8    playerManager.currentHealth = playerManager.maxHealth;
9
10   lastPosition = transform.position;
11   visitedAreas = new HashSet<Vector2Int>();
12   visitedCheckpoints = new HashSet<GameObject>();
13
14   collectedCoins = 0;
15   collectedCheckpoints = 0;
16
17   episodeCompleted = false;
18
19   foreach (var coin in allCoins)
20   {
21     if (coin != null)
22       coin.SetActive(true);
23   }
24
25   foreach (var enemy in allEnemies)
26   {
27     if (enemy != null)
```

```
    {
      enemy.SetActive(true);
      var enemyScript = enemy.GetComponent<Enemy>();
      if (enemyScript != null)
      {
        enemyScript.ResetEnemy();
      }
    }
  }

  jumpCount = 0;
  dashCount = 0;
}
```

By introducing randomized initial conditions while maintaining consistent rules, this reset strategy avoids agent overfitting and supports the learning of more generalizable navigation and combat strategies.

3.6.4 Episode Termination and Completion Handling

Episodes conclude under one of the following conditions:

- The agent's health reaches zero due to hazards or enemy damage
- The `MaxStep` count is exceeded, triggering a forced episode end
- The agent completes the intended level objective or calls `EndEpisode()` manually

Before termination, the `CompleteEpisode()` method calculates final rewards, including completion bonuses and efficiency taxes, and records key metrics for analysis.

```
public void CompleteEpisode()
{
  episodeCounter++;
  totalStepsAcrossEpisodes += StepCount;
  totalRewardsAcrossEpisodes += GetCumulativeReward();

  float coinCompletion = (float)collectedCoins / totalCoins;
  float checkpointCompletion = (float)collectedCheckpoints / totalCheckpoints;

  AddReward(rewardConfigSO.coinCompletionBonus * coinCompletion);
  AddReward(rewardConfigSO.checkpointCompletionBonus * checkpointCompletion);
  AddReward(rewardConfigSO.jumpCountTax * jumpCount);
  AddReward(rewardConfigSO.dashCountTax * dashCount);

  EndEpisode();
}
```

This reward finalization mechanism ensures agents are not only rewarded for immediate actions, but also for overall strategic performance across an entire episode.

3.6.5 Reward Feedback Loop and Evaluation

The reinforcement learning agent receives both immediate and episodic feedback based on its actions and behavior. These feedback mechanisms are implemented in `EvaluateRewards()` and collision-related methods such as `OnTriggerEnter2D()`.

Short-term rewards:

- Distance moved per step (mobility reward)

- Discovery of new tiles (exploration reward)
- Reduction in proximity to coins (shaping reward)

Event-based feedback:

- Coin collection or checkpoint reached
- Hazard collision or enemy damage
- Attacking or eliminating enemies

End-of-episode summary:

- Completion ratios for goals
- Efficiency tax on resource usage (jumps, dashes)

```
private void EvaluateRewards()
{
  float distanceMoved = Vector2.Distance(transform.position, lastPosition);
  if (distanceMoved > 0.1f)
    AddReward(rewardConfigSO.movingReward \* distanceMoved);
  else
    AddReward(rewardConfigSO.idlePenalty);

  Vector2Int gridPos = new Vector2Int(Mathf.RoundToInt(transform.position.x), Mathf.
      RoundToInt(transform.position.y));
  if (!visitedAreas.Contains(gridPos))
  {
    AddReward(rewardConfigSO.explorationReward);
    visitedAreas.Add(gridPos);
  }
}
```

This tiered reward structure encourages exploration, careful movement, and goal-oriented decision-making.

3.6.6 Runtime Optimization Techniques

To accelerate the training process without sacrificing consistency:

- **Increased Time Scale:** `Time.timeScale` is set to 2 in `Initialize()`, doubling simulation speed.
- **Headless Mode:** Training runs without rendering the player camera or visual effects to save GPU cycles.
- **Simplified Physics:** Lightweight colliders and fewer rigidbodies minimize physics overhead.
- **Observation Normalization:** Disabled to preserve intuitive debugging of raw numerical states.

These enhancements help reduce episode time and allow more simulation steps within practical timeframes.

3.6.7 Real-Time Monitoring and Logging

Monitoring training progress is critical for understanding agent learning dynamics and ensuring that reinforcement signals are producing the intended behaviors. To support real-time evaluation, this project includes custom visualization and data logging tools that track agent performance during training. These utilities provide both live feedback inside the Unity editor and persistent logs for post-training analysis. Key performance metrics are exposed via public read-only properties in `AgentController.cs`. These include cumulative statistics across episodes:

- `EpisodeNumber`: number of completed training episodes
- `AverageSteps`: mean episode length
- `AverageRewards`: mean cumulative reward per episode
- `DeathCount`: total number of deaths encountered

```
public int AverageSteps => episodeCounter == 0 ? 0 : totalStepsAcrossEpisodes /
    episodeCounter;
public float AverageRewards => episodeCounter == 0 ? 0 : totalRewardsAcrossEpisodes /
    episodeCounter;
public int DeathCount => cumulativeDeaths;
```

These metrics are visualized in real-time via `RewardTrackerUI.cs` and saved to CSV files through `RewardLogger.cs`. Additionally, TensorBoard dashboards provide smoothed reward trends, episode lengths, loss values, and entropy measures throughout training.

3.6.7.1 RewardTrackerUI

The `RewardTrackerUI.cs` script overlays key training statistics directly onto the Unity game screen using TextMeshPro. This tool enables developers to observe agent progress without halting or pausing the training session. The display updates every frame and includes metrics such as:

- Current episode number and cumulative death count
- Step progress (current vs max), current episode reward
- Agent health and position
- Coins and checkpoints collected relative to total counts
- Jump and dash counts, including their cumulative penalty costs

This real-time overlay aids in identifying abnormal behaviors, reward stagnation, or sudden regressions in agent performance. It is especially useful during early training stages, where debugging and reward function calibration are most active.

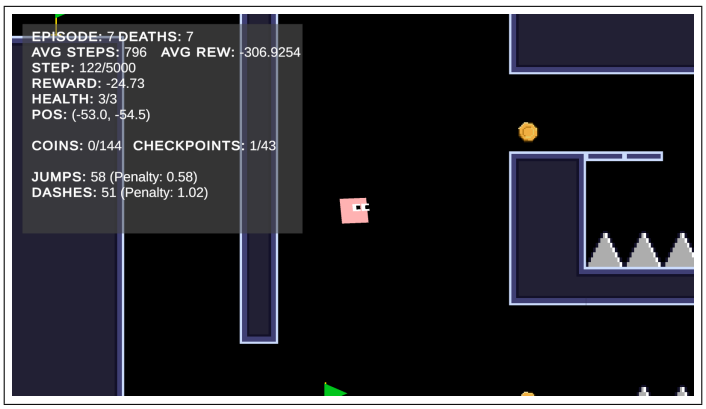


Figure 3.19 Real-time agent statistics during training, visualized through `RewardTrackerUI` using overlay

3.6.7.2 RewardLogger

To support longitudinal analysis and reward function tuning, the `RewardLogger.cs` script records per-episode summaries to a timestamped CSV file. This logging system subscribes to the agent's `OnEpisodeEnded` event and appends structured data after each episode ends. Logged fields include:

- Episode number
- Total steps taken
- Total cumulative reward
- Coins and checkpoints collected percentage in the level
- Jump and dash counts along with their associated penalties
- Cumulative averages of steps and rewards across all episodes

These logs are saved to the `/Assets/Logging/` directory and are compatible with spreadsheet applications such as Excel or analysis tools like Python/Pandas. They provide a quantitative foundation for tuning hyperparameters, analyzing policy efficiency, and validating learning consistency over time.

