A 2D-Gaming Benchmark for Deep Reinforcement Learning

by

Alex Lowe

A thesis submitted in partial fulfillment
of the requirements for the degree of

Bachelor's of Science

in

Computer Science

Ontario Tech University

Supervisor: Cristiano Politowski and Ali Neshati

April 2026

Abstract

This thesis evaluates the feasibility of deep reinforcement learning (DRL) as a playtesting tool for small 2D games, focusing on game balance and the question of whether an indie developer can run this workflow on a consumer machine. We constructed a benchmark of three games (Flappy Bird, Pong, and Snake), each implemented from scratch in Pygame and Gymnasium, and trained four DRL algorithms (PPO, A2C, TRPO, and RPPO) on each game for 10 million timesteps on a AMD Ryzen 5 8600G without a discrete GPU. Each trained agent was evaluated across three difficulty levels designed around a doubling-and-halving principle, with a human player and a random agent as baselines.

DRL-based playtesting proved not readily feasible for small studios without significant upfront investment. The hardware barrier was low, but reward function design was the dominant cost, requiring extensive iteration in all three games. TRPO was the most consistent performer, ranking first or joint-first across all three games and being the only algorithm to clear any pipes on Flappy Bird Hard. PPO was the most reliable, never collapsing on any game or difficulty. A2C offered the best trade-off between training speed and performance on standard difficulties, while RPPO was the weakest algorithm in every game despite the longest training times; its LSTM architecture provided no benefit in fully observable environments. Generalisation to easier difficulties was reliable across all games. Hard difficulty was more discrim-

inating: Flappy Bird and Pong showed algorithmic collapse while Snake remained stable across all three levels. These results demonstrate that a trained agent evaluated across difficulty variants can serve as a lightweight game balance measurement tool, but the reward engineering required to produce a competent agent represents a substantial barrier for indie adoption.

Acknowledgements

I want to thank everyone who had a part in shaping this thesis, from the initial idea of making an AI play games into a fully developed research project spanning multiple games, algorithms, and difficulty levels. This has been one of the most rewarding things I have worked on in my academic career, and I am glad I had the opportunity to pursue something I found genuinely interesting while also producing something that could be useful to other developers.

I would like to specifically thank Cristiano Politowski for going far beyond what was required of a supervisor. He gave time outside of office hours to help with implementation problems, and held practice presentations while out of the country to make sure his students were prepared. His commitment to seeing every student put their best foot forward made a real difference to this project. I would also like to thank Ali Neshati for his role as co-supervisor throughout the project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Game testing is one of the most time-demanding phases of game development. For large studios with dedicated quality assurance teams, automated testing is commonly used [9, 20]. For smaller studios and independent developers, however, the resources required for automated testing make it less feasible [20]. Games ship with undetected bugs and unbalanced difficulty because there was not enough time during development to find and fix them [20].

Deep reinforcement learning (DRL) has demonstrated that agents can play games at a human level and beyond [17, 27]. Rather than using these agents only as benchmarks for algorithm development, there is a growing body of work that applies DRL agents as automated playtesters that explore the game, find bugs, and measure difficulty in ways that complement human testing [8, 9]. Existing solutions, however, are not accessible to the average indie developer. Engine-integrated toolkits such as Godot RL Agents [2], Unity-based wrappers [6, 11], and Unreal integrations [25] require full engine setups that add substantial overhead for small 2D projects. Academic benchmarks such as Procgen [7] and TeachMyAgent [23] focus on evaluating RL algorithms rather than supporting a developer's iteration loop. Industry DRL testing

pipelines exist but are closed and undocumented outside of case-study reports [9]. The simpler 2D games that characterize the independent game market have received comparatively little attention from tools that are actually usable by small teams. It is also worth noting that this thesis addresses a specific subset of the broader playtesting problem: we focus on *game balance* (whether the game is appropriately difficult and whether trained agents generalise across difficulty levels), not on bug detection or the discovery of exploits. These are complementary concerns, but they require different evaluation methodologies.

This thesis asks whether DRL-based playtesting is *feasible* for the kind of small 2D games that an indie developer might build. We define the following research questions:

- **RQ1 (Feasibility):** How feasible is it for small studios (e.g., indie developers) to implement DRL agents in 2D games in terms of time, resources, and technical complexity?
- **RQ2 (Performance):** How do DRL agents (PPO, A2C, TRPO, RPPO) perform across different 2D games with varying mechanics and complexity?
- **RQ3 (Generalizability):** To what extent can trained DRL agents generalize on modified versions of the same stages within the same game?
- **RQ4 (Developer Perspective):** What challenges, trade-offs, and benefits emerge when smaller developers attempt to adopt DRL-based testing?

To answer these questions, we constructed a small benchmark consisting of three 2D games of increasing implementation complexity: clones of Flappy Bird, Pong, and Snake, each implemented from the ground up using the Pygame framework[1].

[1] `https://www.pygame.org`

We trained playtesting agents using four DRL algorithms (PPO, A2C, TRPO, and RPPO) on each game for 10 million timesteps on an AMD Ryzen 5 8600G without a discrete GPU, then evaluated each trained agent across three difficulty levels (easy, normal, and hard) designed around a doubling-and-halving principle [5].

We found that TRPO was the most consistent performer across all three games, ranking first or joint-first in every game and being the only algorithm to clear any pipes on Flappy Hard. PPO was the most reliable choice overall, never collapsing on any game or difficulty. A2C offered the best trade-off between training speed (30 to 47 minutes per game) and performance on Easy and Normal difficulties, but broke down on Pong Hard. RPPO was the weakest algorithm in every game despite the longest training times, a result consistent with its LSTM memory providing no benefit in fully observable environments. Generalisation was strong to Easy but variable on Hard: Flappy Bird lost three of four algorithms on Hard, Pong lost only A2C on Hard, and Snake remained stable across all three levels. The reward function was the dominant source of development effort across all three games, with Snake requiring the most iterations by a wide margin.

The main contributions of this thesis are:

- A lightweight 2D game benchmark built in Pygame, covering three games with adjustable *game difficulty* configurations, and training parameters like *observation spaces* and *reward functions*.
- A comparative evaluation of four DRL algorithms across three games and three difficulty levels, providing a reference for developers choosing an algorithm for their own use case.
- An analysis of how well agents trained at the default difficulty generalize to easier and harder variants, characterizing the limits of DRL policy transfer in

these settings.

- A developer-perspective discussion of the implementation process, documenting the challenges and iteration cycles involved in building each environment.

The thesis has the following structure. Chapter 2 provides background on the DRL algorithms used and a description of each game in the benchmark. Chapter 3 reviews related work on DRL benchmarks, engine-integrated toolkits, and DRL-based game testing. Chapter 4 describes our research questions, experimental methodology, and the design of each game environment. Chapter 5 presents the results of training and evaluation for each game and answers the four research questions. Chapter 6 summarises the findings, discusses limitations, and outlines directions for future work.

Chapter 2

Background

2.1 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) is a branch of machine learning in which an agent learns to make decisions by interacting with an environment. At each timestep, the agent observes a state, selects an action, and receives a reward signal that indicates how well it performed. Over many interactions, the agent learns a policy, a mapping from states to actions, that maximizes its cumulative reward.

Steps/Timesteps refer to a single action taken by the agent within an episode. **Episodes** represent a full playthrough of a game from the initial state to a terminal condition, such as the player dying or reaching the goal. The **reward function** is how we communicate to the agent what constitutes good and bad behaviour: positive rewards are given for desirable outcomes, such as progressing toward the goal, while negative rewards are applied for undesirable ones, such as dying prematurely.

2.1.1 PPO

Proximal Policy Optimization (PPO) [27] is one of the most widely used DRL algorithms due to its stability and sample efficiency. PPO improves upon earlier policy gradient methods by introducing a clipped surrogate objective that prevents excessively large policy updates. By constraining how much the policy can change in a single update, PPO strikes a balance between exploration and stable learning, making it a reliable choice for a wide range of environments.

2.1.2 TRPO

Trust Region Policy Optimization (TRPO) [26] takes a more principled approach to stable training by enforcing a hard constraint on the KL divergence between the old and new policy at each update step. This trust region constraint guarantees that the policy does not change too dramatically in any single step, at the cost of higher computational overhead compared to PPO. TRPO tends to be more conservative in its updates but can achieve strong performance in environments with well-defined structure.

2.1.3 RPPO

Recurrent Proximal Policy Optimization (RPPO) extends PPO with a Long Short-Term Memory (LSTM) network in the policy, enabling the agent to maintain a memory of past observations. This follows the recurrent-DRL pattern first established by DRQN for Q-learning [13] and later adapted to policy-gradient methods in RecurrentPPO [19]. The recurrent architecture is particularly useful in partially observable environments where a single frame of information is insufficient to make an informed decision. The recurrent architecture comes at a high cost in training time due to the

sequential nature of LSTM state computation. In this work, RPPO is implemented using the `RecurrentPPO` class from the `sb3-contrib` library.

2.1.4 A2C

Advantage Actor-Critic (A2C) [16] is a synchronous variant of the actor-critic family of algorithms. It maintains two networks: an *actor* that outputs a probability distribution over actions, and a *critic* that estimates the value of the current state. The advantage function, which measures how much better an action was compared to the average, is used to reduce the variance of the policy gradient estimate. A2C is typically the fastest algorithm to train owing to its simple update rule, though it can produce noisier learning curves than PPO or TRPO.

2.2 Games Used in the Experiments

2.2.1 Flappy Bird

Flappy Bird is an auto-scrolling side-scrolling game that presents the player with a single binary action: flap the bird's wings or do nothing. The screen scrolls continuously toward a series of pipes, and the player's goal is to guide the bird through the gap in each pipe without colliding. The game's difficulty lies in the precision required to navigate the gaps, particularly when the bird's momentum must be managed carefully between flaps.

2.2.2 Pong

Pong is one of the earliest arcade games, consisting of three elements: the player's paddle, the ball, and an opponent paddle controlled by a CPU. The goal is to be

the first to reach a score of 10 points by hitting the ball past the opponent's paddle. Each rally begins with the ball launched in a random direction, and the ball accelerates slightly with each paddle contact, increasing the reaction demands as the rally continues.

2.2.3 Snake

Snake is a classic arcade game in which the player controls a continuously moving snake on a fixed grid. The goal is to collect apples that appear at random positions on the grid; each apple collected causes the snake to grow by one segment. The game ends if the snake collides with a wall or with its own body. As the snake grows longer, navigating the grid without self-collision becomes progressively harder, making Snake a long-horizon planning challenge compared to the reactive control required by Flappy Bird and the reactive tracking required by Pong.

Chapter 3

Related Work

Prior work on DRL and game testing falls into three categories: algorithm benchmarks that use standardised game environments, engine-bound integrations that bring RL training into production game engines, and automated playtesting studies that deploy trained agents to assess game balance. Each line of work is valuable but does not directly address whether a small developer can use DRL for game balance testing on their own 2D game on a consumer machine, or whether a policy trained at one difficulty level generalises to harder or easier variants. Table 3.1 summarises the most relevant works.

Table 3.1: Related works and their main limitation relative to the focus of this thesis.

