# DEVELOPMENT OF A 2D PLATFORMER GAME AND MACHINE LEARNING MODEL

by

MATHEW MICHAEL DAWSON
URN: 6743842

A dissertation submitted in partial fulfilment of the requirements for the award of

BACHELOR OF SCIENCE IN COMPUTER SCIENCE

May 2025

Department of Computer Science
University of Surrey
Guildford GU2 7XH

Supervised by: Daniel Gardham

I declare that this dissertation is my own work and that the work of others is acknowledged and indicated by explicit references.

Mathew Michael Dawson
May 2025

# Abstract

This dissertation explores deep reinforcement learning in video games by developing a Deep Q-Network (DQN) agent capable of autonomously playing a custom 2D platformer game. The game is built using the Godot, a lightweight open-source game engine, and the model is implemented using the PyTorch deep learning framework. The model is lightweight enough to run on the CPU, but can be accelerated using a GPU. Experimental results show the DQN agent successfully learned effective strategies, achieving times comparable to human players. The agent's performance improved with training, demonstrating the potential of deep reinforcement learning in video games.

# Acknowledgements

I would like to thank my supervisor Dr Daniel Gardham for being on board with my idea from our first meeting, as well as his continued guidance and support, had it not been for him I would likely have picked something much more straightforward and within my comfort zone. I would also like to thank my friends and family for encouraging me along the development of this project. Thanks also go to the Godot contributors who work tirelessly, for free, on this wonderful game engine.

# Contents

# List of Figures

# List of Tables

# Abbreviations

| | |
|---|---|
| AI | Artificial Intelligence |
| DQN | Deep Q Network |
| ML | Machine Learning |
| RL | Reinforcement Learning |
| DRL | Deep Reinforcement Learning |
| MDP | Markov Decision Process |
| NPC | Non-Player Character |
| FSM | Finite State Machine |
| BT | Behaviour Tree |
| GOAP | Goal-Oriented Action Planning |
| PPO | Proximal Policy Optimisation |
| A3C | Asynchronous Advantage Actor-Critic |
| CPU | Central Processing Unit |
| GPU | Graphics Processing Unit |

# Chapter 1

# Introduction

## 1.1 Chapter Overview

This chapter will focus on introducing the project with an overview along with its objectives and limitations.

## 1.2 Project Background

In the area of video games, Artificial Intelligence (AI) has been used for many years to create non-player characters (NPCs) that can interact with players in a believable way. This was first seen in the game "Nim" in 1948 (Wikipedia 2025). This is often done using finite state machines (FSMs) or behaviour trees, which allow NPCs to react to player actions in a way that seems intelligent (Carpenter 2019). However, these methods can be limited in their ability to adapt to new situations, due to being based on pre-defined rules and behaviours. For example, developing an FSM for a procedurally generated game, or one with another amount of randomness involved, can be difficult or even impossible, as the FSM must be able to handle all possible situations that may arise. This is the problem that I will be experimenting with and attempting to address in this project. Reinforcement Learning (RL) provides an alternative approach to video game AI. Unlike traditional methods, RL agents learn through interaction with their environment by receiving rewards for desirable actions. This allows them to develop adaptive strategies without explicitly programmed rules. Building upon RL, Deep Reinforcement Learning (DRL) combines traditional RL algorithms with deep neural networks, enabling agents to process complex visual inputs and learn effective policies from high-dimensional data. DRL has demonstrated remarkable capabilities in video games, as seen in systems like OpenAI's DQN that mastered Atari games and AlphaGo which defeated world champions in Go. The adaptive nature of DRL makes it particularly promising for procedurally generated or dynamic game environments where traditional AI approaches struggle.

## 1.3 Project Overview

This project will begin with research and literature review into traditional approaches to AI in video games, as well as reinforcement learning and deep reinforcement learning techniques. Following this research phase, a simple game environment will be developed using the Godot game engine, designed specifically to test and showcase the capabilities of an RL agent. The project will then implement and train an RL agent to operate within this environment, focusing on creating NPCs that can learn and adapt to changes within the game rather than following predetermined patterns. The performance of these agents will be evaluated against traditional AI methods, and the final implementation will include an RL agent as some part of the game.

## 1.4 Project Aim and Objectives

The primary aim of this project be to attempt a non-standard machine learning based apporach to video game AI.

- Research and understand how existing AI in video games works, then the fundamentals of Reinforcement Learning (RL) and its application in video games.
- Design and implement a video game environment suitable for testing RL agents.
- Develop and train an RL agent to interact with the game.
- Evaluate the performance of the RL agent.
- Implement the RL agent as part of the game, and have it interact with players.

## 1.5 Limitations

This project will have the following limitations:

- The game will be custom made, rather than an existing one.
- The game will have simple graphics and gameplay, acting more as a "front-end" for the model and it's integration which will be the main focus of the project.
- The model will be lightweight, and will need to run on a single CPU and/or GPU. It must also not consume too much memory. This is to ensure that I can train and run it on my hardware.

# Chapter 2

# Literature Review

This chapter contains a review of traditional approaches to AI in video games. It then explores RL and deep learning fundamentals, algorithms and data structures, as well as some of their existing applications to video games.

## 2.1 Traditional approaches to AI in video games

This section explores some existing traditional methods of implementing AI in video games.

### 2.1.1 Finite State Machines

A Finite State Machine (FSM) is a computational model that has been foundational in video game AI development for decades. FSMs represent an agent's behavior as a set of discrete states, with well-defined transitions between these states triggered by specific conditions or events (Spiceworks 2021).

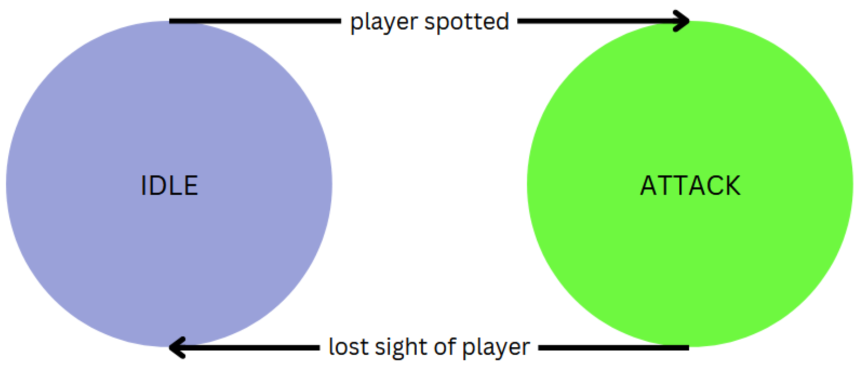


Figure 2.1: A simple FSM for an enemy NPC

Each state encapsulates a particular behavior or action pattern, while transitions define the rules governing when an agent should change its behavior. The simplicity and predictability of FSMs make them particularly suitable for controlling NPCs with straightforward behavioral patterns, as they are computationally efficient and easily debuggable. However, FSMs face significant limitations as complexity increases: the number of states and transitions can grow exponentially, leading to the "state explosion" problem that makes maintenance challenging. Additionally, FSMs struggle with handling concurrent behaviors and can appear rigid when compared to more dynamic AI approaches. Despite these limitations, FSMs remain prevalent in game development due to their intuitive implementation and reliable performance for many common AI tasks.