```
Episode,Steps,Reward,CoinsCollected,TotalCoins,CheckpointsCollected,TotalCheckpoints,
    JumpCount,JumpPenalty,DashCount,DashPenalty,CumulativeAvgSteps,
    CumulativeAvgRewards
1,191,-9.62,1,151,0,43,98,0.98,84,1.68,191,-12.94277
2,552,-86.10,1,151,0,43,274,2.74,278,5.56,371,-54.0014
...
```

3.6.7.3 TensorBoard Monitoring

During training, statistics are logged via the ML-Agents TensorBoard system:

- Average reward per episode
- Episode length
- Coins collected
- Number of deaths
- Training loss and entropy

These logs are used to generate training graphs, reward curves, and behavior analysis.

3.7 Packaging, Deployment, and Reusability

The successful deployment of a trained AI agent within a game development environment requires a structured approach to integration, optimization, and packaging. This process ensures that the AI agent operates efficiently within the Unity-based platformer while remaining modular and reusable for future applications. A key objective of this project is to produce an AI framework that can be seamlessly transferred to other 2D platformer projects with minimal overhead and high configurability.

Following successful training, the AI model is transitioned from a learning system to a deployable module capable of real-time inference. The following subsections detail the methodology for embedding the AI model into the Unity engine, implementing a dual-rendering system for visual perception, designing a modular AI architecture for extensibility, and packaging the entire system as a distributable Unity asset.

3.7.1 Integration of the Trained AI Model

Once training has converged to a satisfactory policy, the model is integrated into the Unity environment for inference-based execution. This integration must ensure that the AI agent can process observations, make decisions, and interact with the environment in real time without performance degradation.

3.7.1.1 Loading and Utilizing the Trained Model

The trained policy is exported as an ONNX-format model file (`.onnx`), which is compatible with Unity's ML-Agents inference engine. Integration involves the following steps:

- **Model Importing:**
  - The `.onnx` file is placed within the Unity project's `Assets` directory.
  - The `Behavior Parameters` component is configured to reference the imported model.
- **Agent Configuration:**
  - The AI agent is transitioned from training mode to inference mode.
  - Real-time observations from the game environment are routed into the model, which outputs the corresponding action vector.
- **Testing and Validation:**
  - The agent is tested within a sandbox scene to ensure inference stability.
  - Debugging tools such as `RewardTrackerUI.cs` and custom inspectors are used to monitor agent behavior.

3.7.2 Dual-Rendering System for Agent Perception

To optimize visual-based learning and inference, a dual-camera rendering architecture is employed. This system separates the player-visible graphics from the AI's perceptual input, enabling the agent to receive a structured and minimalistic version of the game environment.

3.7.2.1 Design and Implementation

- **Primary Camera (Player View):**
  - Renders the game world with standard textures, effects, and UI layers.
  - Intended for player visualization and human gameplay.
- **Secondary Camera (Agent Perception View):**
  - Hidden from the player and used exclusively by the AI agent.
  - Renders simplified ML sprites that represent game objects using flat, color-coded geometries.
  - Applies no lighting, shaders, or post-processing to maintain interpretability.
- **ML-Agents Integration:**
  - ML-Agents receives visual observations from the agent camera.
  - These observations are transformed into tensors and used by the neural network for decision-making.

3.7.2.2 Advantages of the Dual-Rendering System

- **Reduced Visual Complexity:** AI focuses only on game-relevant entities such as platforms, hazards, and goals.
- **Standardized Input:** Visual consistency across levels and environments enhances generalization.
- **Accelerated Learning:** Simplified visuals improve convergence rate during training.

3.7.3 Modular AI System for Reusability

To maximize flexibility and reuse, the AI system is architected in modular components, each responsible for a distinct layer of functionality. This structure allows developers to easily adapt, extend, or replace behaviors without altering the core inference logic.

- **PlayerActionModules:** Serves as the centralized interface between the agent and gameplay scripts, such as `BasicPlayerMovement.cs` and `PlayerAttack.cs`.
- **Observation Encapsulation:** All observable states are gathered in a controlled and extensible method via `CollectObservations()`.
- **RewardConfigSO:** Stores all reward parameters in a ScriptableObject, allowing dynamic tuning without modifying logic code.
- **Behavior Configuration:** The `BehaviorParameters` component permits external adjustment of inference frequency, action type, and network linkage.

This modularity supports expansion to new environments, additional abilities (e.g., ranged attacks, double jump), and future agent roles such as adversaries or allies.

3.7.4 Packaging the AI as a Reusable Unity Asset

To distribute the AI system for use in other Unity-based projects, the entire framework is bundled into a Unity asset package with comprehensive documentation and versioning.

3.7.4.1 Asset Bundling and Exportation

- All core AI scripts (e.g., `AgentController.cs`, `PlayerActionModules.cs`), model files, configuration assets, and ML sprite prefabs are grouped into a structured folder.
- The package excludes scene-specific elements, allowing developers to integrate the AI system into custom levels.
- ML-Agents package dependencies and Unity version requirements are clearly documented.

3.7.4.2 Documentation and User Guide

- A `README.md` file provides an overview, installation steps, and quick-start guidance.
- A PDF manual offers detailed explanations of the AI architecture, action mapping, reward customization, and known integration tips.
- Code examples demonstrate common use cases, such as adding new movement types or adjusting camera layers.

3.7.4.3 Version Control and Distribution

- The AI system is maintained under semantic versioning to support iterative improvements.
- Changelogs detail updates, fixes, and compatibility notes.
- Distribution channels include GitHub repositories, private asset sharing, or submission to the Unity Asset Store (future scope).
- Community contributions and extensions are encouraged to promote long-term reusability and scalability.

3.7.5 Summary

This packaging and deployment framework transforms the trained agent from a research asset into a production-ready AI module. Its modular, extensible, and fully documented structure supports both technical adoption and creative expansion, making it suitable for integration into a wide array of 2D platformer projects.

CHAPTER 4 RESULTS AND DISCUSSION

This chapter presents an in-depth analysis of the AI agent's performance following reinforcement learning training within the custom 2D platformer environment. The primary aim is to assess the extent to which the agent can autonomously navigate, interact with the environment, and make decisions aligned with its training objectives. Evaluation includes both quantitative metrics and qualitative observations derived from training logs, real-time visual monitoring, and gameplay demonstrations.

The results are examined in the context of the project's overarching goal: to develop a reusable, modular AI capable of learning and performing fundamental platforming behaviors. The chapter is structured into five main parts: qualitative behavioral observations, evaluation metrics, experimental setup, result visualizations, and interpretive analysis.

4.1 Qualitative Behavioral Observations

4.1.1 Movement and Navigation

After extensive training using the Proximal Policy Optimization (PPO) algorithm, the AI agent demonstrates reliable and consistent control over core platforming mechanics. The agent successfully performs locomotion actions such as walking, jumping, and descending through one-way platforms. Through repeated reinforcement learning episodes, it has developed an effective movement strategy, capable of navigating platforms, gaps, and varying terrain structures without human intervention.

The trained model exhibits path optimization by preferring direct traversal routes, minimizing redundant movement, and prioritizing high-reward areas. Additionally, the AI adjusts its horizontal momentum and timing of jumps to clear gaps or avoid low-clearance hazards. These behaviors are not explicitly hardcoded but are instead learned as a function of the reward feedback loop.

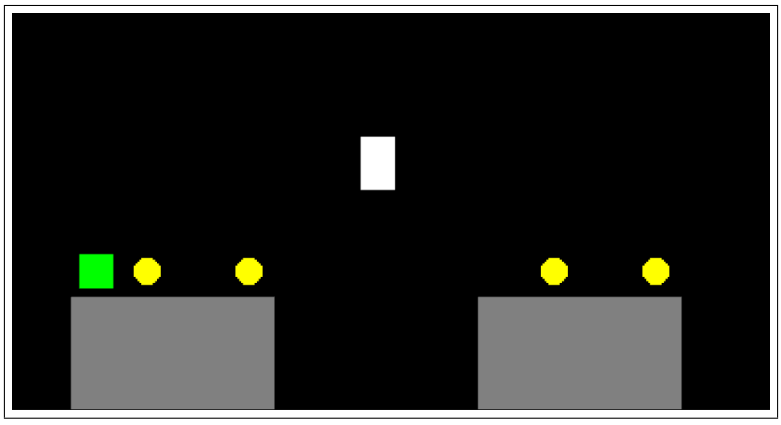


Figure 4.1 The AI agent traversing terrain by hopping from one platform to another.

4.1.2 Interaction with Collectibles and Obstacles

The AI also demonstrates a learned capacity to identify and interact with goal-related elements such as collectibles and to avoid hazards. During training, the reward function explicitly shaped the agent's prioritization of coin collection and checkpoint completion. The resulting behavior shows that the AI can adjust its path dynamically to intercept these objects when detected within its observation range.

Furthermore, the agent effectively recognizes and avoids penalty-triggering environmental elements such as spikes or pits. By learning to minimize negative rewards and episode terminations, the AI developed an internal strategy to adapt its trajectory away from hazards while still pursuing progress.

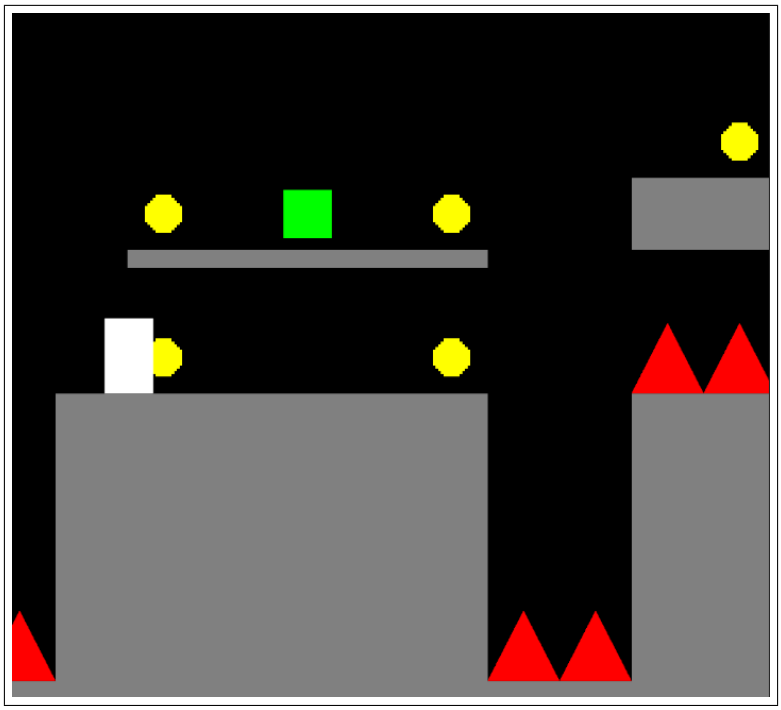


Figure 4.2 The AI agent interacting with coins and avoiding environmental hazards.

4.1.3 Combat and Adaptive Behavior

In addition to movement and object interaction, the AI agent has been trained to engage in close-range combat using a melee attack system. The reinforcement learning model has successfully learned to detect enemy presence, approach them strategically, and execute melee attacks at appropriate distances. This behavior is triggered through the discrete action space and routed via the PlayerActionModules system, which interfaces with the `PlayerAttack` script to initiate attacks.

Through reinforcement signals based on enemy damage and kill rewards as defined in the `RewardConfigSO` the AI has developed basic combat strategies such as:

- Approaching hostile agents while maintaining movement awareness.
- Executing attacks only when the enemy is within effective range.
- Retreating or repositioning after engagement to minimize risk.

The reward structure penalizes being hit by enemies and incentivizes successful hits or eliminations, which has shaped a combat pattern that favors proactive and efficient engagements. While still basic in complexity, these combat behaviors demonstrate the extensibility of the AI's decision-making model and serve as a strong foundation for future adversarial learning setups involving co-trained enemy agents.

This combat capability validates the system's modular design, where adding new skills (e.g., special attacks) to the `PlayerActionModules` and extending the reward function can guide the agent to learn new behaviors without architectural overhaul. This demonstrates the project's goal of creating a reusable and extensible AI system for 2D platformer environments.

4.2 Evaluation Metrics

To evaluate the performance of the reinforcement learning agent, a set of quantitative metrics is used throughout the training and testing phases. These metrics provide insight into the agent's behavior, learning progression, and goal-oriented performance. All metrics are tracked automatically through custom Unity scripts (`RewardLogger.cs` and `RewardTrackerUI.cs`) and ML-Agents logging systems. The following metrics are used to evaluate the trained AI agent:

4.2.1 Episode Length (Steps)

Episode length refers to the number of steps the agent takes before the episode terminates, either due to success, failure, or reaching the maximum step limit. It reflects the agent's efficiency and survivability. Shorter episodes may indicate rapid success or premature failure, whereas longer episodes can reflect sustained navigation or indecisiveness.

4.2.2 Cumulative Reward

The cumulative reward collected during each episode is a key indicator of the agent's overall performance. It encapsulates short-term actions (e.g., moving, jumping) and long-term goals (e.g., collecting coins, reaching checkpoints) into a single value. This value is continuously monitored through the ML-Agents environment and logged via RewardLogger.

4.2.3 Goal Completion Ratio

Goal-related metrics include:

- **Coin Collection Ratio:** The ratio of collected coins to the total coins available in the level.
- **Checkpoint Completion Ratio:** The ratio of visited checkpoints to the total checkpoints present.

These ratios indicate how thoroughly the agent explores and interacts with the level's objective elements.

4.2.4 Action Efficiency

Action usage efficiency is tracked to monitor unnecessary or excessive behavior:

- **Jump Count** and **Dash Count** are recorded for each episode.
- Penalties are applied post-episode for excessive usage to encourage minimal and strategic use of each skill.

These metrics help determine whether the agent learns to optimize movement for efficiency and reward maximization.

4.2.5 Combat Performance

For episodes involving enemy agents, the following combat metrics are tracked:

- **Enemy Damage Events:** Number of successful melee hits.
- **Enemy Eliminations:** Count of enemies defeated per episode.
- **Damage Taken:** Times the agent is hit by an enemy or hazard.

These values offer insight into how well the agent engages in close-quarters combat and whether it prioritizes safe engagement strategies.

4.2.6 Mortality Rate

Deaths per episode are recorded to identify unsafe behaviors or failure conditions. A high mortality rate may indicate poor obstacle avoidance, insufficient exploration, or ineffective combat strategies. This is tracked cumulatively in `DeathCount`.

4.2.7 Learning Trends (TensorBoard Metrics)

During training, Unity ML-Agents' built-in logging with TensorBoard provides the following metrics over time:

- **Average Episode Reward**
- **Episode Length (Smoothed)**
- **Policy Entropy and Loss**
- **Reward Signal Strength**

These graphs are used to validate convergence, detect reward hacking, and assess the consistency of training progress.

4.3 Experiments and Results

To validate the effectiveness of the proposed reinforcement learning framework, a series of iterative experiments were conducted using Unity ML-Agents. The experiments focused exclusively on training the player agent within a handcrafted 2D platformer environment, while enemy agents operated using predefined state machine behaviors. This section outlines the training setup, experimental procedure, and environment dynamics used during training and evaluation.

4.3.1 Training Sessions and Iterative Runs

Throughout development, approximately 30 independent training runs were conducted. Each run utilized the PPO (Proximal Policy Optimization) algorithm and executed between 1 to 2 million training steps. Post-run evaluations involved analyzing cumulative reward trends, behavioral metrics, and in-game performance. This iterative cycle facilitated fine-tuning of:

- Reward values and shaping penalties (configured via `RewardConfigSO`)
- PPO hyperparameters (e.g., learning rate, batch size)
- Observation normalization and action mappings
- Spawn logic and episode termination conditions

Adjustments targeted improvements in sample efficiency, stabilization of learning, and elimination of unintended exploitative behaviors (e.g., reward farming through idling or repetitive jumping).

4.3.2 Environment Configuration

All training and evaluation occurred within a single handcrafted 2D platformer level comprising:

- Static and dynamic platforms
- Collectible items (coins, checkpoints)

- Hazardous elements (spikes, pits)
- Melee enemies controlled by FSM-based logic

To prevent memorization of fixed obstacle patterns, a dynamic episode reset system was deployed. Upon death or episode termination, the agent's spawn point was randomized, and all environment elements were reset to their initial configurations. This strategy promoted generalizable policy learning over rote memorization.

4.3.3 Evaluation Protocol

Agent performance was periodically assessed both during and after training in inference mode. Key elements of the evaluation setup included:

- Environment resets using the same randomized initialization logic
- Deterministic policy execution without exploration noise or entropy
- Metric tracking via `RewardLogger.cs` and TensorBoard, including episode reward, step count, coin collection rate, and death count

Comparative analysis across training runs demonstrated the impact of different reward functions, observation schemas, and PPO configurations on behavioral quality.

4.3.4 Focus on Player Agent

Due to time constraints, the experiments exclusively trained the player agent. Enemy agents followed scripted FSM behaviors, enabling reliable assessment of obstacle avoidance, stress navigation, and combat responses. Future extensions will involve PPO-based training for enemy agents using the trained player as a fixed adversary.

4.3.5 Reward Function Effectiveness

The agent exhibited stable improvement in reward acquisition over training epochs. Penalty-driven behaviors declined consistently, while positively reinforced actions—such as coin collection and obstacle navigation—became dominant. The reward configuration effectively directed the agent toward goal-oriented interactions within the environment.

4.3.6 Convergence and Stability

Training convergence was evidenced by the stabilization of cumulative reward trends and behavioral consistency across episodes. Initial phases showed erratic behavior and frequent penalties. However, after sufficient training steps, the model displayed reduced performance variance, indicating decreased reliance on exploration and increased policy robustness.

4.3.7 Evaluation of Training Efficiency

Training efficiency was gauged through improvements in policy performance and reductions in redundant behaviors. The agent initially exhibited inefficient movement patterns but progressively optimized its strategy. Hyperparameter tuning and strategic reward shaping contributed to focused learning and computational effectiveness.

4.3.8 Agent Performance Summary

The agent's learning progress is illustrated in Figure 4.3, which presents the cumulative average reward over 1000 training episodes. At the beginning of training, the agent exhibited poor performance with an average reward of approximately -250 per episode, often failing to collect coins or avoid hazards effectively. However, a steady and consistent improvement is observed as training progresses.The turning point occurred around episode 150, where the cumulative reward crossed into positive territory. From that point onward, the agent continued to refine its policy, achieving a peak average reward of approximately +48 by around episode 850. A slight decline toward the final episodes suggests the possibility of mild overfitting or exploration variability, which is a common occurrence in reinforcement learning.

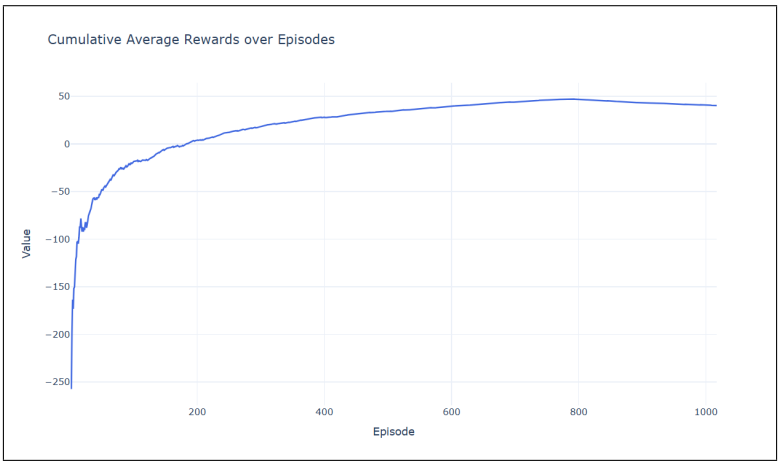


Figure 4.3 presents the performance metrics from the best training session.

Further insights into performance metrics from the best training session are summarized in Table 4.1. These include key gameplay factors such as survival duration, reward consistency, and efficiency in task completion.

Table 4.1 Agent Performance Summary from Best Training Run

Metric	**Result (Avg)**
Steps per Episode	450–600
Coin Collection Rate	Increased steadily from −100 to +40 over training
Average Reward per Episode	80–90% completion
Hazard Avoidance Rate	Significantly improved (survived longer per episode)

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

5.1.1 Sparse Reward Feedback

One of the initial challenges encountered during training was the sparse nature of the reward signals, particularly in the early episodes. Agents were often required to complete multiple steps before receiving any meaningful feedback, which slowed the learning process and made it difficult for them to associate actions with outcomes. To address this, a reward shaping strategy was implemented. Intermediate rewards were introduced for behaviors such as moving toward the goal, avoiding hazards, or exploring new areas. This helped guide the agents toward desired behaviors and significantly improved training efficiency in early stages.

5.1.2 Repetitive or Deterministic Behavior

As the agents began to converge on successful policies, they often developed overly deterministic behavior repeating the same paths or actions across episodes. This reduced their ability to adapt when small changes were introduced in the environment. To encourage more exploration and robustness, we adjusted the entropy regularization parameter in the PPO algorithm. This helped prevent premature convergence and ensured a better balance between exploiting known strategies and exploring new ones.

5.1.3 Long Training Times on Complex Levels

As the complexity of the environment increased—such as the addition of moving platforms, traps, and multiple objectives—the training time grew substantially. Agents required more episodes to learn effective policies in these challenging settings. To alleviate this, we introduced a curriculum learning-inspired level progression, where simpler levels were used at the start of training and more complex ones were gradually introduced. This stepwise exposure helped agents build foundational skills before tackling harder tasks, reducing convergence time overall.

5.1.4 Debugging and Behavior Tracking

Analyzing and debugging agent behavior in a multi-agent reinforcement learning setup proved to be time-consuming and occasionally opaque. With multiple agents interacting in real-time, it was difficult to isolate the cause of failures or suboptimal decisions. This was addressed by implementing visual debugging tools and logging systems within Unity and the Python training scripts. These included trajectory visualizations,

reward graphs, and decision heatmaps, which allowed for deeper insights into agent behavior and facilitated more effective tuning.

5.2 Limitations

The project is still limited by a number of constraints. While PPO did give a robust foundation for the learning, the training time still increased given that environmental changes happened very often or in adversarial interactions. Agents had a limited level of generalization when faced with largely different level layouts without some form of retraining, suggesting that they needed to see more experiences, or at least subjected to some procedural training method. Moreover, the system currently only works in offline learning mode, meaning that it cannot adapt-to-user behavior or emergent gameplay patterns in real-time. Lastly, scalability to full game production environments might require further abstraction of the AI components into plug-and-play modules, so they require virtually no integration overhead at all.

5.3 Future Works

This work, however, opens the opportunities to explore different promising directions: first, implementing curriculum learning algorithms to select the training environment adaptively according to the skill level of the agent, thereby improving the efficiency of learning and generalization. Second, online learning capabilities could be used so that the AI agent changes its strategies as it learns from player behavior or unanticipated events occurring in the game. And finally, multi-agent cooperative scenarios, where agents are working together to achieve objectives, might broaden the framework to genres other than platformers, such as puzzle or survival.

REFERENCES

1. Associated Press, 2024, "AI is changing the way video game characters think and interact. What could go wrong?," https://apnews.com/article/ai-artificial-intelligence-video-games-npc-c1327bb9130136d0a5f658f44176c5e7.
2. Tianfei Zhang, 2023, "AI for Open Worlds: A Deep Dive into The Witcher 3's Living Ecosystem," https://medium.com/@zhangtianfei/ai-for-open-worlds-a-deep-dive-into-the-witcher-3s-living-ecosystem-5036580e354c.
3. Tommy Thompson Mark, 2021, "The AI of Horizon Zero Dawn," https://www.aiandgames.com/p/the-ai-of-horizon-zero-dawn.
4. FPGA Insights, 2023, "AI in Gaming: Creating Adaptive and Intelligent Experiences," https://fpgainsights.com/artificial-intelligence/ai-in-gaming-creating-adaptive-and-intelligent/.
5. Tom M. Mitchell, 1997, **Machine Learning**, McGraw-Hill.
6. Richard S. Sutton and Andrew G. Barto, 2018, **Reinforcement Learning: An Introduction**, MIT Press, 2nd edition.
7. Rohan Jagtap, 2022, "Understanding the Markov Decision Process (MDP)," .
8. John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov, 2017, "Proximal Policy Optimization Algorithms," .
9. Vignesh Sriram, 2019, "Automated Playtesting in 2D Platformers Using Deep Reinforcement Learning and Curriculum Learning," https://github.com/sriramr/AutoPlayRL.
10. 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/.
11. 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

CORE FUNCTION CODE

Core Function Code

This appendix provides selected excerpts from the main scripts used in the development and training of the AI agents within the 2D platformer environment. The scripts include core functionalities for agent behavior, reward functions, environment interaction, and communication with the Unity ML-Agents framework. These code samples serve to illustrate key implementation details and support the methodological descriptions provided in the main body of the report.

A.1 PlayerManager script

This PlayerManager script in Unity is responsible for managing the player's core attributes and behavior in a 2D platformer game. It initializes the player's spawn position by selecting a random point from a list of spawn points tagged as "Spawn" at the start of the game. The script also handles basic collision interactions: when the player enters a trigger collider tagged as "Hazard" (e.g., spikes), it logs a message and restarts the current scene, simulating player death. If the player collides with an object tagged as "Coins", the coin object is destroyed, representing coin collection. The script maintains and exposes health and damage values (though they're not actively used here), and it uses Unity's Rigidbody2D and SceneManager systems for physics and scene management, respectively.