Work	Key limitation relative to this thesis
[15]	Engine-bound FPS benchmark; Doom-specific and not portable to a Python-only stack
[14]	Platformer benchmark focused on algorithm evaluation; no developer-cost or feasibility analysis
[18]	ROM-based generalisation study; requires commercial ROMs, not applicable to a developer's own game
[7]	Procedural generalisation benchmark; not aimed at game-balance assessment for a specific authored title
[4]	Standard RL interface; does not address indie-developer workflow or game-balance assessment
[2]	Engine-bound; requires the Godot editor and is not applicable without it
[25]	Unreal-dependent; impractical for small Python-based 2D projects
[12]	Offline-only dataset benchmark; not a playtesting or balance assessment setup
[9]	AAA-scale case study; does not address indie-developer feasibility or cross-difficulty generalisation
[8]	3D curiosity-driven testing at industrial scale; different genre and scope
[23]	Curriculum-learning benchmark; not aimed at balance testing or indie developer feasibility
[6]	Unity-specific; engine-bound and not aimed at indie-scale balance assessment
[10, 11]	Unity-dependent; does not address indie feasibility, cross-difficulty generalisation, or developer cost

3.1 DRL Benchmarks in Games

The Arcade Learning Environment (ALE) [3] gave Atari 2600 titles a uniform RL interface and established games as standardised algorithm benchmarks. OpenAI Gym [4] extended that model by separating the `reset`/`step` interface contract from any specific game engine, and Gymnasium [29] continues to maintain this standard. Stable-Baselines3 [21] built reproducible implementations of standard algorithms on top of that interface. Together, these form the training infrastructure on which this thesis's pipeline was built.

Beyond infrastructure, game environments have served as DRL testbeds for decades. The Mario AI benchmark and competition [14, 28] established a platformer sub-genre as a useful testbed for generalisation and robustness. Gym Retro [18] wrapped commercial ROMs, including *Sonic the Hedgehog*, in a Gym-compatible interface to study generalisation across unseen levels. Procgen [7] introduced procedurally generated environments, including several side-scrolling games, to measure out-of-distribution robustness and sample efficiency. TeachMyAgent [23] studied curriculum learning in a 2D morphology-traversal environment.

These benchmarks share a defining characteristic: the environments are fixed or procedurally generated by the researchers, and the object of study is the algorithm's performance within them. None addressed the developer's perspective: whether the workflow is feasible for a solo developer with a consumer machine, how long each phase of the loop takes, or whether a trained policy's performance at one difficulty level provides useful balance information about other configurations. This thesis asked those questions directly, using three developer-authored games as the environment and difficulty variation as the primary evaluation axis.

3.2 Engine-Bound RL Benchmarks and Integrations

A parallel line of work embedded RL training directly inside production game engines. ViZDoom [15] exposed the *Doom* engine as an RL environment and demonstrated that engine-specific wrappers could support pixel-based DRL, but it remained tied to FPS mechanics and the Doom build chain. Godot RL Agents [2] provided a Gymnasium-compatible bridge to Godot with Stable-Baselines3 support, but requires the full Godot editor to use. URLT [25] wrapped Unreal Engine 4 with a modular API and Blueprint-based task design. AI4U [10, 11] added declarative rewards inside Unity. Deep RTS [1] demonstrated the value of a lightweight, genre-specific RL environment with accelerated simulation.

These integrations are valuable for teams already working inside the target engine, but the engine itself is a prerequisite: setup costs, build pipelines, and engine-specific tooling come with the package. This thesis used a Python-only stack: Pygame for game logic, Gymnasium for the RL interface, and Stable-Baselines3 for training. This stack runs on a standard consumer machine without additional software dependencies, and the workflow cost figures reported in Chapter 5 are directly replicable by any developer with a Python environment.

3.3 DRL-Based Automated Playtesting and Balance

Researchers have increasingly deployed DRL agents as automated playtesters rather than as benchmark participants. [22] used DQN-based loss profiles to detect bugs in fault-injected environments. [9] documented the practical challenges of deploy-

ing RL testers in AAA pipelines, including environment heterogeneity, build non-stationarity, and large action spaces. [8] combined intrinsic curiosity with multi-agent RL to improve exploration coverage in 3D environments. [6] trained behaviourally diverse agents in Unity to support persona-based testing. [24] used trained agents to assess balance, difficulty, and fairness in competitive game levels.

This line of work is most directly relevant to this thesis. The core premise, that a trained DRL agent can serve as an automated playthrough and produce balance-relevant measurements, is shared. The closest precedent in spirit is Rupp et al., who assessed balance via trained agents across competitive level configurations. The gap across the entire body of work is one of scale and accessibility: existing DRL playtesting studies ran inside production engines or large-scale infrastructure, and none included a developer-cost analysis, a cross-difficulty generalisation study, or a feasibility discussion for a solo indie developer. This thesis contributed that combination: a controlled benchmark across three games, four algorithms, and three difficulty levels, with full workflow-time measurements and a developer-perspective account of what the loop actually costs at indie scale.

Chapter 4

Approach

In this thesis, we aimed to evaluate the feasibility of deep reinforcement learning agents in 2D games, guided by the four research questions introduced in Chapter 1.

For RQ1, we tracked the hardware usage of training and evaluating each game, as well as the time required for both, including the time spent implementing each game and iterating on the DRL code. For RQ2, we evaluated the average score across 100 episodes for each algorithm on each game and each difficulty, then presented the results graphically to facilitate comparison. For RQ3, difficulty serves as our generalization axis: we set the hard difficulty at twice the default challenge and the easy difficulty at half, giving us a controlled look at how agents behave in similar but modified environments. For RQ4, we maintained a development journal throughout the implementation of each game, documenting major design decisions, iteration choices, and obstacles encountered.

To develop the games, we used the Pygame library[1]. We chose Pygame because it is lightweight, Python-native, and integrates cleanly with Gymnasium, allowing us to keep the full pipeline in a single language without the overhead of a production game

[1] `https://www.pygame.org`

engine. For the DRL code, that is, the code that connects each game to the DRL libraries, we used Stable-Baselines3 [21] and Gymnasium [29]. The main components in the DRL code are the *action space*, the *observation space*, and the *reward function*.

The **action space** defines the set of inputs available to the agent at each timestep, mirroring what a human player can do. For each game, we constructed the action space to give the agent full and meaningful control of the in-game character.

The **observation space** defines what the agent can see at each step. Deciding what information to provide to the agent is one of the most consequential design choices in DRL: providing too much information makes it difficult for the agent to identify what is relevant, while providing too little risks omitting information the agent needs to act correctly. Striking this balance required careful iteration for each game.

The **reward function** is how the agent learns what constitutes good and bad behaviour. Positive rewards are assigned to actions that move the agent toward the goal, such as dodging an obstacle or collecting an item, whereas negative rewards are assigned to actions that move the agent away from the goal or prematurely terminate the episode.

We created three difficulty levels for each game: Easy, Normal, and Hard. Hard is always twice as demanding as Normal, and Easy is half as demanding, following the game design philosophy of doubling and halving [5].

After each agent was fully implemented and able to play the game, training was conducted using Stable-Baselines3. All agents were trained for 10 million timesteps. During training, metrics were written to CSV files and logged to the terminal, allowing progress to be monitored in real time or reviewed after training was completed. These CSV files were subsequently used to generate training curves that visualize reward and game-specific performance metrics over time. Once training finished, further

modifications to the DRL code were made when necessary, with only one variable changed between iterations to isolate the effect of each change.

After no further obvious issues remained with the agent's observation space, action space, or reward function, we evaluated each agent over 100 episodes per difficulty to assess how it actually performs in the environment. This evaluation step is important because an agent can sometimes learn to exploit the reward function in ways that do not correspond to genuine task completion. Each evaluation episode ran for a maximum of 30 minutes of wall-clock time. If evaluation revealed remaining issues or improvements to the DRL code, the agent was adjusted and retrained. This process was repeated for every game.

Chapter 5

Result of the Experiments

Each game was evaluated using four DRL algorithms (A2C, PPO, RPPO, TRPO), a human player, and a random agent as baselines. All DRL agents were trained on the Normal difficulty for 10 million timesteps using 10 parallel environments on an AMD Ryzen 5 8600G, then evaluated over 100 episodes per difficulty. Random baseline data was collected by sampling uniformly from the action space for 100 episodes per difficulty. The three games are presented in order of increasing implementation complexity: Flappy Bird, then Pong, then Snake.

5.1 Experiment #1 - Flappy Bird

Flappy Bird is the simplest game in the benchmark in terms of control: the agent has a single binary action and just needs to guide the bird through pipe gaps. Despite this simplicity, an important failure mode emerged early in development. Agents that cleared pipes too close to the top or bottom edge would immediately crash on their next flap, as they had no vertical margin left to recover. The fix was a gap-centring bonus that pushed agents to aim for the middle of each gap rather than barely clearing

the edge.

5.1.1 What is Flappy Bird?

Flappy Bird is an auto-scrolling side-scrolling game with a single binary action: flap the bird's wings or do nothing. The screen scrolls continuously toward a series of pipes, each with a gap the bird must pass through. Flapping applies an upward impulse; gravity pulls the bird down between flaps. The game ends immediately when the bird collides with a pipe or the screen boundaries.

Flappy Bird was chosen for the benchmark because it represents the simplest form of reactive control: one action, five observations, and a clear success metric (pipes cleared). It serves as a baseline game to confirm that all four DRL algorithms can learn a viable policy before tackling more complex environments.

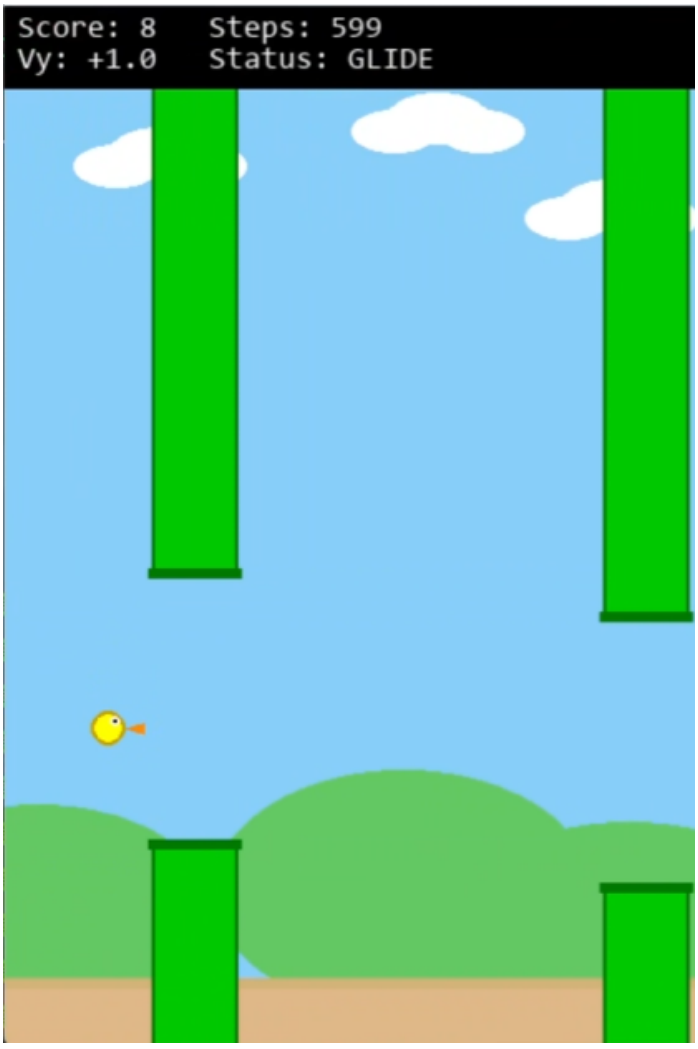

Figure 5.1: Screenshot of the Flappy Bird game environment.

5.1.2 Game Difficulties

Flappy Bird's difficulty is controlled entirely by the pipe gap width. The pipe speed, gravity, and flap strength are identical across all difficulty levels. We used the doubling and halving heuristic to create the game's difficulty: Hard is twice Normal and Easy is half Normal.