### 2.1.2 Behaviour Trees

Behaviour Trees (BTs) represent a significant advancement over FSMs in game AI architecture, offering a hierarchical, modular approach to decision-making. Originally developed for robotics and adopted by the game industry in titles like Halo 2 (Carpenter 2019), BTs organize agent behaviors into a tree structure where leaf nodes represent atomic actions and internal nodes control flow through various composites such as sequences, selectors, and parallels. This structure enables developers to create complex, reusable behavior patterns that can be visually represented and intuitively understood. Unlike FSMs, BTs naturally handle concurrent actions and gracefully manage behavior prioritization through their hierarchical evaluation. BTs excel at creating responsive AI that can react to changing game conditions while maintaining coherent behavior patterns. They facilitate an incremental development approach, allowing designers to progressively refine AI by adding branches without disrupting existing functionality. While BTs require more initial design consideration than FSMs, their scalability, maintainability, and ability to represent sophisticated decision-making logic have made them the standard approach for contemporary game AI systems, particularly in action, strategy, and open-world games where adaptable NPC behavior is critical to player experience.

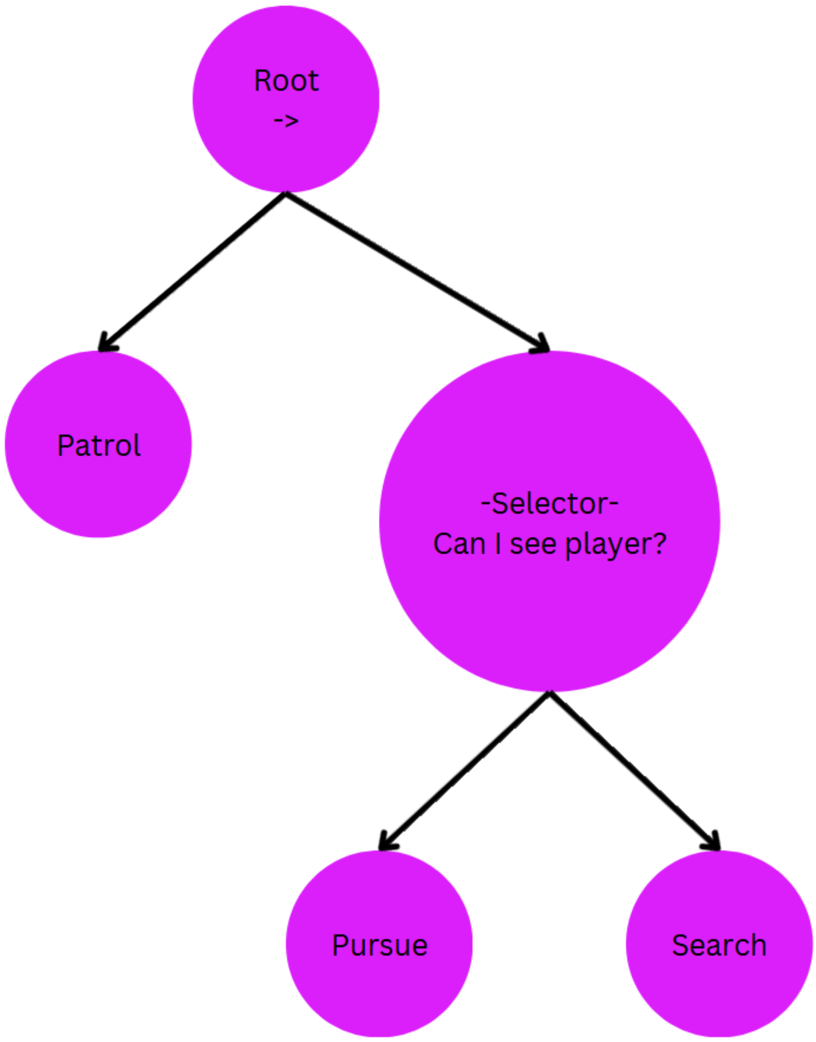


Figure 2.2: A simple BT for an enemy NPC

### 2.1.3 Goal-Oriented Action Planning

Goal-Oriented Action Planning (GOAP) represents a more dynamic approach to AI decision-making compared to FSMs and BTs. GOAP employs principles from automated planning and means-end analysis to create AI agents that formulate plans to achieve specific goals. Unlike more rigid systems, GOAP agents dynamically determine action sequences by considering current world states, available actions with preconditions and effects, and desired goal states. In a GOAP system, each action is associated with both preconditions (requirements that must be satisfied before the action can be taken) and effects (how the action changes the world state). The AI agent uses planning algorithms, commonly A* search, to find the optimal sequence of actions that transforms the current state into the goal state. This approach allows NPCs to solve problems creatively and adapt to unexpected changes in the game environment. GOAP gained prominence through its implementation in F.E.A.R. (2005) (Thompson 2020), where it produced enemies capable of contextually appropriate tactical behaviors like seeking cover, flanking the player, and coordinating with allies. The system's strength lies in its separation of goals (what the agent wants to achieve) from the specific methods to achieve them, creating emergent behavior that can surprise even the developers. While GOAP offers exceptional adaptability and problem-solving capabilities, it comes with higher computational costs and increased implementation complexity compared to FSMs and BTs. Despite these challenges, GOAP remains valuable for games requiring sophisticated AI that can respond intelligently to dynamic and unpredictable gameplay scenarios.