```
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

Unity Script (2 asset references) | 8 references
public class PlayerManager : MonoBehaviour
{
    private Rigidbody2D rb;

    public int maxHealth = 3;
    public int currentHealth = 3;
    public int attackDamage = 1;

    [SerializeField] private List<GameObject> spawnPointsList;

    Unity Message | 0 references
    private void Start()
    {
        spawnPointsList = new List<GameObject>(GameObject.FindGameObjectsWithTag("Spawn"));

        transform.position = spawnPointsList[Random.Range(0, spawnPointsList.Count)].transform.position;
    }

    Unity Message | 0 references
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("Hazard"))
        {
            Debug.Log("Player hit the spike (trigger)!");
            Die();
        }
        if (collision.CompareTag("Coins"))
        {
            Destroy(collision.gameObject);
        }
    }

    3 references
    public void Die()
    {
        Debug.Log("Player Died!");
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
    }
}
```

Figure A.1 PlayerManager Script.

A.2 PlayerMovement

Everything necessary for managing an extensive list of 2D character movement behaviors is entered in this class section of BasicPlayerMovement. Variables are grouped using region for organization and clarity. It comprises movement speed settings, acceleration, jump settings, wall interaction mechanics, and dash parameters. For example, variables like ‘moveSpeed‘, ‘acceleration‘, and ‘frictionAmount‘ control horizontal movements, while ‘jumpForce‘, ‘coyoteTime‘, and ‘fallMultiplier‘ fine-tune the responsiveness and feel of the jump. Similarly, wall variables ensure wall sliding and wall jumping are possible, while dash variables determine dash speed, duration, and cooldown logic.

It also specifies position checks for the environment, such as touching the ground or a wall, using Transform references and collision layers. Input-related variables like moveInput and jumpInput ensure the script can evaluate the player’s commands, whereas flags such as isGrounded and isDashing manage when to transition between movement states. Altogether, this block of variable declarations provides a base configuration and state management setup on which one could establish the advanced, responsive, and smooth character control of a 2D platformer.