- On **Easy**, the pipe gap is `300` pixels (2× default);

- On **Normal**, the pipe gap is `150` pixels;
- On **Hard**, the pipe gap is `75` pixels (0.5× default).

The narrower gap in Hard makes precise vertical positioning substantially more demanding.

5.1.3 Experiment Setup

Action Space The agent has a single binary action: flap or do nothing. Flapping applies upward momentum; gravity pulls the bird down continuously between flaps.

Episode An episode ends when one of two terminal conditions is met: the bird collides with a pipe, or the bird moves outside the top or bottom screen boundary. Either event terminates the episode immediately and triggers the death penalty in the reward function.

Observation Space The agent receives five values: the bird's current y position, its current vertical velocity, the horizontal distance to the next pipe, and the y coordinates of the top and bottom of the upcoming gap.

Reward Function The Flappy Bird reward function went through moderate iteration relative to Snake. The initial version rewarded the agent for clearing pipes and penalised death, achieving approximately 100 pipes consistently on Normal difficulty. The key improvement was the gap-centring bonus: agents clearing near the gap edge lacked vertical margin to survive the next flap, so rewarding proximity to the gap centre substantially improved robustness.

Terminal events. A large reward is given for clearing a pipe; a fixed penalty applies on death.

Shaping terms. Three signals layer on top of the terminal events:

- **Survival bonus:** a small fixed reward each step the agent remains alive.
- **Gap-centring term:** scales the reward based on how close the bird is to the gap midpoint; negative when the bird is near the edge.
- **Score-based accumulating bonus:** a small additional reward proportional to the current pipe count, sustaining motivation at high scores.

```
if terminated:
    return -5.0

r = 0.05

if pipe_cleared:
    r += 3.0

gap_center = (gap_top + gap_bottom) / 2.0
centering = 1.0 - |bird_y - gap_center| / (gap_size / 3.0)
centering = max(centering, -0.05)
if centering > 0: centering *= 2.0
r += 0.5 * centering

r += score / 100.0

return r
```

5.1.4 Training the DRL Agents

All four algorithms were trained for 10 million timesteps using 10 parallel environments on an AMD Ryzen 5 8600G, with the same hyperparameter configuration across all games.

All four algorithms start near 0 and rise in three phases: a flat early phase while the bird crashes immediately, a rapid climb once the agent discovers gap-centring, and a plateau near 1.4 to 1.8. A2C climbs fastest in the first 2 million steps but is the noisiest throughout. TRPO has the smoothest climb and the most consistent plateau. PPO and RPPO oscillate heavily, with periods dropping back near 0 throughout training.

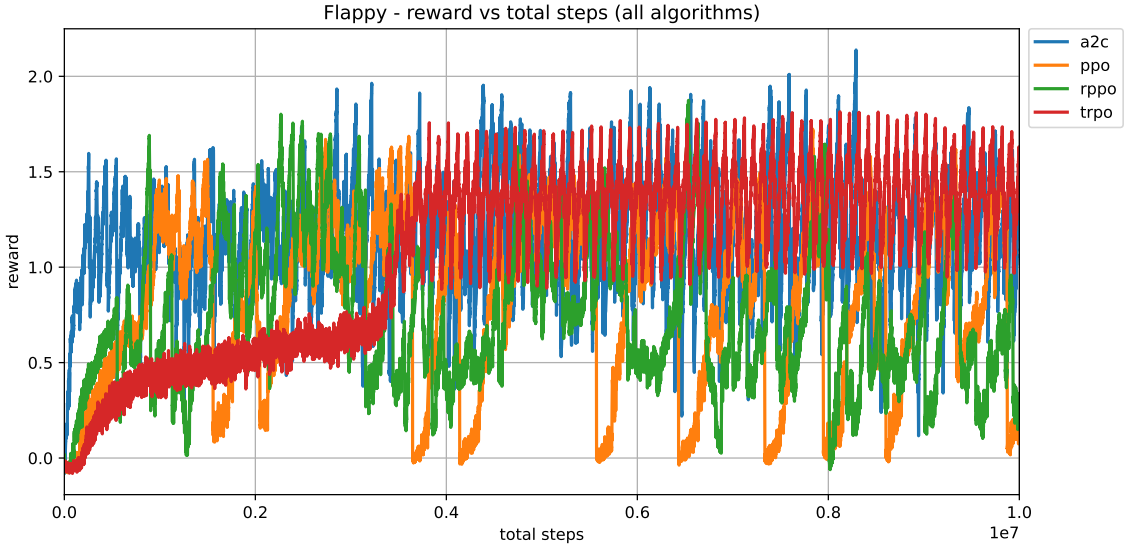

Figure 5.2: Flappy Bird: episode reward versus training steps for all four algorithms.

A2C, PPO, and TRPO all finish within the first 5,000 seconds. RPPO is the only line extending past that, continuing to 25,000 seconds, and does not reach a higher reward ceiling than the other three despite the extra time. RPPO's 7-hour training time is the most extreme gap between time invested and result achieved in the entire benchmark.

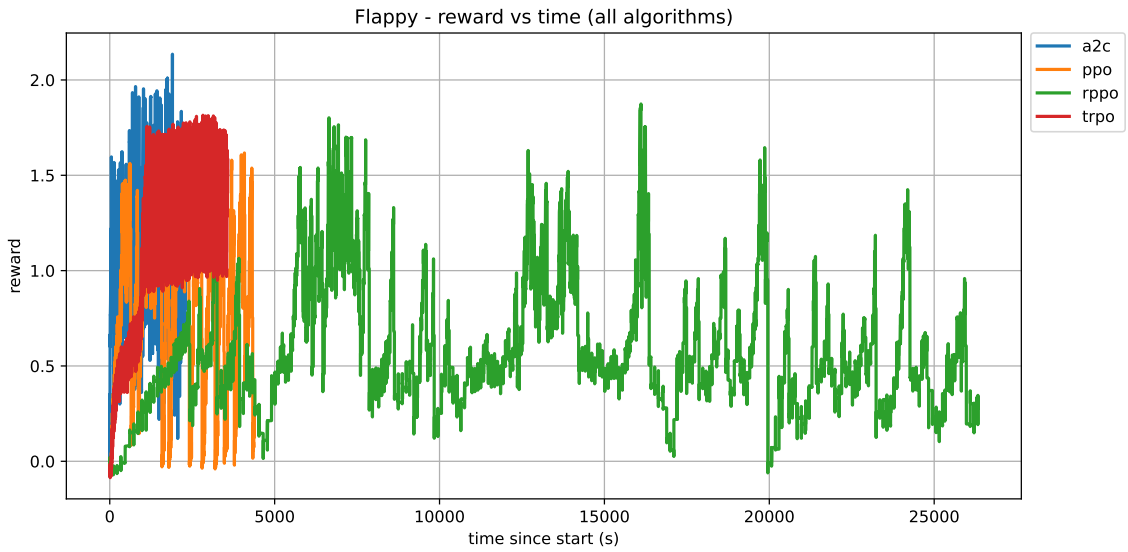

Figure 5.3: Flappy Bird: episode reward versus wall-clock training time for all four algorithms.

A2C reaches its plateau reward within the first 1,000 seconds and TRPO follows shortly after. PPO makes its transition around 3,000 to 4,000 seconds. RPPO remains below 0.5 reward until around 5,000 seconds, making step-like jumps with long flat periods in between. The Hard difficulty failure is not visible in these training curves since training was done on Normal; it only appears at evaluation.

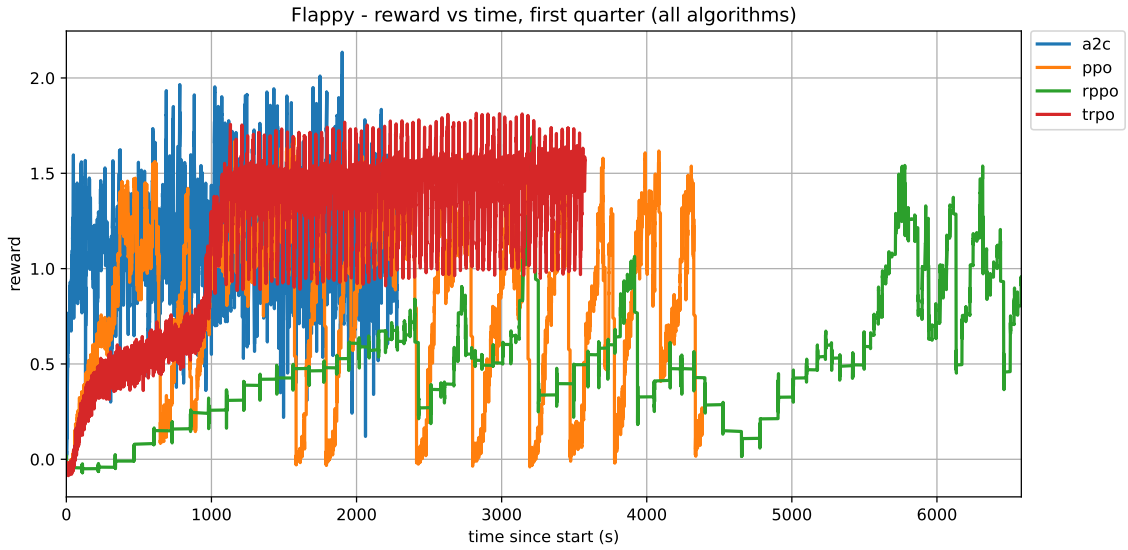

Figure 5.4: Flappy Bird: episode reward versus wall-clock training time, first quarter of training only.

A2C reaches 100+ pipes earliest but with the highest variance, including peaks above 140 and drops back to 0. TRPO settles into a consistent band around 60 to 120 pipes after 4 million steps. PPO and RPPO show the widest swings throughout, with periods of zero pipes mixed with peaks above 100. Every jump in shaped reward corresponds to a jump in pipes cleared, confirming the centring and survival bonuses work alongside pipe collection rather than being optimised independently.

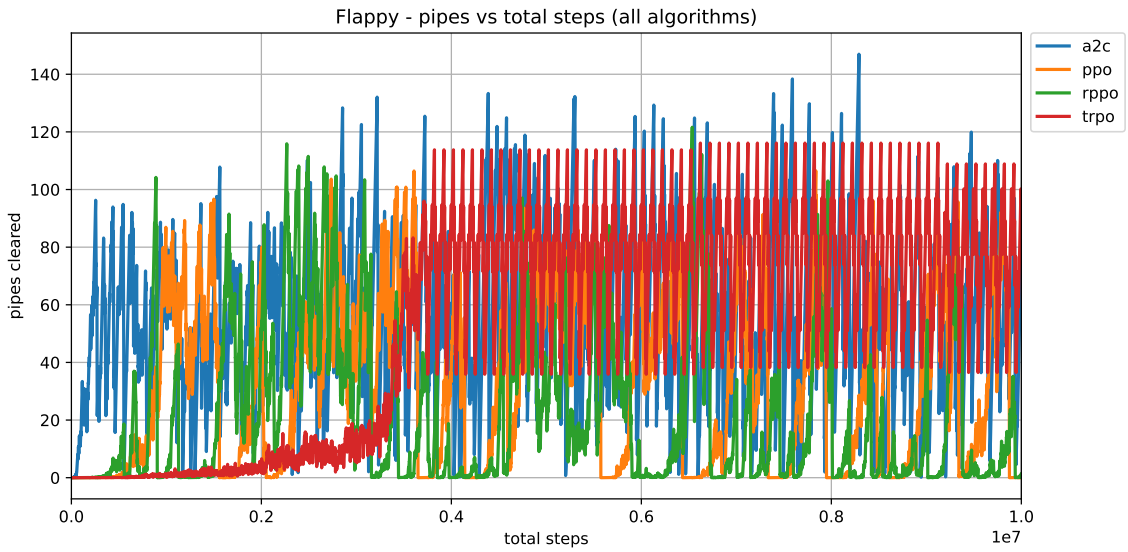

Figure 5.5: Flappy Bird: pipes cleared per episode versus training steps for all four algorithms.