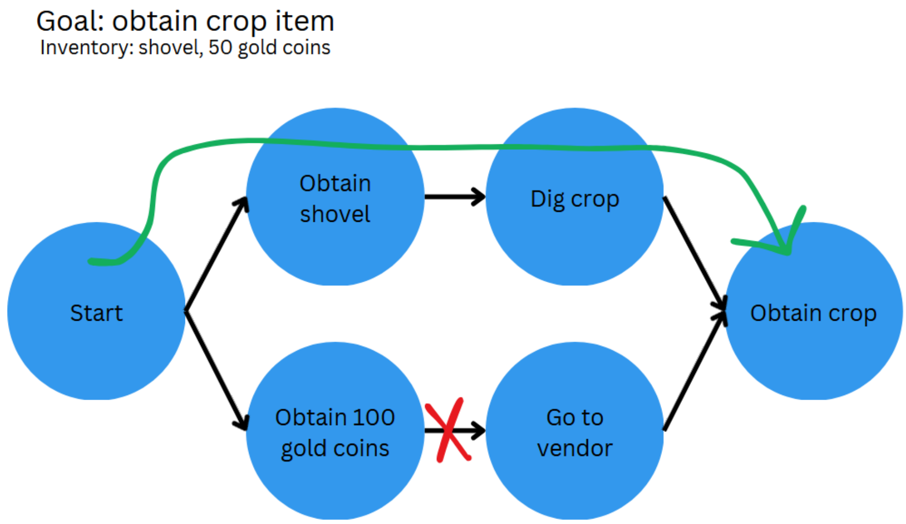


Figure 2.3: Example of a GOAP planning sequence

## 2.2 Reinforcement Learning

This section explores fundamental RL concepts and algorithms.

### 2.2.1 RL Fundamentals and the Markov Decision Process

Reinforcement Learning (RL) represents a departure from traditional game AI approaches by focusing on learning optimal behaviors through trial and error interaction with an environment. Unlike FSMs, BTs, or GOAP systems that rely on pre-programmed rules, RL agents improve their decision-making capabilities through experience.

At its core, RL is formalized as a Markov Decision Process (MDP) consisting of:

- A set of states $S$ representing all possible situations an agent may encounter
- A set of actions $A$ that the agent can take
- Transition probabilities $P(s'|s,a)$ defining the likelihood of moving to state $s'$ after taking action $a$ in state $s$
- A reward function $R(s,a,s')$ providing feedback on the quality of decisions

- A discount factor $\gamma \in [0, 1]$ determining the importance of future rewards

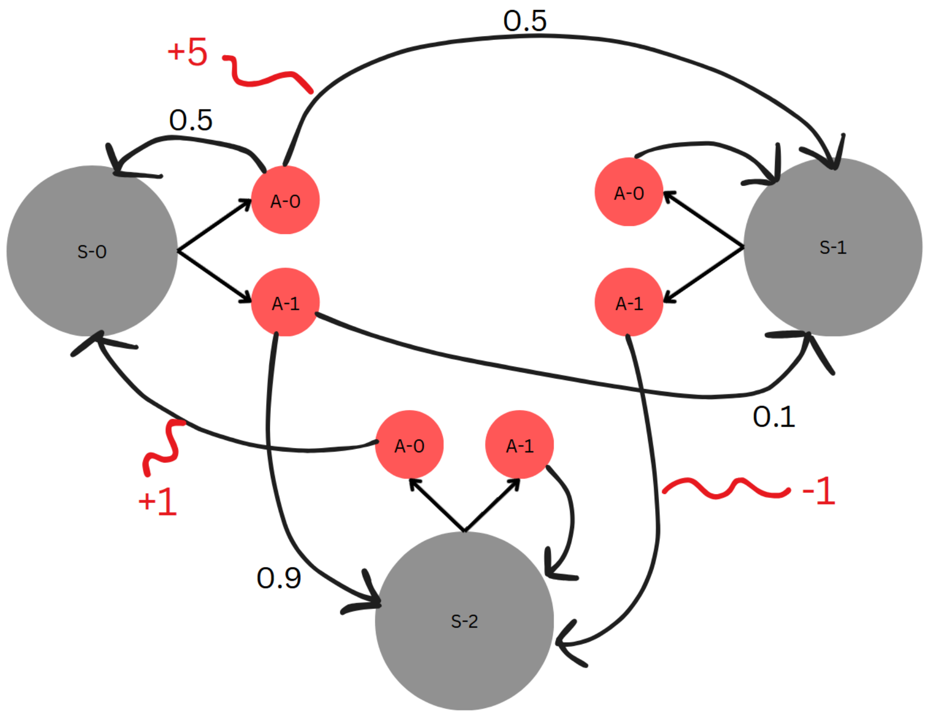


Figure 2.4: A simple MDP Graph example

S Nodes represent states and A nodes represent actions. Some actions can result in more than one state, the transition probabilities are marked in black. Some actions result in a positive or negative reward, marked in red.

The agent's objective is to learn a policy $\pi$ that maps states to actions in a way that maximizes the expected cumulative reward. This optimisation process balances immediate rewards against long-term consequences, addressing the fundamental exploration-exploitation dilemma: whether to capitalize on known good strategies or explore new possibilities that might yield better results. RL algorithms generally fall into three categories: value-based methods (like Q-learning), policy-based methods (such as policy gradients), and actor-critic approaches that combine aspects of both. The choice of algorithm depends on factors including the complexity of the state space, whether the environment is fully observable, and computational constraints. Unlike traditional AI techniques, RL offers adaptability to unexpected situations and can discover novel strategies beyond designer expectations. However, these advantages come with challenges including high sample complexity (requiring many environment interactions), difficulty in specifying appropriate reward functions, and potential convergence to suboptimal solutions when trained in limited scenarios.

#### 2.2.1.1 The Bellman equation

The Bellman equation provides a mathematical foundation for solving Markov Decision Processes (MDPs) by establishing recursive relationships between value functions of different states. It serves as the cornerstone of many RL algorithms. The Bellman equation defines the optimal value function $V^*(s)$ for each state $s$ as:

$$V^*(s) = \max_a \left[ R(s,a) + \gamma \sum_{s'} P(s'|s,a) V^*(s') \right]$$

This recursive formulation elegantly captures the principle that the value of a state equals the reward obtained from the best action plus the discounted expected value of the successor state. For policies, the Bellman expectation equation becomes:

$$V^\pi(s) = \sum_a \pi(a|s) \left[ R(s,a) + \gamma \sum_{s'} P(s'|s,a) V^\pi(s') \right]$$

Similarly, the action-value function $Q^*(s,a)$ represents the expected return starting from state $s$, taking action $a$, and thereafter following the optimal policy:

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q^*(s',a')$$

Many RL algorithms iteratively apply these equations to converge to optimal solutions. Value iteration repeatedly updates state values using the Bellman optimality equation until convergence. Policy iteration alternates between policy evaluation (calculating values for the current policy) and policy improvement (selecting better actions based on updated values). While solving the Bellman equation exactly requires complete knowledge of the environment's dynamics (transition probabilities and rewards), most practical RL algorithms like Q-learning approximate solutions through sampling and iterative updates, enabling agents to learn optimal policies through experience rather than explicit environment models.