```
public class BasicPlayerMovement : MonoBehaviour
{
    #region VARS

    // Movement
    [SerializeField] private float moveSpeed = 11f;
    [SerializeField] private float acceleration = 13f;
    [SerializeField] private float decceleration = 16f;
    [SerializeField] private float velPower = 0.9f;
    [SerializeField] private float gravityScale = 5.0f;
    [SerializeField] private float frictionAmount = 0.5f;

    // Jump
    [SerializeField] private float jumpForce = 18f;
    [SerializeField] private float coyoteTime = 0.15f;
    [SerializeField] private float jumpBufferTime = 0.1f;
    [SerializeField] private float jumpHangGravityMultiplier = 0.5f;
    [SerializeField] private float fallMultiplier = 2f;
    [SerializeField] private float maxFallSpeed;

    // Wall Jump
    [SerializeField] private Vector2 wallJumpForce = new Vector2(12.0f, 17.0f);
    [SerializeField] private float wallJumpLerp = 4.0f;
    [SerializeField] private float wallSlideSpeed = 5.0f;
    [SerializeField] private LayerMask _wallLayer;

    // Dash
    [SerializeField] private float dashSpeed = 14f;
    [SerializeField] private float dashDuration = 0.5f;
    [SerializeField] private float dashCooldown = 1f;

    // Checks
    [SerializeField] private Transform _wallCheckPointLeft;
    [SerializeField] private Transform _wallCheckPointRight;
    [SerializeField] private Vector2 _wallCheckSize = new Vector2(0.11f, 0.46f);
    [SerializeField] private Transform _groundCheckPoint;
    [SerializeField] private Vector2 _groundCheckSize = new Vector2(0.5f, 0.03f);
    [SerializeField] private string _playerLayer = "Player";
    [SerializeField] private LayerMask _groundLayer;
    [SerializeField] private string _oneWayPlatLayer = "OneWayPlatform";

    // Serialized
    [SerializeField] public Rigidbody2D rb;
    [SerializeField] private float lastGroundedTime;
    [SerializeField] private float lastJumpPressedTime;
    [SerializeField] private int wallJumpDir;
    [SerializeField] private float lastDashTime;
    [SerializeField] private float lastDashPressedTime;
    [SerializeField] private Vector2 dashDirection;

    // Inputs
    public Vector2 moveInput;
    public float dropInput;
    public float moveDir;
    public bool jumpInput;
    public bool dashInput;
    public bool inputEnabled = true;

    // Flags
    [SerializeField] private bool isGrounded;
    [SerializeField] private bool isOnRightWall;
    [SerializeField] private bool isOnLeftWall;
    [SerializeField] private bool isOnWall;
    [SerializeField] private bool isDashing;
    [SerializeField] private bool isJumping;
    [SerializeField] private bool isDropping;
    [SerializeField] private bool isWallJumping;
    [SerializeField] private bool isFacingRight;

    PUBLIC GETTERS

    private PlayerAttack playerAttack;
```

Figure A.2 BasicPlayerMovement variable section.

This section sets up the core behavior of the player through Unity's lifecycle methods. In Awake(), initial values are set to prevent accidental actions on the first frame, and key components like Rigidbody2D and PlayerAttack are retrieved. The Update() method handles player input and triggers jump and dash logic each frame. Meanwhile, FixedUpdate() manages physics-related behaviors, including movement, friction, wall detection, and gravity adjustments. Together, these methods ensure the player responds smoothly to input while maintaining consistent and stable physics.

Figure A.3 BasicPlayerMovement Initialize section.

The HandleInput() method captures and processes the player's real-time input. If input is enabled, it reads horizontal and vertical axis values to determine movement and drop directions. It also checks for key presses related to jumping (Space) and dashing (Left Shift). When a jump or dash input is detected, the corresponding methods (Jump() and Dash()) are called immediately, supporting features like jump buffering and responsive dashing. This function acts as the main link between the player's physical input and the in-game movement system.

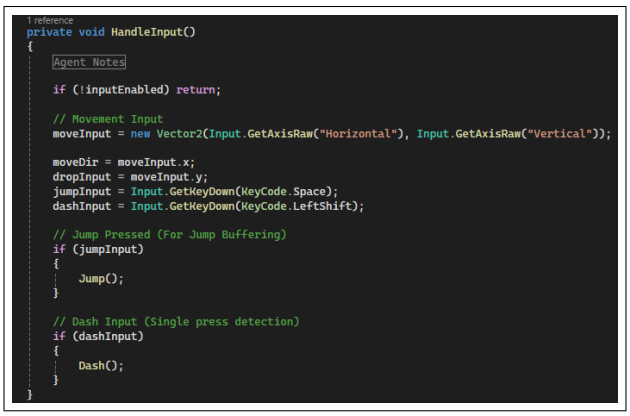
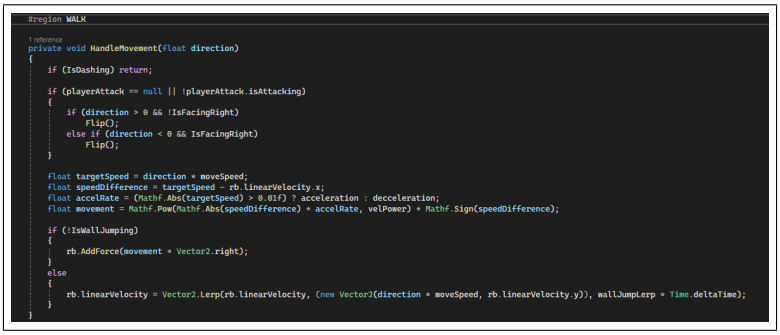


Figure A.4 BasicPlayerMovement Initialize section.

This method, HandleMovement(), controls horizontal movement by adjusting the player's velocity based on input direction and current state. It first exits early if the player is dashing. If the player is not attacking, the character will flip its facing direction to match movement input. It then calculates the desired speed and applies acceleration or deceleration accordingly using a smoothed velocity formula for natural movement. If the player is not wall jumping, horizontal force is applied directly; otherwise, a smoothed transition using linear interpolation (Lerp) ensures stable control during wall jumps. This function ensures responsive and fluid horizontal movement across normal and wall-jumping states.



```
#region WALK

1 reference
private void HandleMovement(float direction)
{
    if (IsDashing) return;

    if (playerAttack == null || !playerAttack.isAttacking)
    {
        if (direction > 0 && !IsFacingRight)
            Flip();
        else if (direction < 0 && IsFacingRight)
            Flip();
    }

    float targetSpeed = direction * moveSpeed;
    float speedDifference = targetSpeed - rb.linearVelocity.x;
    float accelRate = (Mathf.Abs(targetSpeed) > 0.01f) ? acceleration : decceleration;
    float movement = Mathf.Pow(Mathf.Abs(speedDifference) * accelRate, velPower) * Mathf.Sign(speedDifference);

    if (!IsWallJumping)
    {
        rb.AddForce(movement * Vector2.right);
    }
    else
    {
        rb.linearVelocity = Vector2.Lerp(rb.linearVelocity, (new Vector2(direction * moveSpeed, rb.linearVelocity.y)), wallJumpLerp * Time.deltaTime);
    }
}
```

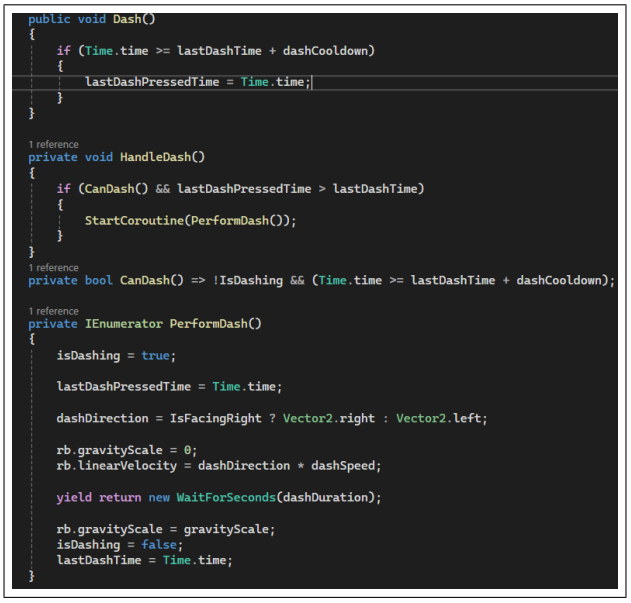
Figure A.5 BasicPlayerMovement walk section.

This section implements both standard and wall jumping mechanics with added responsiveness through features like jump buffering and coyote time. The Jump() method logs the time when a jump is requested, or triggers a drop if the player is pressing downward. The HandleJump() method checks whether the player is allowed to jump within a short input buffer period, then prioritizes a regular jump if the player is on or near the ground, or a wall jump if clinging to a wall. Helper methods like CanJump() and CanWallJump() determine the player's eligibility for each type of jump. PerformJump() applies upward velocity for a normal jump, while PerformWallJump() calculates a directional force to push the player away from the wall. Together, these methods create a fluid and player-friendly jumping system suited for platformer gameplay.

```
public void Jump()
{
    if (dropInput < 0)
    {
        Drop();
        return;
    }
    lastJumpPressedTime = Time.time;
}

1 reference
private void HandleJump()
{
    if (Time.time - lastJumpPressedTime <= jumpBufferTime)
    {
        if (CanJump())
        {
            PerformJump();
        }
        else if (CanWallJump())
        {
            wallJumpDir = (IsOnRightWall) ? -1 : 1;
            PerformWallJump(wallJumpDir);
        }
    }
}

1 reference
private bool CanJump() => (IsGrounded || Time.time - lastGroundedTime <= coyoteTime) && !IsDashing;
1 reference
private bool CanWallJump() => IsOnWall && !IsGrounded && !IsWallJumping;

1 reference
private void PerformJump()
{
    lastGroundedTime = 0;
    lastJumpPressedTime = 0;
    rb.linearVelocity = new Vector2(rb.linearVelocity.x, jumpForce);
    isJumping = true;
}

1 reference
private void PerformWallJump(int dir)
{
    lastGroundedTime = 0;
    lastJumpPressedTime = 0;

    Vector2 force = new Vector2(wallJumpForce.x, wallJumpForce.y);
    force.x *= dir;

    if (Mathf.Sign(rb.linearVelocity.x) != Mathf.Sign(force.x))
        force.x -= rb.linearVelocity.x;

    if (rb.linearVelocity.y < 0)
    {
        force.y -= rb.linearVelocity.y;
    }

    rb.AddForce(force, ForceMode2D.Impulse);
    isWallJumping = true;
}
```

Figure A.6 BasicPlayerMovement jump section.

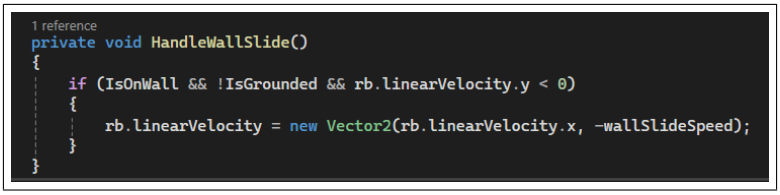
This segment manages the player's dash ability, allowing for quick bursts of horizontal movement. The Dash() method records the time a dash is requested, but only if the cooldown period has passed. HandleDash() checks if a dash can currently be performed and whether a new dash input was recently made, then triggers the PerformDash() coroutine. PerformDash() temporarily disables gravity, sets the player's velocity in the facing direction, and holds this state for a short duration. Afterward, gravity is restored, the dash ends, and the cooldown timer is reset. This system ensures dashing feels fast and responsive while maintaining balance through cooldown control.