5.1.5 Automated Playtesting

Table 5.1 shows the full comparison across all agents and difficulties. Figure 5.6 shows the evaluation bar chart.

Table 5.1: Flappy Bird: pipes cleared per episode across all agents and difficulties (mean and best-episode score over 100 episodes for DRL and Random agents; Human data from a small number of episodes per difficulty). Entries marked † indicate agents that survived the full 30-minute step limit (108,000 steps) every episode; the score reflects the maximum achievable within that cap. Steps are not recorded for Human on Easy.

Agent	Difficulty	Mean	Best	Avg. Steps	Train Time
A2C	Easy	1661.4†	1662	108,000	38 min
A2C	Normal	1661.4†	1662	108,000	38 min
A2C	Hard	0.0	0	93	38 min
PPO	Easy	1661.4†	1662	108,000	1h 13m
PPO	Normal	1640.0	1662	106,607	1h 13m
PPO	Hard	0.0	1	93	1h 13m
RPPO	Easy	1661.4†	1662	108,000	7h 20m
RPPO	Normal	1641.9	1662	106,734	7h 20m
RPPO	Hard	0.0	0	89	7h 20m
TRPO	Easy	1661.4†	1662	108,000	1h 0m
TRPO	Normal	1661.4†	1662	108,000	1h 0m
TRPO	Hard	2.5	11	257	1h 0m
Human	Easy	194.3	305	—	—
Human	Normal	25.3	78	445	—
Human	Hard	0.0	0	91	—
Random	Easy	0.0	0	51	—
Random	Normal	0.0	0	51	—
Random	Hard	0.0	0	51	—

On Easy and Normal difficulties, all four DRL algorithms survived the full 30-

minute episode limit consistently, effectively solving both difficulties within the evaluation budget. Hard was a sharp contrast: A2C, PPO, and RPPO failed to clear any pipes on average, while TRPO managed a mean of 2.5 pipes with a best episode of 11. The random agent cleared no pipes at all, surviving around 51 steps per episode before crashing.

The human baseline averaged 194.3 pipes on Easy and 25.3 pipes on Normal, scoring zero on Hard. On Easy and Normal, the human cleared a meaningful number of pipes but remained well below the DRL ceiling of 1,661. This gap is consistent with the nature of the task: a human can navigate gaps reliably but cannot sustain the reaction precision required over hundreds of consecutive pipe gaps without error.

Figure 5.6 makes the Easy/Normal versus Hard split immediately visible. On Easy and Normal the bars for all DRL algorithms are equal and maxed out, showing that the trained policies generalise perfectly to a wider gap. Hard collapses everything to near zero except for the small TRPO bar, making it easy to see at a glance that the 75-pixel gap represents a qualitative threshold rather than a gradual decline.

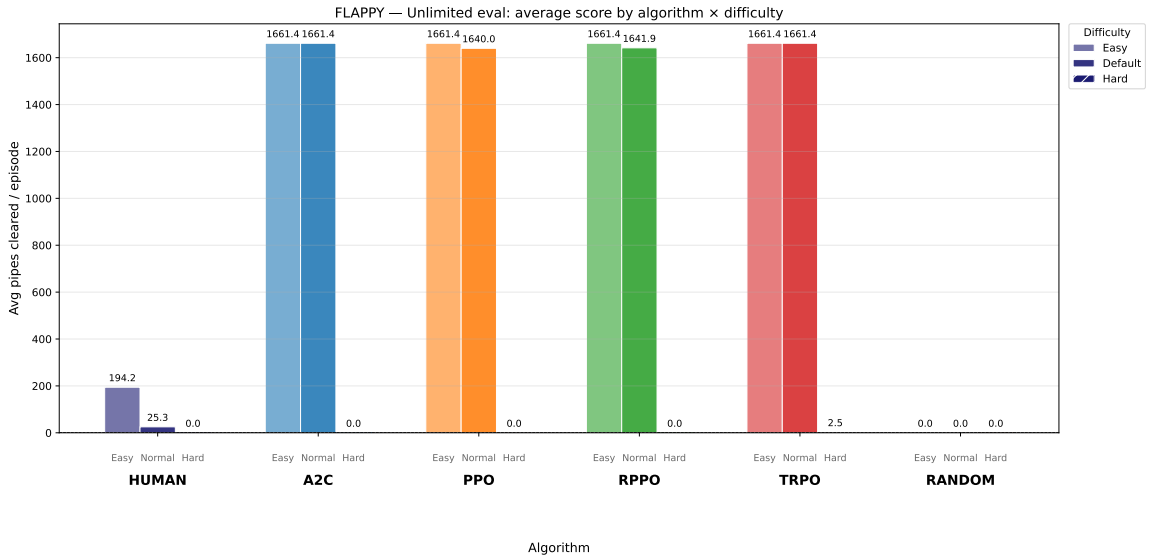

Figure 5.6: Flappy Bird: mean pipes cleared across algorithms and difficulty levels (100 episodes each).

5.1.6 Outcomes

All four algorithms fully solved Easy and Normal difficulty, consistently reaching the 30-minute episode cap of 1,661 pipes. On Hard, three of the four algorithms scored zero; TRPO was the only algorithm to clear any pipes, managing a mean of 2.5 with a best episode of 11. The human player averaged 194.3 pipes on Easy and 25.3 on Normal, well below the DRL ceiling, and also scored zero on Hard. A2C had the fastest training time at 38 minutes and matched the top performers on Easy and Normal, making it the most cost-effective option for this game. RPPO took over seven hours and reached the same ceiling as A2C, representing the worst training efficiency in the benchmark.

Flappy Bird required moderate iteration overall and was the least complex game to implement. The initial reward function, which simply rewarded pipe clears and penalised death, produced agents that cleared around 100 pipes on Normal difficulty. The gap-centring fix was the only major iteration: agents clearing near the pipe edge lacked the vertical margin to survive the next flap decision, and this failure was only apparent from watching the agent play rather than from training curves alone. Adding the centring bonus resolved the issue and brought all four algorithms to the 30-minute episode cap on Easy and Normal.

The Hard difficulty result was a surprise. All agents were trained on Normal and transferred perfectly to Easy, but three of the four algorithms scored zero on Hard. The 75-pixel gap is too narrow for the policies learned on Normal to navigate reliably. This result was not predictable from the training curves alone, and it shows that evaluating at multiple difficulty levels is necessary to characterise what a trained agent has actually learned.

5.2 Experiment #2 - Pong

Pong required fewer iterations than Flappy Bird and far fewer than Snake. With just a ball, two paddles, and a score target, an agent that tracks the ball consistently will perform well. Most of the implementation work went into balancing the CPU opponent so it was competitive but not unbeatable, giving the agent useful signal from both winning and losing points.

5.2.1 What is Pong?

Pong is one of the earliest arcade games. It consists of a player paddle, a ball, and a CPU-controlled opponent paddle. The goal is to be the first to reach a score of 10 points by hitting the ball past the opponent's paddle. Each rally begins with the ball launched in a random direction, and the ball accelerates slightly with each paddle contact, increasing the reaction demands as the rally continues.

Pong was chosen for the benchmark because it represents a qualitatively different challenge from Flappy Bird and Snake: success requires continuous reactive tracking of a moving object and anticipating its trajectory, rather than navigating obstacles or planning paths. It is also the only adversarial game in the benchmark, where the agent must respond to a dynamic CPU opponent rather than a fixed environment.

Figure 5.7: Screenshot of the Pong game environment.

5.2.2 Game Difficulties

Pong's difficulty is controlled by the height of the player's paddle. A larger paddle makes it easier to intercept the ball, while a smaller paddle demands more precise positioning. The ball speed and opponent speed are identical across all difficulty levels. We used the doubling and halving heuristic to create the game's difficulty: Hard is twice Normal and Easy is half Normal.

- On **Easy**, the player paddle height is `180` pixels (2× default);
- On **Normal**, the player paddle height is `90` pixels;
- On **Hard**, the player paddle height is `45` pixels (0.5× default).

5.2.3 Experiment Setup

Action Space The agent has three discrete actions: `stay`, `move up`, and `move down`, mirroring the inputs available to a human player. The action space is discrete; exactly one action is issued per timestep.

Episode An episode ends when one player reaches 10 points. The performance metric is point differential: agent score minus opponent score, ranging from -10 (agent loses every point) to $+10$ (agent wins every point). During training, episodes are additionally capped at 10,000 steps; since a full game to 10 points typically takes 19,000 to 37,000 steps, most training episodes end by truncation rather than by a player reaching the score limit. Evaluation removes this cap and runs each episode to natural completion.

Observation Space The agent observes six values: the ball's normalised x and y position on the screen, the ball's x and y velocity scaled to a reasonable range, and the normalised vertical positions of both paddles. This provides the agent with everything needed to track the ball and anticipate its trajectory.

Reward Function The Pong reward function is built around two ideas: rewarding the agent for keeping the rally alive, and rewarding it more for scoring later in a closer match.

Rally reward. Each step the agent receives a small rally reward that grows with rally length, raised to a power and capped, so longer rallies are intrinsically more valuable.

Scoring bonus. When a point is scored, the agent receives a base bonus plus a small multiplier based on the total number of points already on the board. Conceding a point applies the same formula as a penalty.

```
rally_reward = min(rally_scale * rally_len ^ rally_pow,
   rally_cap)
r += rally_reward


total_points = score_left + score_right
```

```
bonus = score_base + score_progress * total_points


if agent scored:     r += bonus
if opponent scored: r -= bonus
```

5.2.4 Training the DRL Agents

All four algorithms were trained for 10 million timesteps using 10 parallel environments on an AMD Ryzen 5 8600G, with the same hyperparameter configuration across all games.

TRPO, PPO, and RPPO cluster at the top of the chart, stabilising around 0.04 to 0.06 reward after the first 2 million steps. A2C is a clear outlier, plateauing near 0.01 to 0.02 and never closing the gap. Pong showed the cleanest training curves of the three games; all four algorithms improved steadily with low variance compared to Snake or Flappy, reflecting the simpler and more consistent observation space.

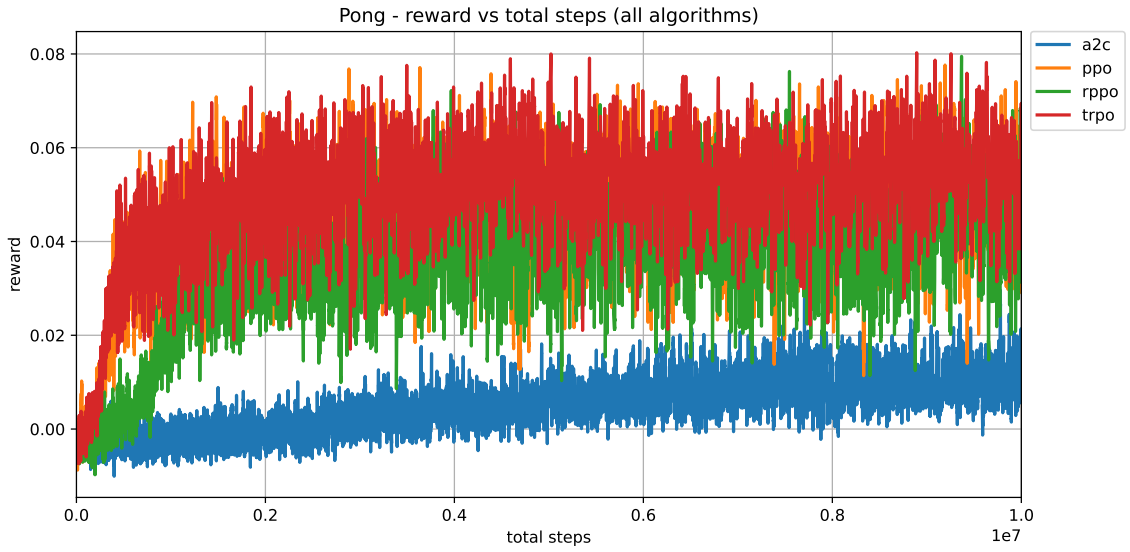

Figure 5.8: Pong: episode reward versus training steps for all four algorithms.