### 2.2.2 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) combines classical RL with deep neural networks to handle high-dimensional state spaces that would be intractable with traditional tabular methods.

This integration enables RL to operate effectively in complex environments with visual inputs, continuous action spaces, and intricate state representations common in modern video games. The breakthrough Deep Q-Network (DQN) algorithm, introduced by DeepMind in 2015, demonstrated superhuman performance on Atari games using only pixel inputs and game scores. DQN employs several key innovations including:

- Experience replay, which stores and randomly samples past experiences to break correlation between sequential samples
- Target networks that stabilize training by reducing moving target problems
- Convolutional neural networks that process visual information effectively

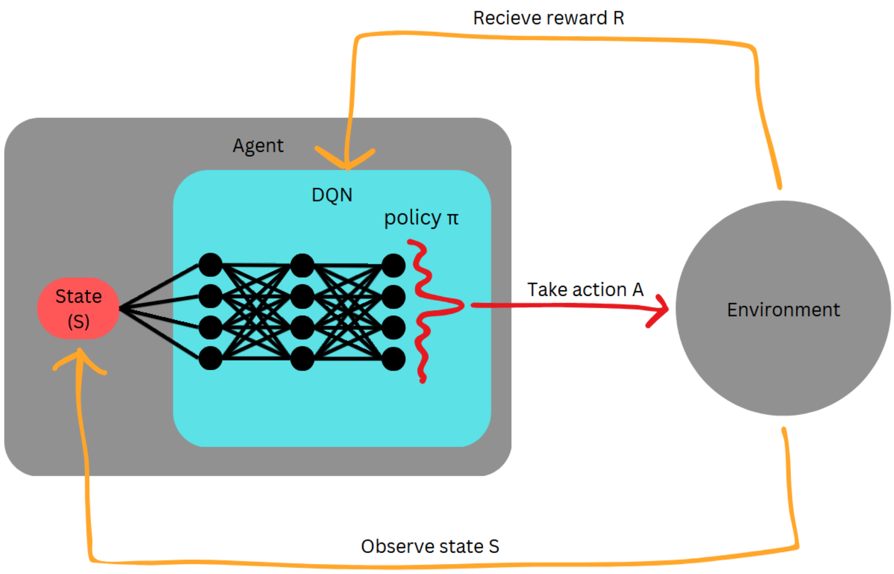


Figure 2.5: Architecture of a Deep Q-Network

# Chapter 3

# Problem Analysis and Requirements

This chapter contains an analysis of the problem and breaks it down into sub problems. It then sets out the functional and non-functional requirements.

## 3.1 Problem Analysis

The problem can be broken down into three main components:

### 3.1.1 Video Game Development

Creating a basic video game involves designing core gameplay mechanics, implementing player controls, creating a game environment, and establishing basic game rules and objectives. This requires selecting an appropriate game engine that supports the planned features and performance requirements. Core gameplay mechanics and rules must be defined to provide an engaging player experience, while visual assets and user interface elements need implementation to create a cohesive visual identity. Player control systems need to be intuitive and responsive, allowing for meaningful interaction with the game world. Finally, clear win/lose conditions need establishment to provide players with goals and motivation.

### 3.1.2 Network Communication Interface

Connecting an external AI agent to the game requires establishing a reliable communication channel. This involves designing a networking protocol that efficiently transmits necessary game state information and receives AI decisions. A server-client architecture must be implemented to facilitate this exchange, along with data serialization and parsing mechanisms to ensure proper interpretation of messages on both ends. Synchronization between game state and AI agent is critical to maintain consistency, while also managing network latency and potential disconnections to ensure smooth gameplay even under non-ideal network conditions.

### 3.1.3 AI Integration into Gameplay

Incorporating the AI agent as part of the gameplay loop involves defining the AI's role within the game context, whether as an opponent, ally, or environmental element. Interfaces for AI action and observation must be created to allow the AI to both perceive the game state and affect it meaningfully. Game state representation needs careful implementation to provide the AI with relevant information in a consumable format. The AI's capabilities must be balanced with player experience to ensure engaging but fair gameplay. Additionally, feedback mechanisms must be designed to showcase AI behavior, allowing players to understand and respond to AI actions appropriately.

## 3.2 Functional Requirements

| **ID** | **Requirement** |
|---|---|
| FR1 | The system must implement a playable video game with clear objectives and mechanics |
| FR2 | The game must provide intuitive player controls and interface |
| FR3 | The system must implement a client-server network architecture for AI communication |
| FR4 | The system must be able to transmit game state information to an external AI agent |
| FR5 | The system must be able to receive and process action commands from an external AI agent |
| FR6 | The game must incorporate AI actions into the gameplay loop in real-time |
| FR7 | The system must provide a complete representation of relevant game state to the AI |
| FR8 | The game must provide visual feedback of AI actions to the human player |
| FR9 | The system must handle connection initialization and termination with the AI agent |
| FR10 | The game must implement win/lose conditions that account for AI participation |

Table 3.1: Functional Requirements

## 3.3 Non-functional Requirements

| ID | Requirement |
|---|---|
| NFR1 | The game must maintain a minimum framerate of 30 FPS during gameplay |
| NFR2 | Network communication latency should not exceed 100ms under normal conditions |
| NFR3 | The system must recover gracefully from network disconnections |
| NFR4 | The game should be accessible to users with minimal gaming experience |
| NFR5 | The communication protocol must be well-documented for AI integration |
| NFR6 | The game's codebase must be maintainable and well-structured |
| NFR7 | The system should provide logging capabilities for debugging AI interactions |
| NFR8 | The game must run on standard desktop hardware without specialized requirements |

Table 3.2: Non-functional Requirements

# Chapter 4

# Design Decisions and Implementation

Over the course of the development of my project, I had to make numerous design decisions for each part. This chapter lays out and explains the design decisions I made, along with justifications for them, and details the implementation process that followed these decisions.

## 4.1 Version Control System

Throughout the development of the project, I will use the industry-standard Git version control system. This allows me to keep snapshots of the project over time, so I can revert if anything goes wrong, as well as keeping the project backed up to GitHub.