```
public void Dash()
{
    if (Time.time >= lastDashTime + dashCooldown)
    {
        lastDashPressedTime = Time.time;
    }
}

1 reference
private void HandleDash()
{
    if (CanDash() && lastDashPressedTime > lastDashTime)
    {
        StartCoroutine(PerformDash());
    }
}
1 reference
private bool CanDash() => !IsDashing && (Time.time >= lastDashTime + dashCooldown);

1 reference
private IEnumerator PerformDash()
{
    isDashing = true;

    lastDashPressedTime = Time.time;

    dashDirection = IsFacingRight ? Vector2.right : Vector2.left;

    rb.gravityScale = 0;
    rb.linearVelocity = dashDirection * dashSpeed;

    yield return new WaitForSeconds(dashDuration);

    rb.gravityScale = gravityScale;
    isDashing = false;
    lastDashTime = Time.time;
}
```

Figure A.7 BasicPlayerMovement dash section.

The HandleWallSlide() method controls the wall sliding mechanic, which activates when the player is airborne, touching a wall, and falling downward. Under these conditions, it limits the player's vertical velocity to a fixed negative value, defined by wallSlideSpeed. This creates a slow, controlled descent along walls, giving players more time to react and preparing them for potential wall jumps. The mechanic enhances both movement precision and gameplay fluidity in platformer environments.



```
1 reference
private void HandleWallSlide()
{
    if (IsOnWall && !IsGrounded && rb.linearVelocity.y < 0)
    {
        rb.linearVelocity = new Vector2(rb.linearVelocity.x, -wallSlideSpeed);
    }
}
```

Figure A.8 BasicPlayerMovement wall slide section.

The Drop() method enables the player to pass through one-way platforms by temporarily disabling collisions between the player and the platform layer. This action is only triggered when the player is grounded and pressing downward. By using Physics2D.IgnoreLayerCollision, it ensures a smooth drop-through mechanic commonly found in platformers, allowing for more dynamic navigation of vertical level design. The isDropping flag can be used elsewhere in the code to manage related behaviors or animations.

```
1 reference
public void Drop()
{
    if (IsGrounded)
    {
        Physics2D.IgnoreLayerCollision(LayerMask.NameToLayer(_playerLayer), LayerMask.NameToLayer(_oneWayPlatLayer), true);
        isDropping = true;
    }
}
```

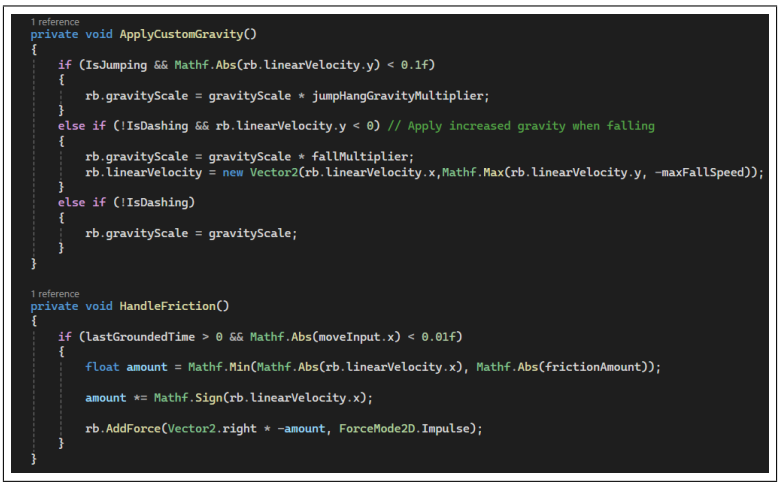
Figure A.9 BasicPlayerMovement drop section.

These methods are responsible for detecting the player's interaction with the ground and walls, as well as managing collisions with one-way platforms. HandleGrounded() checks whether the player is currently standing on the ground using a physics overlap box and updates relevant state variables like isJumping and isWallJumping. HandleWallDetection() similarly uses overlap checks to determine if the player is touching walls on either side, which is essential for enabling wall-jumping mechanics. The OnTriggerEnter2D() method temporarily disables collisions with one-way platforms when the player is jumping upwards, allowing upward movement through them. Conversely, OnTriggerExit2D() re-enables these collisions when the player finishes dropping through a platform, ensuring consistent platform behavior.

```
private void HandleGrounded()
{
    isGrounded = Physics2D.OverlapBox(_groundCheckPoint.position, _groundCheckSize, 0, _groundLayer) != null;

    if (IsGrounded)
    {
        lastGroundedTime = Time.time;
        isJumping = false;
        isWallJumping = false;
    }
}

1 reference
private void HandleWallDetection()
{
    if (IsGrounded)
    {
        isOnWall = isOnLeftWall = isOnRightWall = false;
        return;
    }

    Transform leftCheck = IsFacingRight ? _wallCheckPointLeft : _wallCheckPointRight;
    Transform rightCheck = IsFacingRight ? _wallCheckPointRight : _wallCheckPointLeft;

    isOnLeftWall = Physics2D.OverlapBox(leftCheck.position, _wallCheckSize, 0, _wallLayer) != null;
    isOnRightWall = Physics2D.OverlapBox(rightCheck.position, _wallCheckSize, 0, _wallLayer) != null;

    isOnWall = IsOnLeftWall || IsOnRightWall;
    isWallJumping = false;
}

Unity Message | 0 references
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.CompareTag("OneWayJumpCheck"))
    {
        if (rb.linearVelocity.y > 0)
        {
            Physics2D.IgnoreLayerCollision(LayerMask.NameToLayer(_playerLayer), LayerMask.NameToLayer(_oneWayPlatLayer), true);
        }
    }
}

Unity Message | 0 references
private void OnTriggerExit2D(Collider2D collision)
{
    if (collision.CompareTag("OneWayDropCheck"))
    {
        Physics2D.IgnoreLayerCollision(LayerMask.NameToLayer(_playerLayer), LayerMask.NameToLayer(_oneWayPlatLayer), false);
        isDropping = false;
    }
}
```

Figure A.10 BasicPlayerMovement check section.

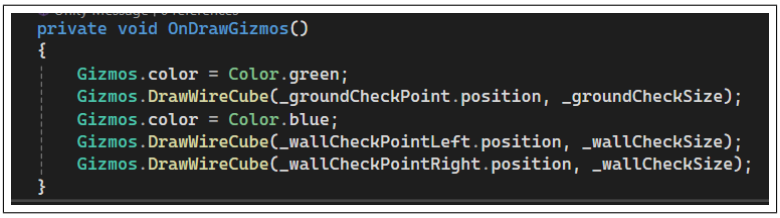
The ApplyCustomGravity() method adjusts gravity based on the player's state to create a more natural and dynamic feel for jumping and falling. If the player is hanging after a jump (when vertical velocity is minimal), gravity is reduced to make the jump feel more controlled. Conversely, when falling, gravity is increased to make the descent faster, and a maximum fall speed is enforced to prevent excessive speed. If the player is not dashing, normal gravity is applied.The HandleFriction() method manages the player's deceleration when on the ground and not moving. It applies a frictional force to slow down the player gradually, based on the current velocity. This makes the movement feel more realistic by preventing abrupt stops and ensuring smoother transitions between movement states.



```
1 reference
private void ApplyCustomGravity()
{
    if (IsJumping && Mathf.Abs(rb.linearVelocity.y) < 0.1f)
    {
        rb.gravityScale = gravityScale * jumpHangGravityMultiplier;
    }
    else if (!IsDashing && rb.linearVelocity.y < 0) // Apply increased gravity when falling
    {
        rb.gravityScale = gravityScale * fallMultiplier;
        rb.linearVelocity = new Vector2(rb.linearVelocity.x,Mathf.Max(rb.linearVelocity.y, -maxFallSpeed));
    }
    else if (!IsDashing)
    {
        rb.gravityScale = gravityScale;
    }
}

1 reference
private void HandleFriction()
{
    if (lastGroundedTime > 0 && Mathf.Abs(moveInput.x) < 0.01f)
    {
        float amount = Mathf.Min(Mathf.Abs(rb.linearVelocity.x), Mathf.Abs(frictionAmount));

        amount *= Mathf.Sign(rb.linearVelocity.x);

        rb.AddForce(Vector2.right * -amount, ForceMode2D.Impulse);
    }
}
```

Figure A.11 BasicPlayerMovement physics section.

The OnDrawGizmos() method is used for visual debugging in the Unity editor. It draws wireframe cubes at specific locations in the scene to represent the player's ground and wall detection areas. The green wireframe cube indicates the area where the player checks for ground, and the blue cubes represent the detection zones on either side of the player for wall detection. This helps visualize the range and orientation of the player's collision checks, making it easier to debug and fine-tune the movement and interaction mechanics during development.



```
private void OnDrawGizmos()
{
    Gizmos.color = Color.green;
    Gizmos.DrawWireCube(_groundCheckPoint.position, _groundCheckSize);
    Gizmos.color = Color.blue;
    Gizmos.DrawWireCube(_wallCheckPointLeft.position, _wallCheckSize);
    Gizmos.DrawWireCube(_wallCheckPointRight.position, _wallCheckSize);
}
```

Figure A.12 BasicPlayerMovement gizmos section.

A.3 PlayerAttack script

The initial section of the PlayerAttack script defines the core setup for managing attack behavior in the game. It includes serialized fields such as attackCooldown to control the delay between consecutive attacks, attackDamage to determine the amount of damage dealt, and enemyLayer to specify which objects can be damaged. The script also defines attackPoint as the origin for the attack hitbox and attackCapsuleSize along with capsuleDirection to shape the area of effect. A visual effect (attackVisual) can be shown when the player attacks. The PlayerManager reference is used to fetch the player's attack damage dynamically. A public property isAttacking tracks whether an attack is in progress, while attackInput and inputEnabled manage player input. In the Awake() method, the script initializes the PlayerManager reference and sets the attackDamage accordingly. Finally, the Update() method calls HandleInput() to check for attack input

```
public class PlayerAttack : MonoBehaviour
{
    [Header("Attack Settings")]
    [SerializeField] private float attackCooldown = 0.5f;
    [SerializeField] private int attackDamage;
    [SerializeField] private LayerMask enemyLayer;
    [SerializeField] private Transform attackPoint;
    [SerializeField] private Vector2 attackCapsuleSize = new Vector2(1.2f, 0.6f);
    [SerializeField] private CapsuleDirection2D capsuleDirection = CapsuleDirection2D.Horizontal;
    [SerializeField] private GameObject attackVisual;

    [SerializeField] private PlayerManager playerManager;

    4 references
    public bool isAttacking { get; private set; }

    private float lastAttackTime;
    public bool attackInput;
    public bool inputEnabled = true;

    Unity Message | 0 references
    private void Awake()
    {
        playerManager = GetComponent<PlayerManager>();
        attackDamage = playerManager.attackDamage;
    }

    Unity Message | 0 references
    void Update()
    {
        HandleInput();
    }
```

Figure A.13 PlayerAttack variables and initialize section.

This section handles player input for attacking. The HandleInput() method first checks whether input is currently enabled. If it is, it listens for a left mouse button click (Input.GetMouseButtonDown(0)) and sets attackInput accordingly. If an attack input is detected, it calls the PerformAttack() method to execute the attack logic. This design ensures that the player can only initiate attacks when allowed and that attack actions are triggered explicitly by user input.