A2C finishes around 1,800 seconds, PPO and TRPO around 4,500 seconds, and RPPO extends to 15,000 seconds. Despite taking roughly three times as long as PPO and TRPO, RPPO converges to a similar final reward ceiling. A2C was fastest at 30 minutes and achieved strong Normal results, but the policy depended on paddle size; it broke down on Hard with a 45-pixel paddle despite this early advantage.

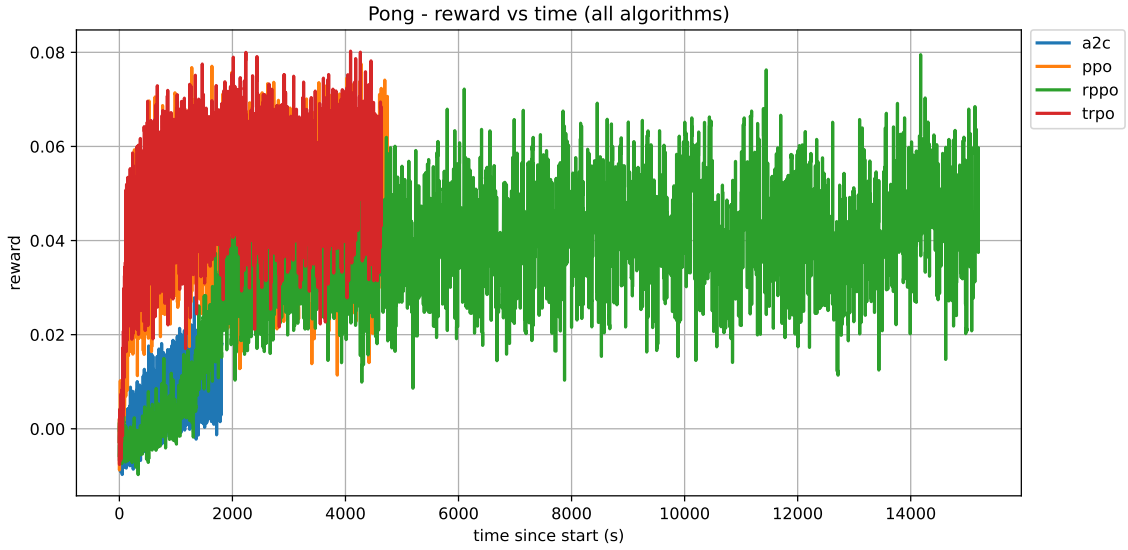

Figure 5.9: Pong: episode reward versus wall-clock training time for all four algorithms.

PPO and TRPO reach their plateau near 0.05 reward within the first 500 seconds. RPPO climbs more slowly, crossing 0.03 around 2,000 seconds. A2C stalls near 0.01 to 0.02 for the entire window and does not make the same transition to a higher reward band. PPO and TRPO are both fast to converge and produced the most robust policies; A2C's plateau stall here foreshadows its Hard-difficulty collapse at evaluation.

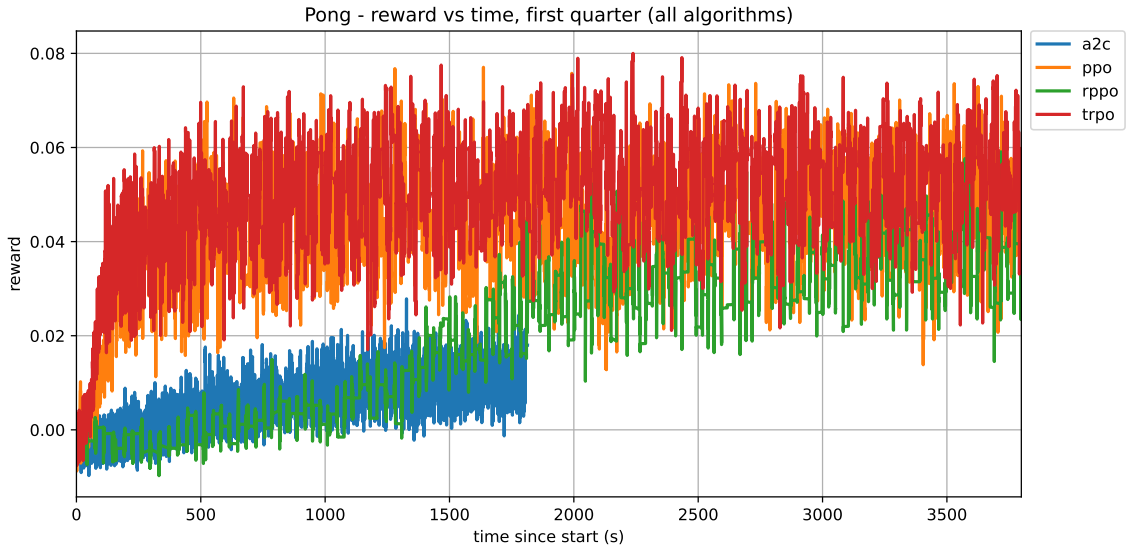

Figure 5.10: Pong: episode reward versus wall-clock training time, first quarter of training only.

A2C sits between -3 and -1 for the entire training run, meaning the CPU wins more points throughout. TRPO, PPO, and RPPO all cross above 0 within the first 1 million steps and stabilise around $+1$ to $+2$. The differentials are small because training episodes are truncated at 10,000 steps before either player reaches 10 points, so the curve reflects partial-game scoring rather than full match outcomes. The ordering and direction of the curves are still meaningful: a consistently positive differential means the agent is scoring more points than it concedes within each truncated episode, and that signal is sufficient for the policy to learn effective ball-tracking.

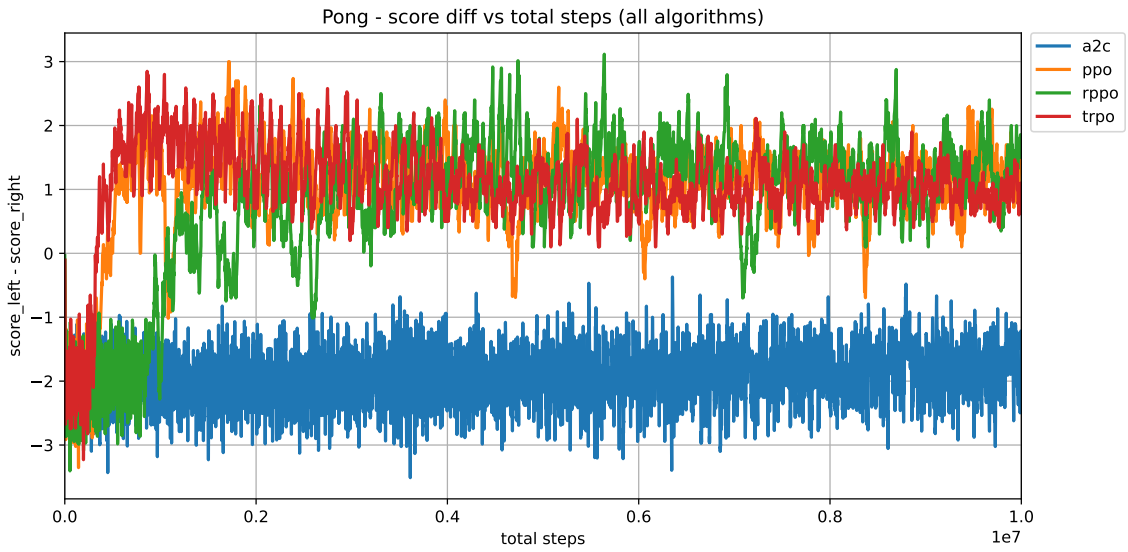

Figure 5.11: Pong: point differential (agent score minus opponent score) versus training steps.

5.2.5 Automated Playtesting

Table 5.2 shows the full comparison across all agents and difficulties. Score is the point differential at the end of the match (agent points minus opponent points), ranging from -10 (agent loses all points) to $+10$ (agent wins all points). Figure 5.12 shows the evaluation bar chart.

Table 5.2: Pong: point differential per episode across all agents and difficulties (mean and best-episode score over 100 episodes for DRL and Random agents; Human data from a small number of episodes per difficulty). Score = agent points − opponent points (−10 to +10; positive = agent wins the match).

Agent	Difficulty	Mean	Best	Avg. Steps	Train Time
A2C	Easy	+10.0	+10	28,027	30 min
A2C	Normal	+7.9	+10	19,189	30 min
A2C	Hard	−4.0	+5	7,531	30 min
PPO	Easy	+10.0	+10	37,293	1h 19m
PPO	Normal	+10.0	+10	34,050	1h 19m
PPO	Hard	+9.0	+10	8,021	1h 19m
RPPO	Easy	+10.0	+10	33,181	4h 14m
RPPO	Normal	+9.6	+10	30,692	4h 14m
RPPO	Hard	+6.9	+10	10,638	4h 14m
TRPO	Easy	+10.0	+10	32,450	1h 17m
TRPO	Normal	+8.8	+10	31,801	1h 17m
TRPO	Hard	+9.6	+10	8,946	1h 17m
Human	Easy	+8.8	+10	9,539	—
Human	Normal	+8.2	+10	16,854	—
Human	Hard	+8.6	+10	9,097	—
Random	Easy	−10.0	−10	3,291	—
Random	Normal	−10.0	−10	2,342	—
Random	Hard	−9.7	−8	2,193	—

All four DRL algorithms achieved a perfect mean of +10 on Easy. On Normal, PPO was the only one to stay perfect; RPPO and TRPO held up well while A2C

started showing some inconsistency. Hard showed the biggest split: TRPO, PPO, and RPPO stayed competitive with means of +9.6, +9.0, and +6.9, while A2C dropped to −4.0 on average, meaning the CPU was winning more often than not. The random agent lost every match with a score of −10.

The human baseline was competitive across all three difficulties, averaging +8.8, +8.2, and +8.6 on Easy, Normal, and Hard respectively. On Normal, the human outperformed A2C (+7.9). On Hard, the human (+8.6) outperformed both RPPO (+6.9) and A2C (−4.0), and came close to PPO (+9.0) and TRPO (+9.6). Pong is the only game in the benchmark where a human player remains competitive with the strongest DRL algorithms; the reactive tracking required by Pong is a task humans perform naturally, whereas the long-horizon spatial planning required by Snake is not.

Figure 5.12 visualises this across all agents and difficulties in one chart. The most striking feature is A2C's bar on Hard dropping well below zero while every other DRL algorithm stays positive. The chart also shows that the gap between the three stronger algorithms (PPO, TRPO, RPPO) on Hard is relatively small compared to A2C's collapse, suggesting the Hard difficulty is genuinely challenging but within reach for algorithms that converge to a more generalised policy.