## 4.2 Choosing Technologies

### 4.2.1 Choosing a Game Engine

To develop the game frontend, I needed to choose a game engine or game development framework. Throughout my research, I considered three different options, Godot, Unity and Pygame. I selected Godot over Unity and Pygame after careful consideration of my project's requirements. While Unity offers powerful features, its proprietary nature, heavyweight installation, and steep learning curve made it less suitable for my academic project. Unity's reliance on the C Sharp programming language would also have required additional time investment to master. Pygame, despite its Python compatibility that would have allowed native interfacing with

PyTorch, lacks a graphical development environment and many built-in features that would speed up development. In contrast, Godot offered the perfect balance: it's open-source and free, extremely lightweight (only 130MB), uses a Python-like language (GDScript) that was quick to learn, features an intuitive interface I was already familiar with, and provides excellent 2D capabilities that matched my game requirements. Additionally, Godot's plain text resources integrate seamlessly with Git, its cross-platform compatibility ensures the game works on both Windows and Linux environments (critical for working between home and university), and its networking features provide essential connectivity to the backend model.

### 4.2.2 Choosing an ML Framework

For the model backend, there were only two real options - PyTorch and Tensorflow. I chose PyTorch over TensorFlow for my machine learning framework due to several key advantages. While TensorFlow offers a robust ecosystem and production-ready deployment tools, PyTorch's more intuitive API and dynamic computation graph better suited the experimental nature of this project. TensorFlow's static graph design, though efficient for production, would have hindered the rapid prototyping and debugging I needed during development. PyTorch's Python-first approach also aligned better with my existing workflow, offering seamless integration with Python data science libraries like NumPy and Pandas. The framework's extensive documentation, community support, and built-in tools for neural network development made implementation more straightforward. Additionally, PyTorch's native support for GPU acceleration provided the necessary computational power for model training, and its reinforcement learning libraries offered ready-made solutions for developing game AI agents. TensorFlow's steep learning curve and more verbose syntax would have increased development time without providing significant benefits for this particular academic project.

## 4.3 Design of the Game

For the game component of my project, I decided to develop a 2D tile-based platformer, similar to Super Mario Bros. This decision was driven by several practical considerations. First, 2D platformers offer a good balance between implementation complexity and gameplay depth, making them ideal for academic projects with limited timeframes. The tile-based approach simplifies level design, collision detection, and environmental interactions while still allowing for creative gameplay mechanics. The tile-based approach is also ideal for RL, as the environment is formed of tiles that can be classified by type, numbered, and then fed directly into a model.

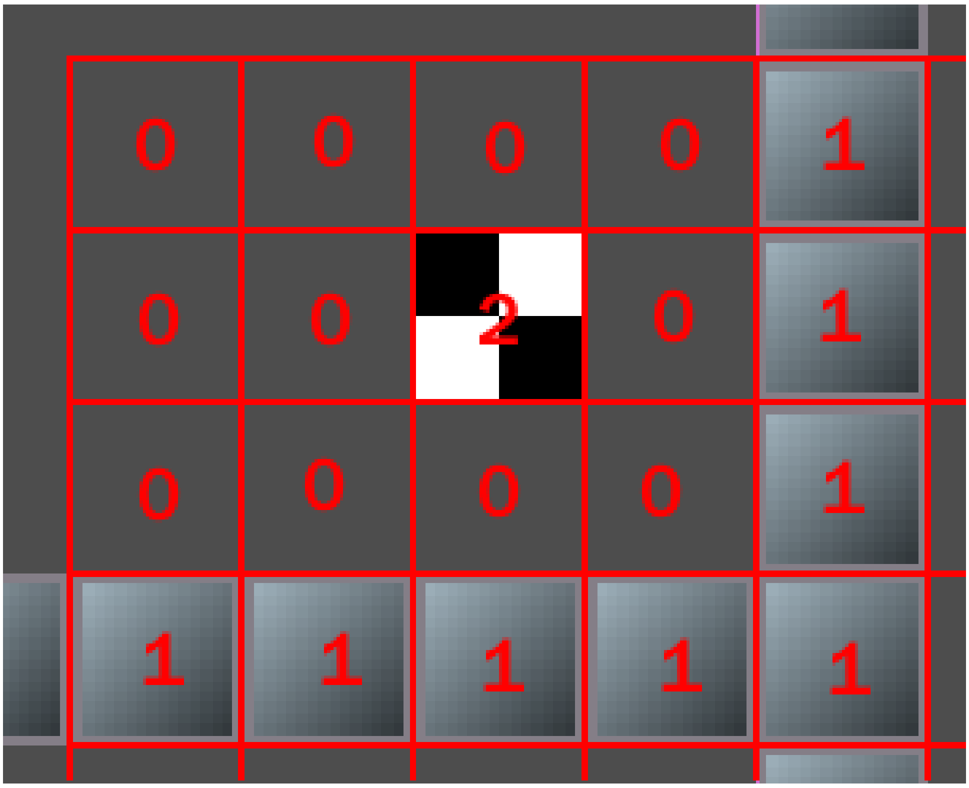


Figure 4.1: Classifying tiles by type

From a technical perspective, 2D platformers require fewer computational resources than 3D games, enabling me to focus more processing power on the machine learning components rather than graphics rendering. Godot's excellent 2D capabilities made this genre particularly suitable, as the engine provides robust tools for sprite animation, tilemap management, and physics-based movement. Additionally, platformers offer clear, discrete states and actions that translate well to reinforcement learning problems. The character can move left, right, jump, or perform other specific actions, while the game state can be represented efficiently for the ML model to process. This clarity in the action space makes platformers an excellent testbed for AI development, as the model must learn fundamental gaming concepts like obstacle avoidance, timing, and spatial navigation. The tile-based structure also facilitates procedural level generation, allowing me to create varied environments for training the AI without manually designing each level. This was particularly important for developing robust models that could generalize across different scenarios rather than overfitting to specific level layouts.

I decided to design the game around a clear, measurable objective: completing a time-based obstacle course. This design choice was deliberate as it provides several advantages for both gameplay and AI development. From a gameplay perspective, a timed obstacle course offers an intuitive goal that requires no elaborate explanation - players simply need to reach the end as quickly as possible. This straightforward objective creates natural replay value as players (human or AI) attempt to optimize their route and execution to achieve faster completion times. From a machine learning standpoint, this objective is ideal because it creates a well-defined reward function where success can be quantified precisely through time measurements. The faster an agent completes the course, the better its performance, providing a clear optimization target for reinforcement learning algorithms. This design also allows for incremental learning progression. An AI agent can first learn the fundamental task of reaching the goal, then gradually optimize for speed - mirroring how human players typically approach such challenges. The continuous nature of the time metric (as opposed to binary success/failure) gives the learning algorithm more nuanced feedback about small improvements in performance. Additionally, timed courses naturally incorporate key platforming challenges like precise jumping, obstacle avoidance, and route optimization, creating a rich environment for AI learning while remaining accessible to human players for comparison.