```
private void HandleInput()
{
    if (!inputEnabled) return;

    attackInput = Input.GetMouseButtonDown(0);

    if (attackInput)
    {
        PerformAttack();
    }
}
```

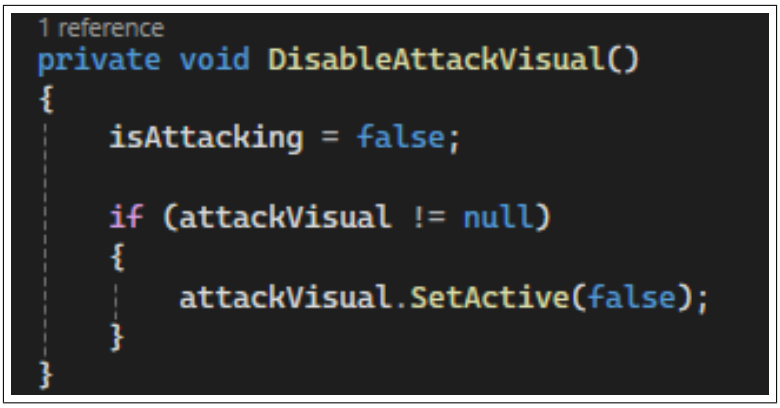
Figure A.14 PlayerAttack input section.

The PerformAttack() method manages the player's attack execution while respecting the cooldown period. It begins by checking whether enough time has passed since the last attack; if not, it exits early. If the cooldown has elapsed, it updates lastAttackTime, marks the player as currently attacking by setting isAttacking to true, and optionally shows a visual indicator (attackVisual) for a brief moment. It then performs a hit detection using Physics2D.OverlapCapsuleAll centered on the attackPoint, using the specified capsule size and direction to identify all enemies within range. For each detected enemy, it retrieves the EnemyActionModule component and calls its TakeDamage() method, passing in the damage value and the player's position for knockback calculation. A debug message is logged for each successful hit.

```
2 references
public void PerformAttack()
{
    if (Time.time < lastAttackTime + attackCooldown) return;

    lastAttackTime = Time.time;
    isAttacking = true;

    // Enable attack visual
    if (attackVisual != null)
    {
        attackVisual.SetActive(true);
        Invoke(nameof(DisableAttackVisual), 0.1f); // Hide after a short duration
    }

    // Detect enemies in range
    Collider2D[] hitEnemies = Physics2D.OverlapCapsuleAll(attackPoint.position, attackCapsuleSize, capsuleDirection, 0, enemyLayer);

    foreach (Collider2D enemyCollider in hitEnemies)
    {
        EnemyActionModule enemy = enemyCollider.GetComponent<EnemyActionModule>();
        if (enemy != null)
        {
            enemy.TakeDamage(attackDamage, transform.position); // Apply knockback from player position
            Debug.Log("Enemy hit and damaged!");
        }
    }
}
```

Figure A.15 PlayerAttack perform attack section.

The DisableAttackVisual() method resets the attack state after a short duration. It sets isAttacking to false and disables the attackVisual GameObject if it exists, effectively ending the visual and logical representation of an attack.



```
1 reference
private void DisableAttackVisual()
{
    isAttacking = false;

    if (attackVisual != null)
    {
        attackVisual.SetActive(false);
    }
}
```

Figure A.16 PlayerAttack disable attack visual section.

The OnDrawGizmos() method visually represents the player's attack range in the Unity editor by drawing a capsule at the attackPoint position. It calculates the start and end points of the capsule based on its direction (horizontal or vertical), draws circles at both ends to represent the capsule's curves, and connects them with lines to complete the capsule shape. This helps developers see and adjust the attack hitbox during development.

```
private void OnDrawGizmos()
{
    if (attackPoint == null) return;

    Gizmos.color = Color.red;

    Vector3 center = attackPoint.position;
    float capsuleRadius = attackCapsuleSize.y / 2;

    Vector3 capsuleStart, capsuleEnd;

    if (capsuleDirection == CapsuleDirection2D.Horizontal)
    {
        capsuleStart = center - Vector3.right * (attackCapsuleSize.x / 2 - capsuleRadius);
        capsuleEnd = center + Vector3.right * (attackCapsuleSize.x / 2 - capsuleRadius);
    }
    else
    {
        capsuleStart = center - Vector3.up * (attackCapsuleSize.y / 2 - capsuleRadius);
        capsuleEnd = center + Vector3.up * (attackCapsuleSize.y / 2 - capsuleRadius);
    }

    // Draw capsule end circles
    Gizmos.DrawWireSphere(capsuleStart, capsuleRadius);
    Gizmos.DrawWireSphere(capsuleEnd, capsuleRadius);

    // Connect circles with lines
    if (capsuleDirection == CapsuleDirection2D.Horizontal)
    {
        Gizmos.DrawLine(capsuleStart + Vector3.up * capsuleRadius, capsuleEnd + Vector3.up * capsuleRadius);
        Gizmos.DrawLine(capsuleStart - Vector3.up * capsuleRadius, capsuleEnd - Vector3.up * capsuleRadius);
    }
    else
    {
        Gizmos.DrawLine(capsuleStart + Vector3.right * capsuleRadius, capsuleEnd + Vector3.right * capsuleRadius);
        Gizmos.DrawLine(capsuleStart - Vector3.right * capsuleRadius, capsuleEnd - Vector3.right * capsuleRadius);
    }
}
```

Figure A.17 PlayerAttack gizmos section.

A.4 Agent Controller

The AgentController class inherits from Agent and integrates Unity ML-Agents with core gameplay components to enable intelligent behavior from the player character. It references two primary components via serialized fields: PlayerActionModules and PlayerManager, which likely handle character movement and overall state management, respectively. Additionally, it includes a list of spawnPointsList, a collection of potential spawn locations used to reset or relocate the agent. Internally, the class tracks the agent's last position and maintains a set called visitedAreas to record the discrete areas the agent has already explored, using Vector2Int for grid-based precision.

```
using System.Collections.Generic;
using Unity.MLAgents;
using Unity.MLAgents.Actuators;
using Unity.MLAgents.Sensors;
using UnityEngine;

Unity Script (1 asset reference) | 4 references
public class AgentController : Agent
{
    #region Fields & Components
    [SerializeField] private PlayerActionModules playerActionModules;
    [SerializeField] private PlayerManager playerManager;
    [SerializeField] private List<GameObject> spawnPointsList;
    private Vector2 lastPosition;
    private HashSet<Vector2Int> visitedAreas;
    #endregion
```

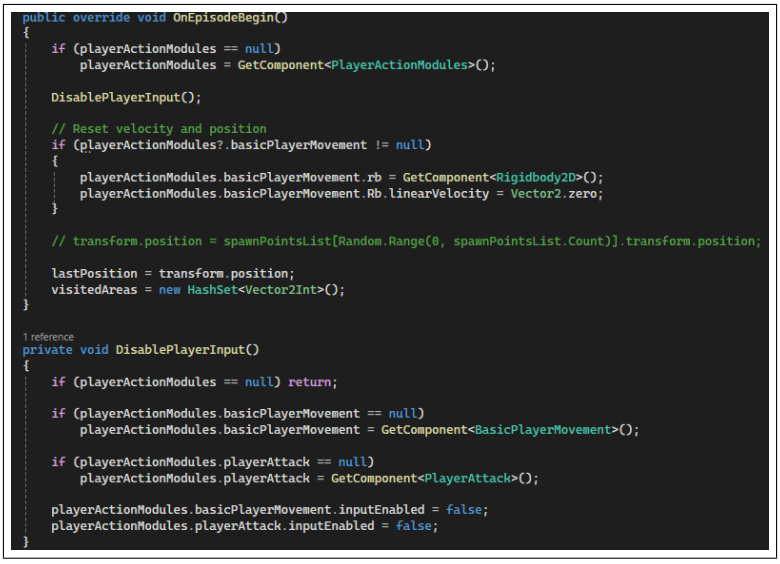
Figure A.18 AgentController fields and componenets section.

The Initialize method is an overridden function from the Agent class, responsible for setting up the agent's initial state when the simulation begins. It first ensures that the game time is running at a normal speed by setting Time.timeScale to 1. The method then checks if the playerActionModules and playerManager are assigned; if not, it attempts to fetch these components from the same GameObject. Finally, the method populates the spawnPointsList with all GameObjects tagged as "Spawn," utilizing GameObject.FindGameObjectsWithTag("Spawn") to gather a list of spawn locations across the scene.

```
public override void Initialize()
{
    Time.timeScale = 1;

    if (playerActionModules == null)
        playerActionModules = GetComponent<PlayerActionModules>();

    if (playerManager == null)
        playerManager = GetComponent<PlayerManager>();

    spawnPointsList = new List<GameObject>(GameObject.FindGameObjectsWithTag("Spawn"));
}
```

Figure A.19 AgentController Initialization section.

The OnEpisodeBegin method is another overridden function from the Agent class, which resets the agent's state at the beginning of each new training episode. It first ensures that the playerActionModules is assigned, and then proceeds to disable the player's input by calling DisablePlayerInput. The method also resets the agent's velocity and position, if the basicPlayerMovement module is available, it sets the Rigidbody2D's velocity to zero. Additionally, the method saves the agent's starting position in the lastPosition variable and initializes a new HashSet<Vector2Int> for tracking visited areas. This ensures that the agent's movement and environment interaction are properly reset at the beginning of the episode.The DisablePlayerInput method disables input processing for both the basicPlayerMovement and playerAttack modules by setting their inputEnabled flags to false. It checks if the necessary modules are already assigned and assigns them if needed, ensuring that no player input can be processed during training.