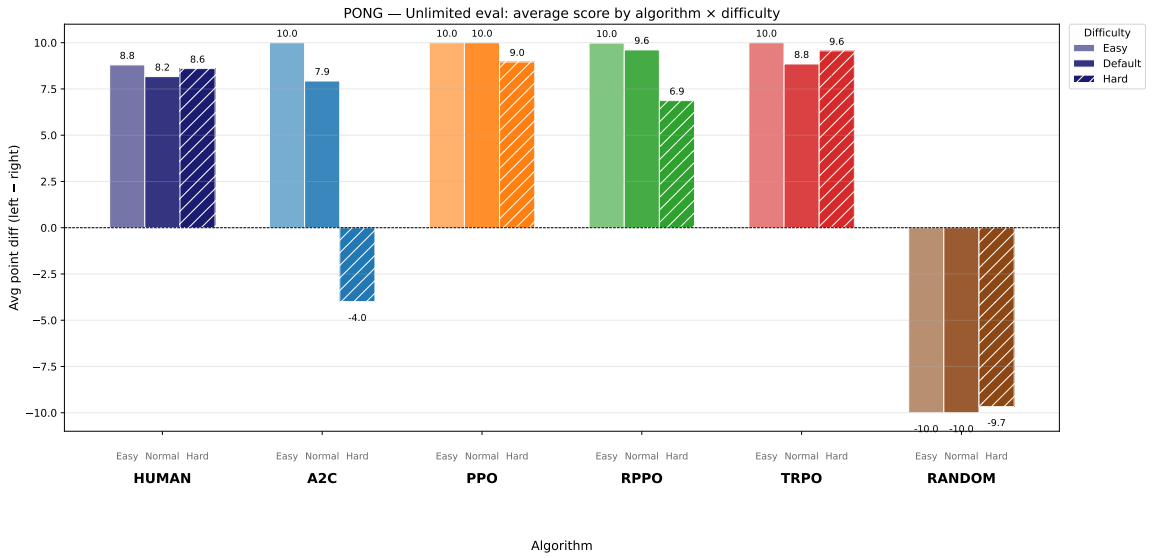

Figure 5.12: Pong: mean point differential across algorithms and difficulty levels (100 episodes each).

5.2.6 Outcomes

Three algorithms performed strongly across all difficulties. PPO was perfect on both Easy and Normal (+10.0) and remained highly competitive on Hard (+9.0). TRPO was the most robust on Hard (+9.6) and converged in around 70 minutes. RPPO took over four hours but reached a similar ceiling to PPO and TRPO. A2C trained the fastest at 30 minutes and scored well on Easy and Normal, but collapsed on Hard with a mean of −4.0. The human baseline was the notable finding in Pong: humans averaged +8.8, +8.2, and +8.6 across Easy, Normal, and Hard respectively, outperforming A2C on Normal and Hard, and finishing close to the strongest algorithms on Hard. Pong is the only game in the benchmark where a human player remains competitive with the best DRL agents.

Pong required the fewest iterations of the three games. The observation space and action space were straightforward from the start, and the reward function needed only a few tuning cycles to produce a competitive agent. The main implementation

challenge was balancing the CPU opponent: an opponent that was too fast eliminated the learning signal by making it impossible for the agent to score, while an opponent that was too slow allowed the agent to win without developing a genuine tracking strategy. Using a reaction-frame delay, giving the CPU a fixed number of frames before it responds to the ball's direction change, kept matches competitive without making the opponent unbeatable.

The biggest surprise in the results was A2C's Hard-difficulty collapse. A2C achieved a mean of $+7.9$ on Normal, indicating a strong policy, but dropped to -4.0 on Hard with the 45-pixel paddle. The other three algorithms maintained positive differentials on Hard, suggesting that A2C's policy overfits to the paddle size seen during training and does not generalise to the smaller target. This result is consistent with A2C's fastest convergence: it found a working strategy on Normal quickly but did not explore enough to develop a policy that is robust to a smaller paddle.

5.3 Experiment #3 - Snake

5.3.1 Summary

Snake was the most challenging game to implement in this benchmark. The objective is straightforward: collect as many apples as possible without colliding with the walls or the snake's own body. However, the agent consistently discovered a simpler strategy: orbit the apple indefinitely without collecting it. Suppressing that behaviour without destroying the learning gradient required more reward function iterations than either Flappy Bird or Pong. Despite this, the trained agents generalized well: performance remained stable across all three difficulty levels, and all DRL algorithms substantially exceeded the human baseline, which averaged 17.3 apples on Normal compared to 50+ for TRPO and A2C.

5.3.2 What is Snake?

Snake is a classic tile-based arcade game, originating in the 1970s and widely recognized from its inclusion on Nokia mobile phones. The player controls a continuously moving snake on a bounded grid to collect as many apples as possible. Each apple collected extends the snake's body by one tile, reducing the available free space on the grid and making safe navigation increasingly difficult over time. The game ends immediately if the snake collides with a wall boundary or its own body.

Snake was chosen for this benchmark because it represents a qualitatively different class of problem from Flappy Bird and Pong. Rather than requiring precise reactive control, Snake demands long-term spatial planning: an agent that greedily chases the nearest apple without considering its future path will routinely trap itself in a dead end. The difficulty also scales naturally with agent performance; the more apples collected, the longer the snake grows, and the harder the remaining free space

becomes to navigate. This self-compounding challenge makes Snake a strong test of an agent's ability to reason about long-horizon consequences within an otherwise deterministic, rule-simple environment.

We implemented Snake using Pygame, as described in Chapter 4. The game world is a 500×500 pixel window divided into a 50×50 tile grid at 10 pixels per tile. At the start of each episode, the snake spawns with a body length of one tile at a random grid position. A single apple is placed at a random unoccupied tile. When the snake's head reaches that tile, the body grows by one tile, and a new apple immediately spawns at another random unoccupied location.

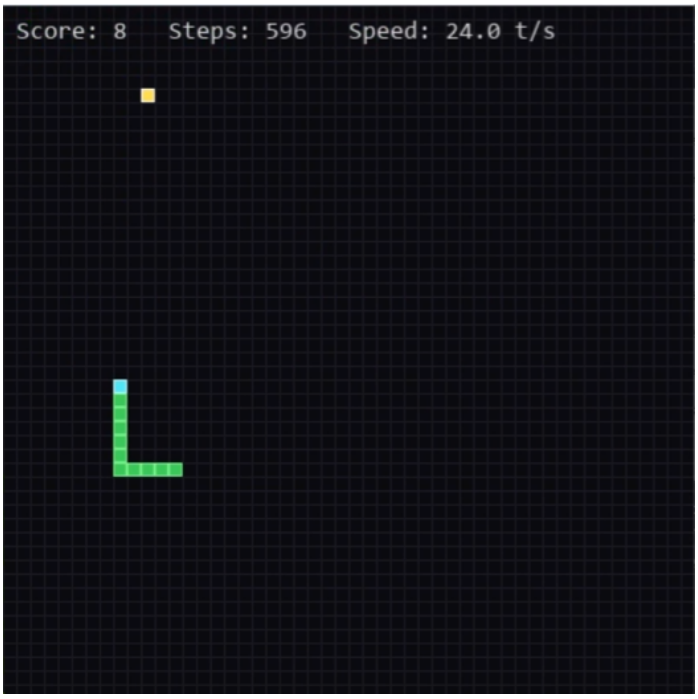

Figure 5.13: Screenshot of Snake environment.

5.3.3 Game Difficulties

Snake's difficulty is primarily a function of movement speed. All three difficulty levels (easy, normal, and hard) share the same starting speed of *16 tiles per second* and the

same maximum speed of *26 tiles per second*. The only parameter that differs is the rate of acceleration per apple collected. We used the doubling and halving heuristic to create the game's difficulty: Hard is twice Normal and Easy is half Normal.

- On **Easy**, the snake accelerates by `0.25` tiles/s per apple;
- On **Normal** the snake accelerates by `0.5` tiles/s per apple;
- On **Hard** the snake accelerates by `1.0` tiles/s per apple.

5.3.4 Experiment Setup

Action Space The agent has four discrete actions: `move up`, `move right`, `move down`, and `move left`, mirroring the directions available to a human player. The action space is discrete, exactly one action is issued per timestep, and it is not possible to combine two directions simultaneously. Each action sets the snake's heading for the next movement step.

Episode An episode ends when one of two terminal conditions is met: the snake's head moves into a wall boundary, or the snake's head moves into a tile occupied by its own body. Either event terminates the episode immediately and triggers the death penalty in the reward function.

Observation Space The observation consists of two parts: a local spatial grid and six global scalar values.

- **Local grid.** The agent receives a 7×7 grid centred on the snake's head (49 cells). Each cell contains a value representing how much closer moving to that cell would bring the snake to the apple relative to its current distance: positive values indicate progress, negative values indicate moving away. Cells occupied

by a wall or a body segment are assigned a large negative value to mark them as lethal. Cells immediately adjacent to a lethal cell also receive a small negative *bleed penalty*, giving the agent advance warning before it steps into a fatal position.

- **Global scalars.** Six scalar values are appended to the flattened grid:
  - **Direction (dx, dy):** the snake's current movement direction, remapped to $[0, 1]$ per axis ($0 =$ negative, $0.5 =$ stationary on that axis, $1 =$ positive), indicating which way the snake is heading;
  - **Apple direction (ax, ay):** the normalised vector from the snake's head to the apple on the full grid, giving the agent a global pointer to the goal regardless of local grid contents;
  - **Speed:** the snake's current tiles-per-second speed, normalised between the minimum (base) and maximum speed;
  - **Length:** the snake's current body length, normalised by the total number of grid tiles.

The local grid provides immediate awareness of danger and nearby structure; the global scalars supply context about the agent's heading, the apple's position on the full grid, and how far the episode has progressed.

Reward Function Snake required the most iterations of any game in the benchmark. Rewarding the agent for moving closer to the apple was necessary but not sufficient: without counterpressure, the agent would orbit the apple indefinitely rather than collect it.

Terminal events. Collecting an apple gives a reward of $+10$; a death penalty of -5 applies on termination.

Shaping terms. Six signals layer on top of the terminal events:

- **Potential shaping:** a step reward of $1.5 \times \Delta\text{dist}$, where Δdist is the reduction in Euclidean distance to the apple that step.
- **Local field:** a reward of $0.5 \times v$ where v is the value of the cell the snake just entered in the observation grid, rewarding moves already flagged as good by the local grid.
- **Turning penalty:** a penalty of -0.15 when the agent turns while already within 0.25 normalised units of the apple, discouraging unnecessary direction changes near the goal.
- **Oscillation penalty:** a flat -0.5 when an ABAB direction pattern is detected, directly targeting the orbiting behaviour.
- **Stall penalty:** starts at -0.1 after two steps without distance progress and ramps by 0.1 per additional step, capped at -0.8, to break sustained orbits without destroying the learning gradient.
- **Time pressure:** a fixed -0.005 per step to prevent idling.

The full reward is clipped to $[-10, +15]$ to prevent outliers from destabilising training.

```
if score_inc:   r += 10.0                # Apple collected
if terminated:  r -= 5.0                 # Death


r += 1.5 * dist_delta                    # Potential shaping
r += 0.5 * move_delta                    # Local field


if d < 0.25 and turning:
```

```
    r -= 0.15                                   # Turning penalty
       near apple
if oscillating:
    r -= 0.5                                    # Oscillation (ABAB
       pattern)
if no_progress_steps >= 2:
    r -= min(0.8, 0.1 * (no_progress_steps - 1))  # Stall ramp


r -= 0.005                                      # Time pressure
r  = clip(r, -10.0, 15.0)
```

5.3.5 Training the DRL Agents

All four algorithms were trained for 10 million timesteps using 10 parallel environments on an AMD Ryzen 5 8600G, with the same hyperparameter configuration across all games.