## 4.4 Spriting and Graphics

A small part of this project involves the creation of graphics, so we can see what is going on. In 2D games these are referred to as sprites. For the graphical elements of the game, I opted to create all assets using Aseprite, a specialized pixel art editor. This decision aligned perfectly with both the technical requirements of the project and the aesthetic direction I wanted to pursue. I deliberately chose a pixel art style for several reasons. First, the simplicity of pixel graphics reduced development time, allowing me to focus more on the AI components that were the primary focus of my research. Second, pixel art's grid-based structure naturally complemented the tile-based nature of the platformer, creating visual coherence between the mechanics and aesthetics. Finally, the minimalist style ensured that game elements remained visually distinct and easily recognisable. Using Aseprite, I created character sprites for the player character and AI character, as well as tiles for terrain and the finish line. I maintained a consistent color palette throughout to ensure visual harmony, limiting myself to a small set of colors that clearly differentiated between game elements while providing adequate contrast. For creating the user interface I used two fonts: PressStart2P, and the default Godot font provided with the engine. Both are open source and free to use.

## 4.5 Building a Basic Game Environment

My first task was to build a basic game environment within Godot that could then be built further upon. This consists of:

- **Player Character**: I implemented a character with basic movement controls including running left/right, jumping, and falling with appropriate physics, using Godot's built in 2D physics system.
- **Camera**: The game's camera is a Camera2D node that remains static providing a view of the whole map.
- **Tile Based Map**: I created a tilemap system using Godot's TileMapLayer node, allowing me to construct levels from reusable tiles representing walls, floors and ceilings.
- **Kill Plane**: At the bottom of the map, there is a player collidable box that will reset their position if it is touched. This ensures that the player cannot fall out of the map endlessly.
- **Finish Line and Timer**: Each map contains a finish line tile. As soon as the player is instantiated, their timer starts and will stop once they touch the finish line.

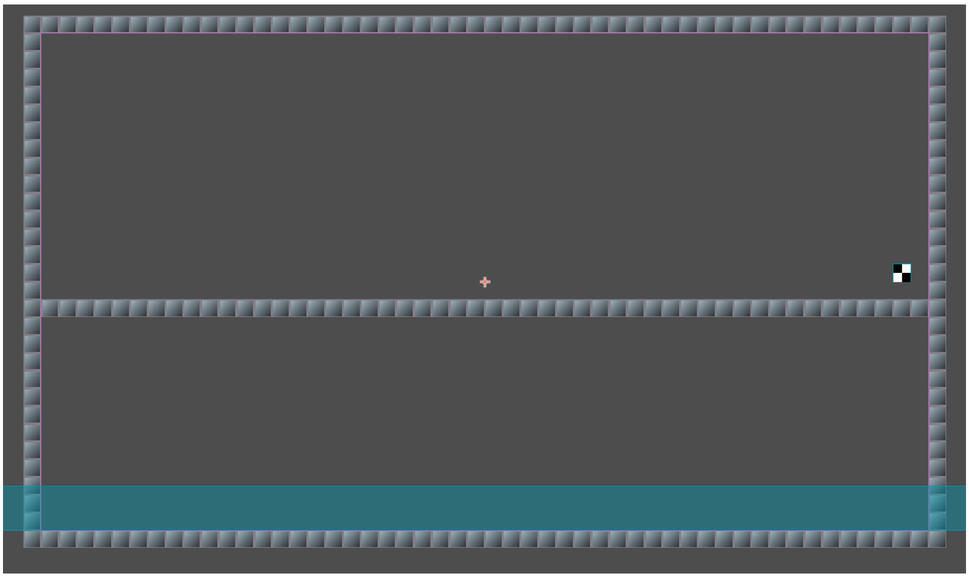

Figure 4.2: Basic Godot game environment

Godot's node based architecture allowed me to neatly lay these all out.

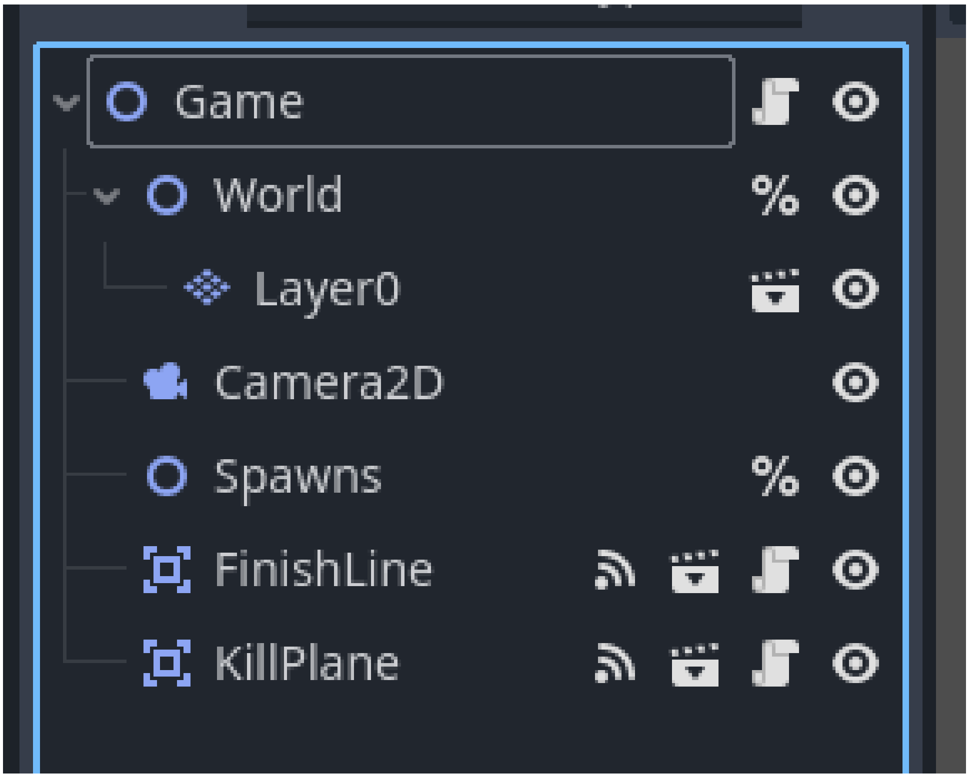


Figure 4.3: Environment's node hierarchy

## 4.6 State Observation System

For the reinforcement learning agent to make informed decisions, it needs to perceive its environment effectively. This required designing an observation system that captures relevant environmental information while maintaining computational efficiency.