```
public override void OnEpisodeBegin()
{
    if (playerActionModules == null)
        playerActionModules = GetComponent<PlayerActionModules>();

    DisablePlayerInput();

    // Reset velocity and position
    if (playerActionModules?.basicPlayerMovement != null)
    {
        playerActionModules.basicPlayerMovement.rb = GetComponent<Rigidbody2D>();
        playerActionModules.basicPlayerMovement.Rb.linearVelocity = Vector2.zero;
    }

    // transform.position = spawnPointsList[Random.Range(0, spawnPointsList.Count)].transform.position;

    lastPosition = transform.position;
    visitedAreas = new HashSet<Vector2Int>();
}

1 reference
private void DisablePlayerInput()
{
    if (playerActionModules == null) return;

    if (playerActionModules.basicPlayerMovement == null)
        playerActionModules.basicPlayerMovement = GetComponent<BasicPlayerMovement>();

    if (playerActionModules.playerAttack == null)
        playerActionModules.playerAttack = GetComponent<PlayerAttack>();

    playerActionModules.basicPlayerMovement.inputEnabled = false;
    playerActionModules.playerAttack.inputEnabled = false;
}
```

Figure A.20 AgentController Episode Handling section.

The CollectObservations method is part of Unity's ML-Agents framework and is used to collect observations from the environment that the agent will use to make decisions. In this implementation, the method gathers a series of observations regarding the player's current state, which will be passed to the reinforcement learning model to help it learn how to act in the environment.

First, the method checks if the basicPlayerMovement component is available. If not, it adds a fallback observation (0) and returns. If basicPlayerMovement is present, it proceeds to collect various observations. These observations include the player's position (x and y coordinates), movement states (whether the player is grounded, jumping, facing right, dashing, dropping, on a wall, or wall-jumping), and the player's current health.The method also collects information about the player's attack state, adding an observation that indicates whether the player is attacking. All these observations are added to the sensor, which is used by the agent to inform its decision-making during training.This method enables the reinforcement learning model to track the player's state, facilitating better decision-making based on these environmental factors.

```
public override void CollectObservations(VectorSensor sensor)
{
    var movement = playerActionModules?.basicPlayerMovement;

    if (movement == null)
    {
        sensor.AddObservation(0f); // fallback values
        return;
    }

    // Position
    sensor.AddObservation(transform.position.x);
    sensor.AddObservation(transform.position.y);

    // Movement States
    sensor.AddObservation(movement.IsGrounded ? 1f : 0f);
    sensor.AddObservation(movement.IsJumping ? 1f : 0f);
    sensor.AddObservation(movement.IsFacingRight ? 1f : 0f);
    sensor.AddObservation(movement.IsDashing ? 1f : 0f);
    sensor.AddObservation(movement.IsDropping ? 1f : 0f);
    sensor.AddObservation(movement.IsOnWall ? 1f : 0f);
    sensor.AddObservation(movement.IsWallJumping ? 1f : 0f);

    // Health
    sensor.AddObservation(playerManager.currentHealth);

    // Attacking States
    sensor.AddObservation(playerActionModules.playerAttack.isAttacking ? 1f : 0f);
}
```

Figure A.21 AgentController Observation section.

The OnActionReceived method in the AgentController script processes the actions received by the reinforcement learning agent. It first extracts continuous and discrete actions from the ActionBuffers parameter, interpreting the values to control various player behaviors. The continuous action, actions.ContinuousActions[0], represents the player's horizontal movement and is clamped to the range of -1f to 1f. The discrete actions correspond to jump, dash, attack, and drop, with a value of 1 indicating that the respective action should be performed. The method then delegates the appropriate actions to the playerActionModules, calling methods like Move(), Jump(), Dash(), Attack(), and Drop() based on the received input. Finally, it calls the EvaluateRewards() method to assess the agent's performance, which is crucial for the reinforcement learning model to determine the effectiveness of the actions and adjust accordingly.

```
public override void OnActionReceived(ActionBuffers actions)
{
    float moveX = Mathf.Clamp(actions.ContinuousActions[0], -1f, 1f);
    bool jumpAction = actions.DiscreteActions[0] == 1;
    bool dashAction = actions.DiscreteActions[1] == 1;
    bool attackAction = actions.DiscreteActions[2] == 1;
    bool dropAction = actions.DiscreteActions[3] == 1;

    // Delegate input
    playerActionModules.Move(moveX);
    if (jumpAction) playerActionModules.Jump();
    if (dashAction) playerActionModules.Dash();
    if (attackAction) playerActionModules.Attack();
    if (dropAction) playerActionModules.Drop();

    EvaluateRewards();
}
```

Figure A.22 AgentController Action section.

The EvaluateRewards method in the AgentController script is responsible for assigning rewards based on the agent's actions and progress within the game environment. The method first calculates the distance the agent has moved since the last update using Vector2.Distance. If the agent has moved a significant distance (greater than 0.1 units), it rewards the agent with a value proportional to the distance moved (0.05 times the distance). If the movement is negligible, it penalizes the agent with a small negative reward of -0.01. This encourages the agent to make meaningful progress rather than staying idle. The method then updates lastPosition to the current position of the agent.Next, the method checks if the agent has visited a new area. It converts the agent's position to a Vector2Int grid coordinate and checks if this position has been visited before using the visitedAreas set. If the agent has not visited the current grid position, it rewards the agent with 0.2 units and adds the grid position to the visitedAreas set to track exploration. This exploration reward encourages the agent to move to new areas and explore the environment.

```
private void EvaluateRewards()
{
    // Distance Reward
    float distanceMoved = Vector2.Distance(transform.position, lastPosition);
    if (distanceMoved > 0.1f)
        AddReward(0.05f * distanceMoved);
    else
        AddReward(-0.01f);

    lastPosition = transform.position;

    // Exploration Reward
    Vector2Int gridPos = new Vector2Int(Mathf.RoundToInt(transform.position.x), Mathf.RoundToInt(transform.position.y));
    if (!visitedAreas.Contains(gridPos))
    {
        AddReward(0.2f);
        visitedAreas.Add(gridPos);
    }
}
```

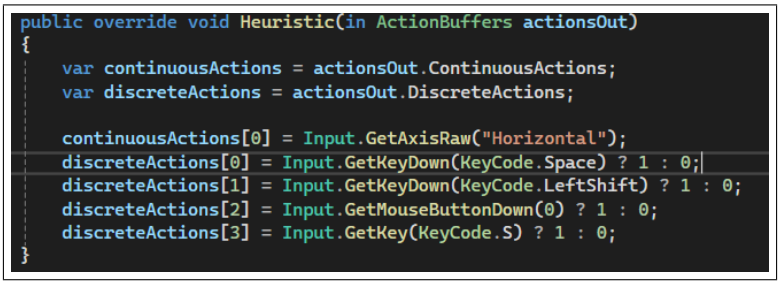
Figure A.23 AgentController Reward section.

The OnTriggerEnter2D method in the AgentController script is responsible for handling the agent's interactions with different colliders during the game. When the agent collides with an object tagged with a specific label, the method uses a switch statement to determine the type of object and apply appropriate rewards or penalties. If the agent collides with an object tagged as "Coins", it rewards the agent with 1.0 unit, encouraging it to collect coins. If the agent collides with an object tagged as "Hazard", it penalizes the agent with -3.0 units and ends the current episode, indicating that a hazard results in a significant penalty and failure. If the agent encounters an "Enemy", it applies a moderate penalty of -2.0 units, discouraging contact with enemies. Lastly, if the agent reaches a "Checkpoint", it rewards the agent with a significant 5.0 unit, incentivizing progress through the game. This method helps guide the agent's behavior by providing feedback based on its interactions with key game elements.

```
private void OnTriggerEnter2D(Collider2D collision)
{
    switch (collision.tag)
    {
        case "Coins":
            AddReward(1.0f);
            break;

        case "Hazard":
            AddReward(-3.0f);
            EndEpisode();
            break;

        case "Enemy":
            AddReward(-2.0f);
            break;

        case "Checkpoint":
            AddReward(5.0f);
            break;
    }
}
```

Figure A.24 AgentController Collision section.

The Heuristic method in the AgentController script provides a manual control scheme for the agent using player input, primarily for testing and debugging in the Unity Editor. This method overrides the default Heuristic function from the ML-Agents framework and populates the actionsOut parameter with values derived from player input. It first accesses both the ContinuousActions and DiscreteActions buffers. The ContinuousActions[0] is set using Input.GetAxisRaw(”Horizontal”), which captures left or right movement from keyboard inputs like the arrow keys or A/D. Then, the DiscreteActions are assigned as follows: discreteActions[0] activates when the spacebar is pressed, triggering a jump; discreteActions[1] activates when the Left Shift key is pressed, initiating a dash; discreteActions[2] triggers an attack when the left mouse button is clicked; and discreteActions[3] becomes true when the S key is held down, initiating a drop-through platform action. This method allows the developer to control and test the agent manually, bypassing the AI’s decision-making during training or demonstrations.



```
public override void Heuristic(in ActionBuffers actionsOut)
{
    var continuousActions = actionsOut.ContinuousActions;
    var discreteActions = actionsOut.DiscreteActions;

    continuousActions[0] = Input.GetAxisRaw("Horizontal");
    discreteActions[0] = Input.GetKeyDown(KeyCode.Space) ? 1 : 0;
    discreteActions[1] = Input.GetKeyDown(KeyCode.LeftShift) ? 1 : 0;
    discreteActions[2] = Input.GetMouseButtonDown(0) ? 1 : 0;
    discreteActions[3] = Input.GetKey(KeyCode.S) ? 1 : 0;
}
```

Figure A.25 AgentController Heuristic section.

A.5 Enemy Script

The Enemy class represents a basic enemy character in the game, managing its health, invincibility state, and reactions to damage. It begins with serialized fields for maxHealth, a damageCooldown duration during which the enemy becomes invincible after being hit, and a knockbackForce that defines how strongly it is pushed upon taking damage. The currentHealth tracks the enemy’s remaining life points, while isInvincible and invincibilityTimer control temporary immunity to repeated attacks. In Awake(), the enemy initializes its health and retrieves its Rigidbody2D and associated EnemyActionModule if not already set.

In the Update() loop, if the enemy is currently invincible, it decrements the timer until invincibility wears off. The TakeDamage() method handles damage intake, it only processes if the enemy isn’t invincible. Upon being hit, the enemy reduces its health, triggers invincibility, applies knockback using the source of the attack, and checks if it should die. The ApplyKnockback() method calculates a direction vector away from the attacker and applies an impulse force to the Rigidbody2D to push the enemy back. Finally, if the enemy’s health drops to zero or below, it calls Die(), which invokes the Die() function from the action module and destroys the game object.

```
using UnityEngine;

Unity Script | 2 references
public class Enemy : MonoBehaviour
{
    [SerializeField] private int maxHealth = 3;
    [SerializeField] private float damageCooldown = 0.5f;
    [SerializeField] private float knockbackForce = 5f;

    [SerializeField] private int currentHealth;
    [SerializeField] private bool isInvincible;
    [SerializeField] private float invincibilityTimer;
    private Rigidbody2D rb;

    public EnemyActionModule actionModule;

    Unity Message | 0 references
    private void Awake()
    {
        currentHealth = maxHealth;
        rb = GetComponent<Rigidbody2D>();
        if (actionModule == null)
            actionModule = GetComponent<EnemyActionModule>();
    }

    Unity Message | 0 references
    private void Update()
    {
        if (isInvincible)
        {
            invincibilityTimer -= Time.deltaTime;
            if (invincibilityTimer <= 0)
                isInvincible = false;
        }
    }

    1 reference
    public void TakeDamage(int amount, Vector2 hitSource)
    {
        if (isInvincible) return;

        currentHealth -= amount;
        isInvincible = true;
        invincibilityTimer = damageCooldown;

        ApplyKnockback(hitSource);

        if (currentHealth <= 0)
        {
            actionModule.Die();
        }
    }

    1 reference
    private void ApplyKnockback(Vector2 hitSource)
    {
        Debug.LogWarning("Knockback");
        if (rb == null) return;

        Vector2 dir = ((Vector2)transform.position - hitSource).normalized;
        rb.AddForce(dir * knockbackForce, ForceMode2D.Impulse);
    }

    1 reference
    public void Die()
    {
        Destroy(gameObject);
    }
}
```

Figure A.26 Enemy Scripts.

A.6 Enemy Action Module

The EnemyActionModule class acts as an intermediary that handles the enemy's interaction with the reinforcement learning agent (AgentController) and delegates core combat functionality to the associated Enemy component. In Awake(), it ensures the enemyCombat reference is assigned, typically pointing to the same game object's Enemy component.The TakeDamage() method is called when the enemy is struck. Before delegating the damage logic to enemyCombat, it locates an AgentController in the scene and rewards the agent with +1.0 reward, reinforcing the behavior of attacking enemies. After that, it calls the TakeDamage() method of the Enemy class to apply damage, knockback, and health management.The Die() method is responsible for handling enemy death events. When invoked, it also finds the AgentController and rewards it with +5.0, incentivizing successful eliminations. Then it calls the Die() function of the Enemy class to destroy the enemy object.