All four algorithms start near -0.4 and rise sharply in the first 1 to 2 million steps, after which reward plateaus with high variance throughout. PPO and TRPO tend to sit at the top of the band (≈ 0.5–0.6), A2C settles slightly lower, and RPPO is the most unstable, with two large periodic drops before recovering. TRPO held the highest reward throughout training, consistent with its position at the top of the evaluation results.

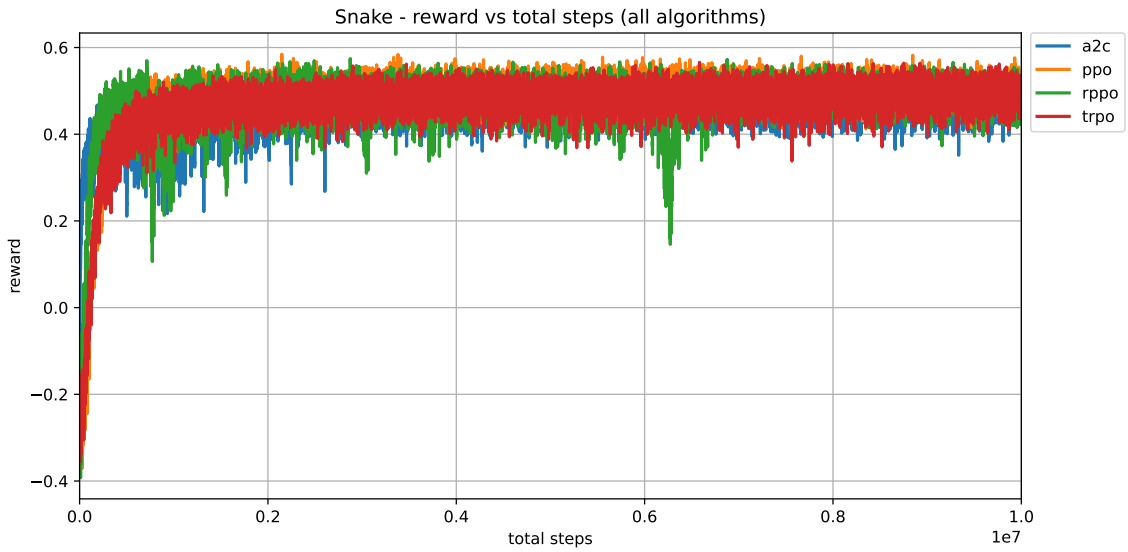

Figure 5.14: Snake: episode reward versus training steps for all four algorithms.

A2C and TRPO finish within the first 5,000 seconds (≈47 min and ≈70 min respectively), PPO extends to around 7,500 seconds, and RPPO occupies the full 22,000-second axis at roughly six hours. Despite the extra time, RPPO does not reach a higher reward ceiling. PPO took more than twice as long as A2C and settled at a lower ceiling. RPPO's LSTM adds overhead without benefit: the local observation grid already captures the short-term state the agent needs, and the orbiting problem is a reward design challenge rather than a memory problem.

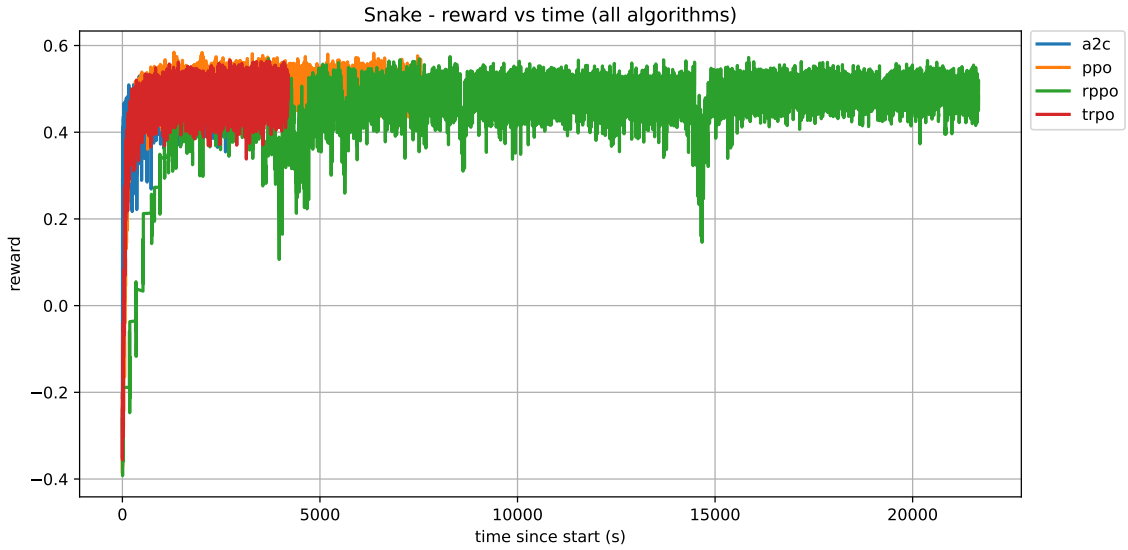

Figure 5.15: Snake: episode reward versus wall-clock training time for all four algorithms.

A2C, PPO, and TRPO all converge to their plateau reward within the first 500 seconds, with no meaningful improvement after that point. A2C had the steepest early-training progress of any algorithm, finding a working strategy quickly and continuing to refine it. RPPO is still climbing throughout this window, showing step-like increases punctuated by deep drops, and has not found a stable strategy by the time the other three have finished converging.

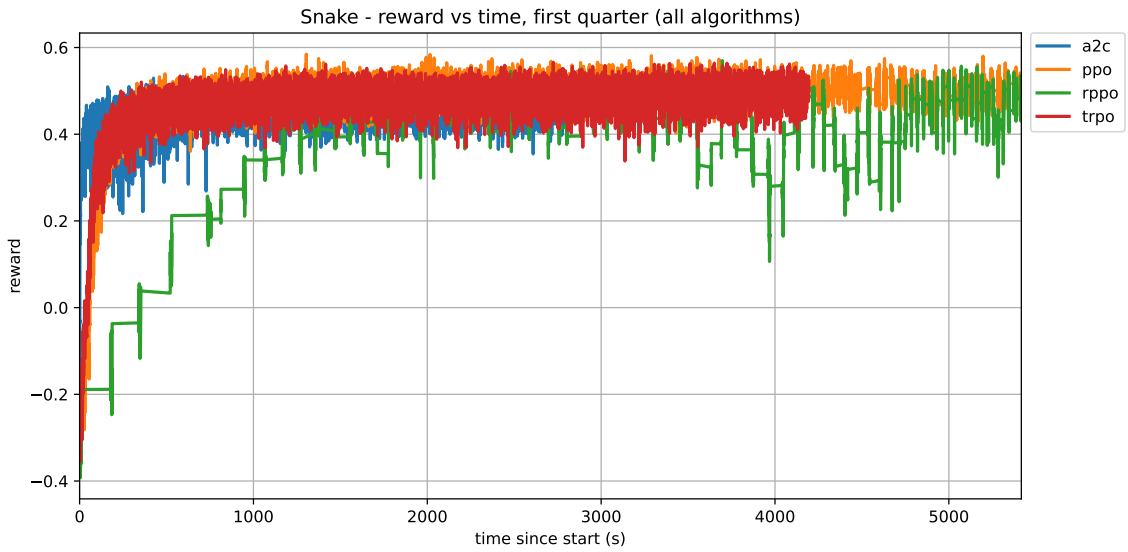

Figure 5.16: Snake: episode reward versus wall-clock training time, first quarter of training only.

Four clearly separated bands emerge: TRPO at the top (peaking above 70 apples, settling around 40 to 55), A2C just below (30 to 60), PPO in the middle (20 to 35), and RPPO at the bottom (10 to 25). The ordering is stable from around 2 million steps onward and matches the evaluation results exactly. Since the reward function includes many shaping terms beyond apple collection, it is possible for reward to increase without apple count improving; the apples curve confirms this is not the case, meaning the reward is well-calibrated for this task.

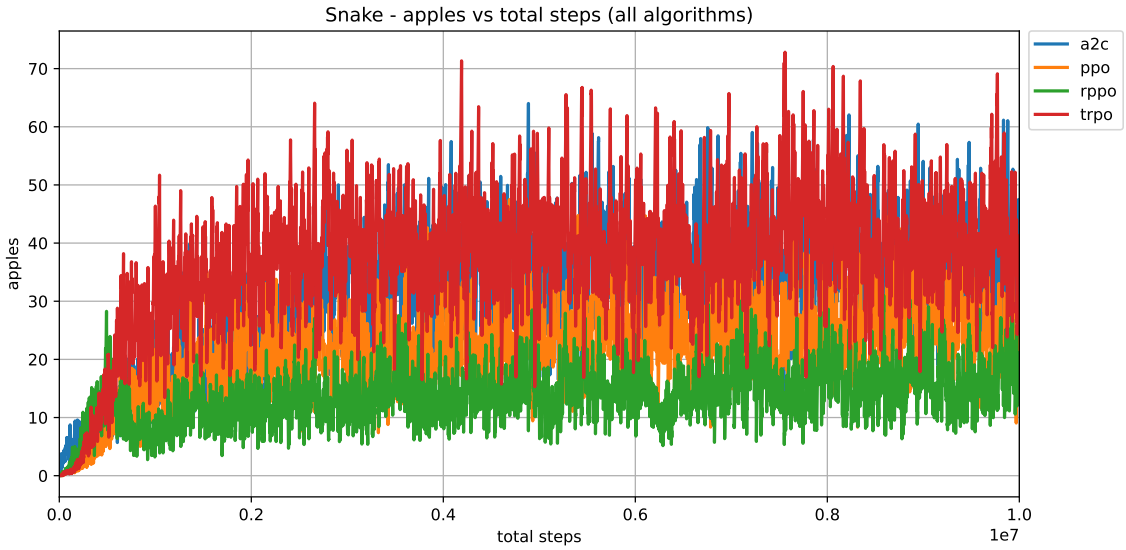

Figure 5.17: Snake: apples collected per episode versus training steps for all four algorithms.

5.3.6 Automated Playtesting

Table 5.3 shows the full comparison across all agents and difficulties, and Figure 5.18 shows the evaluation bar chart. TRPO and A2C were the strongest performers overall, with TRPO hitting the highest mean across all three difficulties. The random agent collected virtually no apples. RPPO was the weakest DRL algorithm despite taking the longest to train.

The human baseline averaged 6.7 apples on Easy, 17.3 on Normal, and 5.1 on Hard. Every DRL algorithm substantially exceeded human performance on all three difficulties: the weakest DRL result (RPPO Easy at 26.1) was still nearly four times the human Easy mean. Snake is the game where the gap between DRL and human performance is largest. The long-horizon spatial planning required to navigate the grid without self-collision, while continuously tracking an apple that can appear anywhere, is a task that trained DRL agents handle consistently and humans do not.

Table 5.3: Snake: apples collected per episode across all agents and difficulties (mean and best-episode score over 100 episodes for DRL and Random agents; Human data from a small number of episodes per difficulty). Score = apples collected.