### 4.6.1 Observation Scope and Representation

While capturing the entire environment state would provide the agent with complete information, such an approach proved prohibitively expensive from a computational perspective. A full environment representation would:

- Significantly increase the input dimensionality of the neural network
- Require substantially more training time to converge
- Demand greater memory resources during both training and inference
- Scale poorly with larger environment sizes

After experimentation with various observation window sizes, I determined that a 7×7 grid centered on the player offered an optimal balance between environmental awareness and computational efficiency. This window size provides sufficient context about nearby obstacles, pathways, and goals while keeping the state representation compact and manageable. The constrained field of view also encourages the agent to develop more generalizable navigation strategies rather than memorizing specific map layouts, potentially improving transfer learning capabilities across different environments. The agent perceives a 7×7 grid centered on its position, providing sufficient contextual awareness of the immediate surroundings.

This observation window uses a simple numerical encoding scheme:

- **0**: Empty space / traversable areas
- **1**: Player character (Always at the centre of the grid)
- **2**: Finish line
- **3**: Walls / non-traversable terrain

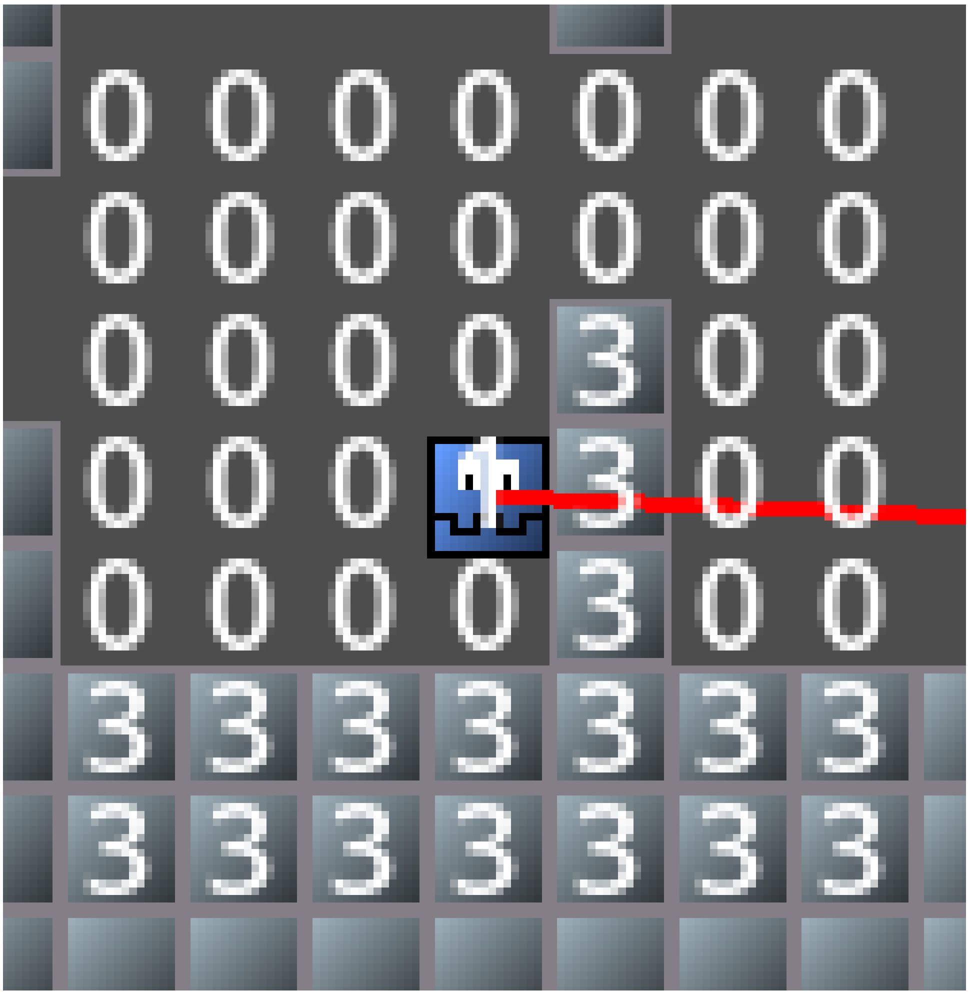


Figure 4.4: Visualisation of the observation window

### 4.6.2 Implementation Mechanics

The observation mechanism operates on each frame update, scanning the surrounding tiles relative to the agent's position. For each coordinate within the 7×7 grid, the system queries the tilemap to determine the tile type at that location. This information is then encoded according to the numerical scheme and assembled into a two-dimensional array. This process is run twice, once for the initial observation, and again for the second observation used alongside the reward.

This representation offers several advantages:

- **Translation invariance**: The agent learns based on relative positions rather than absolute map coordinates
- **Computational efficiency**: The fixed-size input simplifies network architecture and re-

duces processing requirements

- **Informational clarity**: The categorical encoding provides clear distinctions between critical environmental features

Once constructed, this state representation is transmitted to the reinforcement learning model, which uses it as input for determining the next optimal action.

## 4.7 Determining reward

Designing an effective reward function was critical for the reinforcement learning agent's success. After extensive experimentation, I implemented a straightforward distance-based reward mechanism that provides immediate feedback after each action:

- **+1 reward** when the agent moves closer to the finish line
- **-1 penalty** when the agent moves further from the finish line
- **+1 additional reward** upon reaching the goal

This reward function is run once each frame, alongside the second observation, after the model has made an action.

This simple approach encourages the agent to make continuous progress toward the objective while maintaining computational efficiency. The reward function balances immediate guidance with the freedom to discover optimal paths independently. During development, I explored several alternative reward structures, including proportional multipliers ($\times 1.2/\times 0.8$) and larger magnitude rewards (+600 for goal completion, -300 for falling). However, these higher values led to training instability, causing the model to struggle with convergence. The proportional multipliers proved beneficial and were retained, while the reward magnitudes were scaled back to $\pm 1$ to ensure stable learning. I also experimented with line-of-sight rewards using raycasting techniques, where the agent would receive additional rewards when establishing visual contact with the goal. After testing, this approach was ultimately not incorporated into the final implementation as it did not significantly improve performance while adding computational overhead.