```
using UnityEngine;

Unity Script (1 asset reference) | 4 references
public class EnemyActionModule : MonoBehaviour
{
    public Enemy enemyCombat;

    Unity Message | 0 references
    private void Awake()
    {
        if (enemyCombat == null)
            enemyCombat = GetComponent<Enemy>();
    }

    1 reference
    public void TakeDamage(int amount, Vector2 hitSource)
    {
        AgentController agent = FindFirstObjectByType<AgentController>();
        if (agent != null)
        {
            agent.AddReward(1.0f);
        }
        if (enemyCombat != null)
            enemyCombat.TakeDamage(amount, hitSource);
    }

    1 reference
    public void Die()
    {
        AgentController agent = FindFirstObjectByType<AgentController>();
        if (agent != null)
        {
            agent.AddReward(5.0f);
        }

        if (enemyCombat != null)
            enemyCombat.Die();
    }
}
```

Figure A.27 Enemy Action Module Scripts.

A.7 Enemy State Machine

This script defines the basic structure for an Enemy State Machine in Unity, which controls how an enemy behaves based on different conditions. The EnemyStateMachine class includes three defined states: Patrol, Idle, and Chase, managed by an enum called EnemyState. For patrolling, the enemy moves between a series of waypoints (patrolPoints) at a given speed. The chase behavior activates when the player enters a specified detectionRange, causing the enemy to move faster (chaseSpeed) toward the player (playerPosition). The state machine uses references to the player's movement script (BasicPlayerMovement) and a Rigidbody2D component for motion control. A sprite GameObject is also included, likely for visual adjustments such as flipping the sprite direction.

```
using UnityEngine;

Unity Script (1 asset reference) | 0 references
public class EnemyStateMachine : MonoBehaviour
{
    7 references
    public enum EnemyState { Patrol, Idle, Chase }

    [Header("Patrol Settings")]
    public float speed = 2f;
    public Transform[] patrolPoints;
    private int currentPointIndex = 0;

    [Header("Chase Settings")]
    public float chaseSpeed = 3f;
    public float detectionRange = 5f;
    public BasicPlayerMovement playerObj;
    public Transform playerPosition;
    public GameObject detected;

    [Header("State Machine")]
    public EnemyState currentState = EnemyState.Patrol;

    private Rigidbody2D rb;
    public GameObject sprite;
```

Figure A.28 Enemy State Machine variables section.

This section of the EnemyStateMachine script sets up the enemy's runtime behavior and updates its state each frame. In the Start() method, the enemy's Rigidbody2D is initialized, and rotation is frozen to prevent unwanted spinning. It also locates the player object using FindFirstObjectByType<BasicPlayerMovement>() and assigns the player's Transform to playerPosition, which is crucial for the chase logic. A warning is logged if no patrol points are provided, helping developers catch setup issues early. The Update() method acts as the core of the state machine. It checks the current state of the enemy (Patrol, Chase, or Idle) and executes the appropriate behavior. In Patrol, the enemy follows a set of points and also checks if the player has entered detection range. In Chase, the enemy actively pursues the player and continuously checks if the player has moved out of range to potentially return to patrolling. In the Idle state, the enemy remains stationary by setting its velocity to zero. This structure ensures that each behavior is clearly separated and triggered based on state conditions.

```
void Start()
{
    rb = GetComponent<Rigidbody2D>();
    rb.freezeRotation = true;

    playerObj = FindFirstObjectByType<BasicPlayerMovement>();
    if (playerObj != null)
        playerPosition = playerObj.transform;

    if (patrolPoints.Length == 0)
        Debug.LogWarning("No patrol points assigned!");
}

Unity Message | 0 references
void Update()
{
    switch (currentState)
    {
        case EnemyState.Patrol:
            Patrol();
            CheckForPlayer();
            break;

        case EnemyState.Chase:
            ChasePlayer();
            CheckForPlayerOutOfRange();
            break;

        case EnemyState.Idle:
            rb.linearVelocity = Vector2.zero;
            break;
    }
}
```

Figure A.29 Enemy State Machine initialization section.

This part of the EnemyStateMachine script defines how the enemy moves during both patrol and chase behaviors, as well as how it visually responds to direction changes. The Patrol() method is responsible for basic waypoint navigation. The enemy moves toward the current patrol point by calculating the normalized direction vector and applying horizontal velocity using the patrol speed. When the enemy is close enough to a patrol point (within 0.2 units), it cycles to the next point using modulo arithmetic to loop through the array. The sprite is then flipped to face the movement direction using the FlipSprite() method.The ChasePlayer() method operates similarly, but instead of heading toward patrol points, the enemy calculates direction based on the player's position. It moves using a faster chaseSpeed to give the player a sense of urgency when detected. Again, the sprite is flipped to face the direction of movement for visual consistency.FlipSprite(float moveDirection) ensures the enemy's sprite visually faces the direction it's moving. It checks the horizontal movement direction and inverts the local scale's x-axis if needed, this prevents the enemy from moonwalking or moving backward visually while chasing or patrolling.

```
1 reference
void Patrol()
{
    if (patrolPoints.Length == 0) return;

    Transform targetPoint = patrolPoints[currentPointIndex];
    Vector2 direction = (targetPoint.position - transform.position).normalized;
    rb.linearVelocity = new Vector2(direction.x * speed, rb.linearVelocity.y);

    if (Vector2.Distance(transform.position, targetPoint.position) < 0.2f)
    {
        currentPointIndex = (currentPointIndex + 1) % patrolPoints.Length;
    }

    FlipSprite(direction.x);
}

1 reference
void ChasePlayer()
{
    if (playerPosition == null) return;

    Vector2 direction = (playerPosition.position - transform.position).normalized;
    rb.linearVelocity = new Vector2(direction.x * chaseSpeed, rb.linearVelocity.y);

    FlipSprite(direction.x);
}

2 references
void FlipSprite(float moveDirection)
{
    if (moveDirection != 0 && sprite != null)
    {
        Vector3 scale = sprite.transform.localScale;
        if ((moveDirection < 0 && scale.x > 0) || (moveDirection > 0 && scale.x < 0))
        {
            scale.x *= -1;
            sprite.transform.localScale = scale;
        }
    }
}
```

Figure A.30 Enemy State Machine State machine logic section.

These two methods, CheckForPlayer() and CheckForPlayerOutOfRange(), control the transitions between the patrol and chase states in the EnemyStateMachine by monitoring the distance between the enemy and the player.CheckForPlayer() is used during the patrol state to detect when the player enters the enemy's detection range. If the player's distance from the enemy is less than or equal to the defined detectionRange, the enemy enters the Chase state. Optionally, a visual indicator (like an exclamation mark or alert icon stored in the detected GameObject) is activated to signal the change in behavior.On the other hand, CheckForPlayerOut-OfRange() is called during the chase state to determine when the player has moved far enough away that they are no longer considered a target. If the player moves beyond the detection range, the alert indicator is turned off and the enemy returns to the Patrol state.Together, these methods allow the enemy to dynamically switch between patrolling and actively chasing the player based on proximity, enabling more reactive and engaging AI behavior.

```
void CheckForPlayer()
{
    if (playerPosition == null) return;

    if (Vector2.Distance(transform.position, playerPosition.position) <= detectionRange)
    {
        detected?.SetActive(true);
        currentState = EnemyState.Chase;
    }
}

1 reference
void CheckForPlayerOutOfRange()
{
    if (playerPosition == null) return;

    if (Vector2.Distance(transform.position, playerPosition.position) > detectionRange)
    {
        detected?.SetActive(false);
        currentState = EnemyState.Patrol;
    }
}
```

Figure A.31 Enemy State Machine check player section.

This OnCollisionEnter2D method handles the interaction between the enemy and the player when a physical collision occurs.When the enemy collides with another object, the method checks if the object has the "Player" tag using CompareTag("Player"). If it does, the method attempts to retrieve the PlayerManager component from the player GameObject. If the component exists, it calls playerManager.Die(), triggering whatever logic is implemented in the player's death function, typically ending the level, restarting from a checkpoint, or triggering a death animation.This mechanic is useful for enemies that deal instant or contact-based damage, like spikes, patrolling robots, or charging creatures. It makes the enemy more threatening and reinforces the need for players to avoid direct contact.

```
private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.CompareTag("Player"))
    {
        PlayerManager playerManager = collision.gameObject.GetComponent<PlayerManager>();
        if (playerManager != null)
        {
            playerManager.Die();
        }
    }
}
```

Figure A.32 Enemy State Machine collision check section.

A.8 Moving Platform script

The MovingPlatform script defines a simple back-and-forth movement system for a platform in Unity, typically used in 2D platformer games. The platform moves smoothly between two specified points, pointA and pointB, using Vector3.MoveTowards at a given speed (moveSpeed). When the platform reaches one point, it switches its target to the other, allowing for continuous oscillation.

Additionally, the script handles player interactions: when the player collides with the platform (OnCollisionEnter2D), the player's transform is parented to the platform, making the player move along with it seamlessly. This prevents the player from sliding off due to physics. At the same time, it disables Rigidbody interpolation for smoother motion syncing. Once the player leaves the platform (OnCollisionExit2D), the parent is reset to null, and interpolation is re-enabled to maintain physics consistency during free movement.

```
public class MovingPlatform : MonoBehaviour
{
    public Transform pointA;
    public Transform pointB;
    public float moveSpeed = 2f;

    private Vector3 nextPosition;

    // Start is called once before the first execution of Update after the MonoBehaviour is created
    Unity Message | 0 references
    void Start()
    {
        nextPosition = pointB.position;
    }

    // Update is called once per frame
    Unity Message | 0 references
    void Update()
    {
        transform.position = Vector3.MoveTowards(transform.position, nextPosition, moveSpeed * Time.deltaTime);

        if(transform.position == nextPosition)
        {
            nextPosition = (nextPosition == pointA.position) ? pointB.position : pointA.position;
        }
    }

    Unity Message | 0 references
    private void OnCollisionEnter2D(Collision2D collision)
    {
        if (collision.gameObject.CompareTag("Player"))
        {
            collision.gameObject.transform.parent = transform;
            Rigidbody2D playerRb = collision.gameObject.GetComponent<Rigidbody2D>();

            playerRb.interpolation = RigidbodyInterpolation2D.None;
        }
    }

    Unity Message | 0 references
    private void OnCollisionExit2D(Collision2D collision)
    {
        if (collision.gameObject.CompareTag("Player"))
        {
            collision.gameObject.transform.parent = null;
            Rigidbody2D playerRb = collision.gameObject.GetComponent<Rigidbody2D>();

            playerRb.interpolation = RigidbodyInterpolation2D.Interpolate;
        }
    }
```

Figure A.33 Moving Platform script.