A significant limitation of this reward structure is that it discourages exploratory backtracking, effectively requiring level designs to maintain a linear progression toward the goal. This constraint arises because the agent receives consistent penalties when moving away from the finish line, even when such movements might be strategically necessary to navigate complex environmental features. While this simplifies the learning process for straightforward courses, it potentially limits the agent's ability to discover optimal solutions in more complex, non-linear environments where temporary retreat may be advantageous.

## 4.8 Implementing Networking

To facilitate communication between the Godot game environment and the Python-based reinforcement learning model, I developed a robust networking system using TCP sockets and JSON encoding. This approach allows the game state to be effectively transmitted to the model for processing, and action decisions to be received back for execution within the game environment.

### 4.8.1 Communication Protocol Design

After evaluating several options for inter-process communication, I selected a TCP socket-based approach with JSON encoding for several key reasons:

- **Reliability**: TCP's connection-oriented nature ensures that no state observations or action commands are lost during transmission
- **Flexibility**: JSON encoding provides a structured yet adaptable format for representing complex game state information
- **Cross-platform compatibility**: The socket-based approach works seamlessly across different operating systems
- **Low implementation overhead**: Both Godot and Python offer built-in libraries for TCP communication without requiring additional dependencies

To implement this, I used the *socket* library on the python side, and a *StreamPeerTCP* node on the Godot side.

### 4.8.2 Operational Modes

The Python component of the system was designed with two distinct operational modes:

- **Training Mode**: Incorporates epsilon-greedy exploration strategies and periodically saves model checkpoints to preserve training progress. This mode focuses on model improvement through experience gathering and optimization.
- **Play Mode**: Loads the latest saved checkpoint and plays deterministically with epsilon set to zero, demonstrating the current capabilities of the trained model without further exploration.

### 4.8.3 Communication Cycle Implementation

The core networking loop on the Godot side follows a structured pattern of state transmission and action reception:

1. First, the game engine calculates and sends the reward from the previous action along with the current state observation
2. It then awaits confirmation that the Python script is ready for the next cycle
3. Upon confirmation, the game sends a fresh observation of the current environment state
4. Finally, it receives the model's selected action (encoded as an integer: 0 for left movement, 1 for right movement, 2 for jump) and executes this action within the game environment

This communication cycle repeats once each frame during gameplay, maintaining a consistent feedback loop between the game environment and the learning model. The implementation ensures that state observations, rewards, and actions are synchronized properly, preventing timing issues that could otherwise lead to misaligned learning experiences.

The code excerpt below illustrates the essential communication pattern:

```
# Transmit previous reward and current state
var model_reward : float = reward()
peer.put_float(model_reward)
peer.poll()
var next_state : Dictionary = observe()
var next_state_encoded : PackedByteArray = JSON.stringify(next_state).to_utf8_buffer()
peer.put_data(next_state_encoded)
peer.poll()

# Await ready signal from Python
peer.poll()
```

```
var _isready : String = peer.get_string(5)

# Reset reward flags
deathPenalty = false
winReward = false

# Send observation for decision making
var visibleTiles : Dictionary = observe()
var visibleTilesEncoded : PackedByteArray = JSON.stringify(visibleTiles).to_utf8_buffer()
peer.put_data(visibleTilesEncoded)
peer.poll()

# Receive and execute model's action decision
peer.poll()
var model_action : int = peer.get_u8()
action(delta, model_action)
```

This networking implementation provides the critical infrastructure that allows the reinforcement learning agent to interact with the game environment, facilitating the entire learning process.

## 4.9 Implementing the Model

For the reinforcement learning component, I implemented a Deep Q-Network (DQN) architecture with convolutional layers to process the spatial information from the game environment.

### 4.9.1 Model Architecture

After considering various neural network architectures, I selected a convolutional DQN model that could efficiently process the 2D grid observation data from the game environment:

- **Input Layer**: Accepts the 7×7 grid observation (49 values) reshaped into a 2D format
- **Convolutional Layer**: A 2D convolutional layer with 16 filters, kernel size of 3×3, stride of 1, and padding of 1, enabling the network to detect spatial patterns in the environment
- **Hidden Layers**: Two fully-connected layers with 128 neurons each and ReLU activation functions
- **Output Layer**: A fully-connected layer with 4 outputs corresponding to the possible actions (move left, move right, jump, do nothing)

The convolutional layer was particularly important for this implementation as it allows the model to recognize patterns in the spatial arrangement of walls, empty spaces, and the goal within the observation window. This spatial awareness is critical for navigating the platformer environment effectively.

### 4.9.2 Training Algorithm

I implemented a standard Q-learning algorithm with experience replay and a target network for stable training:

- **Epsilon-greedy Exploration**: Starting with a high exploration rate that gradually decreased over time, balancing exploration of new strategies with exploitation of learned knowledge
- **Target Network**: A separate network with parameters periodically updated from the main network to provide stable Q-value targets during training
- **Optimizer**: Adam optimizer with a learning rate of 0.001, selected for its adaptive learning rate capabilities
- **Loss Function**: Mean Squared Error between predicted Q-values and target Q-values

### 4.9.3 Training Process

The training process operated on a cycle-by-cycle basis with the game environment:

1. Receive the current state observation from the Godot environment
2. Select an action using the epsilon-greedy policy
3. Send the selected action to the game environment for execution
4. Receive the reward and next state observation
5. Update the Q-network using the Bellman equation
6. Periodically update the target network parameters

During training, the epsilon value (controlling exploration vs. exploitation) decayed from an initial value of 0.9 to a minimum of 0.1, allowing the agent to gradually transition from predominantly random exploration to informed decision-making based on learned values.

Chapter 5

# Implementation

# Chapter 6

# Evaluation

# Chapter 7

# Ethics

# Bibliography


Carpenter, S. (2019), 'Simulacrum: Building believable behaviors in video game ai', https://medium.com/@selcarpe/https-medium-com-simulacrum-28477a0e759e. Accessed: 26/04/25.

Spiceworks (2021), 'What is finite state machine (fsm)?', https://www.spiceworks.com/tech/tech-general/articles/what-is-fsm/. Accessed: 26/04/25.

Thompson, T. (2020), 'Building the ai of f.e.a.r. with goal-oriented action planning', https://www.gamedeveloper.com/design/building-the-ai-of-f-e-a-r-with-goal-oriented-action-planning. Accessed: 26/04/25.

Wikipedia (2025), 'Artificial intelligence in video games', https://en.wikipedia.org/wiki/Artificial_intelligence_in_video_games. Accessed: 26/04/25.