Agent	Difficulty (level)	Mean (score)	Best (score)	Avg. Steps	Train Time
A2C	Easy	48.7	108	1,688	47 min
A2C	Normal	50.6	88	1,755	47 min
A2C	Hard	48.8	86	1,686	47 min
PPO	Easy	34.9	79	1,206	2h 6m
PPO	Normal	37.6	68	1,300	2h 6m
PPO	Hard	41.4	99	1,446	2h 6m
RPPO	Easy	26.1	63	924	6h 2m
RPPO	Normal	26.9	66	949	6h 2m
RPPO	Hard	30.9	70	1,071	6h 2m
TRPO	Easy	52.7	96	1,854	1h 10m
TRPO	Normal	51.6	93	1,818	1h 10m
TRPO	Hard	54.4	111	1,934	1h 10m
Human	Easy	6.7	25	1,259	—
Human	Normal	17.3	22	3,041	—
Human	Hard	5.1	16	997	—
Random	Easy	0.0	1	103	—
Random	Normal	0.1	1	113	—
Random	Hard	0.0	1	88	—

One counterintuitive result is that PPO and TRPO both score higher on Hard than on Easy. PPO goes from 34.9 on Easy to 41.4 on Hard, and TRPO from 52.7 to 54.4. The likely explanation is that Hard's faster acceleration rate per apple

makes orbiting more costly: the snake reaches dangerous speeds sooner, so the agent's stall and oscillation penalties kick in earlier and more forcefully. Hard inadvertently suppresses the main failure mode better than Easy does, where the snake can orbit slowly for longer before the penalties accumulate enough to redirect it.

Figure 5.18 summarises these results visually, grouping all algorithms side by side per difficulty. It makes immediately apparent that performance is stable across all three difficulty levels for every DRL algorithm, with no algorithm showing a meaningful drop on Hard relative to Easy or Normal. This stability is the defining characteristic of Snake's results and it stands in contrast to what we saw in Flappy Bird and Pong.

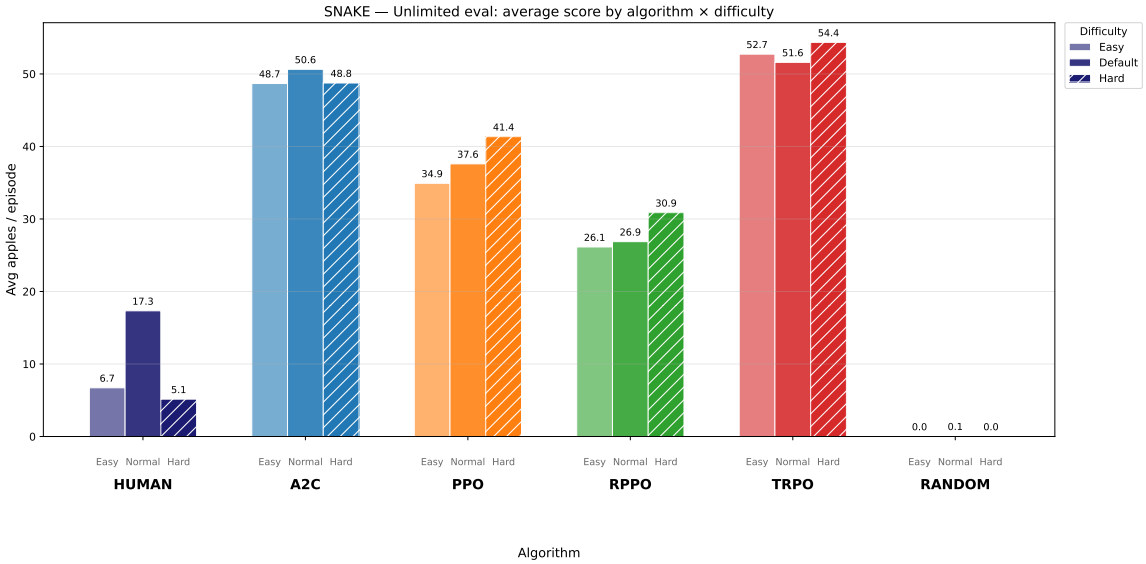

Figure 5.18: Snake: mean apples collected across algorithms and difficulty levels (100 episodes each).

5.3.7 Outcomes

The trained agents substantially exceeded the human baseline across all three difficulties. TRPO and A2C were the strongest performers, both averaging over 50 apples on Normal compared to the human baseline of 17.3. Performance was stable across

all three difficulty levels for every algorithm; no DRL agent showed a meaningful drop on Hard relative to Easy or Normal. This stability is the defining result of Snake: the change in acceleration rate between difficulty levels did not meaningfully affect agent performance, suggesting that speed within this range is not the limiting factor for trained policies. RPPO was the weakest algorithm despite taking six hours to train; the LSTM it uses provided no measurable benefit in this fully observable environment.

Snake presented unexpected implementation challenges despite its apparently simple objective. The core goal is to collect as many apples as possible without colliding with the snake's own body or the walls, but the initial approach of providing the apple's position and rewarding proximity quickly led to the agent orbiting the apple indefinitely rather than collecting it. Even with explicit penalties for this behaviour, the only reliable fix we could find was to terminate the episode and apply a heavy penalty when the agent entered a sustained orbit. Even in the best trained models, this behaviour occasionally resurfaces.

This game required the most iteration of any in the benchmark, both in terms of observation design (trying different combinations of inputs, including speed and full body position) and reward function tuning to suppress the orbiting without destroying the learning gradient.

5.4 Summary of the Results

5.4.1 RQ1. Feasibility: How feasible is it for small studios (e.g., indie developers) to implement DRL agents in 2D games in terms of time, resources, and technical complexity?

All 12 agent configurations (4 algorithms $\times$ 3 games) were trained on a single consumer machine: an AMD Ryzen 5 8600G with integrated Radeon 760M graphics and 32 GB of RAM at 6800 MT/s. No discrete GPU was used. Each agent was trained using Stable-Baselines3 [21] for 10 million timesteps with 10 parallel environments, using each algorithm's default hyperparameter configuration without modification.

Which algorithm performed best? TRPO was the strongest algorithm overall. It ranked first or joint-first in all three games and was the only algorithm to clear any pipes on Flappy Hard. PPO was the most consistent, never collapsing on any game or difficulty. A2C produced strong results on Easy and Normal but broke down on Pong Hard (-4.0 mean). RPPO was the weakest performer in every game despite incurring the longest training times.

Which algorithm trained fastest? A2C was the fastest, completing in 30 to 47 minutes per game. TRPO was second at 60 to 70 minutes. PPO took roughly twice as long as A2C. RPPO was by far the slowest, requiring 4 to 7.5 hours per game.

Total workflow time. Tables 5.4 and 5.5 break down the time required for each phase of the DRL playtesting workflow per game. Table 5.4 shows the one-time setup costs: building the Pygame game and Gymnasium wrapper (Game Development) and

writing the first working version of the observation space, action space, and reward function before any iteration began (Initial DRL Setup). Flappy Bird's initial DRL setup took longer because it was the first game developed and required establishing the full pipeline from scratch. Table 5.5 shows the recurring and final-stage costs. DRL Code and Training are incurred once per iteration cycle; Evaluation and Reporting are incurred once per finalized configuration. Evaluation covers 100-episode assessments across all three difficulties and all four algorithms. Flappy Bird's evaluation time is high because well-trained agents on Easy and Normal survive for very long episodes, with each algorithm requiring approximately 10 hours per difficulty; Hard evaluation is much shorter at around 30 minutes per algorithm.

Table 5.4: One-time setup costs per game. Game Development covers building the Pygame game and its Gymnasium wrapper. Initial DRL Setup covers the first implementation of the observation space, action space, and reward function before iteration began.

Game	Game Dev.	Initial DRL Setup	Total
Flappy Bird	4 h	2 h	6 h
Pong	3 h	1 h	4 h
Snake	6 h	1 h	7 h
Total	13 h	4 h	17 h

Table 5.5: Recurring and final-stage costs per game. DRL Code and Training (all) are per-iteration costs incurred each time the observation space, action space, or reward function was modified and retrained. Evaluation and Reporting are incurred once per finalized configuration. Training (all) is the sum across all four algorithms for a single training run.

Game	DRL Code	Train (all)	Eval (total)	Reporting
Flappy Bird	30–45 min	≈10 h	≈82 h	20 min
Pong	30–45 min	≈7.5 h	≈4 h	20 min
Snake	30–45 min	≈10 h	≈3 h	20 min

The one-time setup costs across all three games totalled 17 hours. Evaluation dominated the per-game time budget for Flappy Bird; Snake and Pong were each completed within a single working day of evaluation.

Is DRL-based playtesting feasible for small studios? Based on our experience, the answer is no, at least not without significant upfront investment. The workflow requires both game development skills and machine learning expertise. Stable-Baselines3 handled the DRL side without modification, but designing the observation space and reward function required substantial iteration in all three games. In Snake, suppressing orbiting behaviour alone took more engineering effort than implementing the game itself. Even after that iteration, some configurations still fail completely: three of four algorithms score zero pipes on Flappy Hard, and A2C collapses on Pong Hard despite performing well on Normal. These failures only appeared at evaluation time and were not visible during training.

For the workflow to be practical in a game development context, the full loop (coding, training, evaluating, and reading the results) should be completable in a working day. Using A2C brings the training step close to that target, but the reward function iteration is not automatable and is not captured in the training times above.

A developer must watch the agent play, identify what is going wrong, and redesign the reward; this cycle can repeat many times before the agent behaves correctly. The upfront cost is high, and the benefit depends on the scope of the project: for a game with a single release it is difficult to justify, but for a live service game with ongoing balance updates the investment could pay off over time.

It is also important to note that this benchmark targets game balance, not bug detection. The agents measure whether the game is too easy or too hard at each difficulty level and how well a trained policy transfers to a modified version. Finding unexpected crashes, logic errors, or exploitable mechanics is a different problem that this approach does not address.

5.4.2 RQ2. Performance of the agents: How do DRL agents (PPO, A2C, TRPO, RPPO) perform across different 2D games with varying mechanics and complexity?

Getting the agents to play well. Before any meaningful performance results could be obtained, each agent had to be made capable of playing the game in the first place. This was by far the largest single investment of development time across the entire project. The reward function, observation space, and episode termination logic all had to be designed and iterated on together, and none of them converged quickly.

In Flappy Bird, early versions of the reward function produced agents that would fly straight into pipes consistently. The gap-centring insight took multiple training runs to identify: the agent was technically clearing pipes, but always near the edge, leaving no margin for the next gap. Only by watching the agent play in real time did it become clear that the reward was guiding the agent toward the wrong behaviour.

A single run to discover this failure, modify the reward, and retrain took between 40 minutes and 1.5 hours depending on the algorithm.

In Pong, the main iteration was on the CPU opponent rather than the reward function. An opponent that was too capable meant the agent never scored and had no learning signal; one that was too slow produced an agent that won easily but only by exploiting the speed gap rather than learning to track the ball. Tuning the reaction-frame delay to a range where the agent could learn genuine tracking behaviour took several training runs to get right.

Snake was the most time-intensive by a wide margin. The orbiting problem, where the agent circles the apple indefinitely rather than collecting it, was not an obvious failure from training curves alone. Reward climbed steadily, but apples were not being collected. Diagnosing the failure required watching hundreds of episodes, identifying the pattern, and then designing penalties that would break the orbit without pushing the agent into dying instead. The stall penalty, oscillation detector, turning penalty, and local field term were all added in separate iterations. Each one required a full training run to evaluate, meaning hours of work per change. This cycle repeated more times than for any other game in the benchmark.

Observed performance across games and algorithms. Once reward functions were stable, performance varied considerably across games and algorithms. In Flappy Bird, all algorithms solved Easy and Normal by surviving to the 30-minute time limit, but only TRPO cleared any pipes on Hard, where the human also scored zero. In Pong, PPO and TRPO were the most consistent on Normal and Hard; A2C held up on Easy and Normal but fell apart on Hard with a mean of -4.0. In Snake, TRPO and A2C were the strongest, both averaging over 50 apples on Normal compared to the human baseline of 17.3. All DRL algorithms substantially exceeded human

performance on Snake, reflecting that the long-horizon spatial planning required by the game is within reach for DRL once the reward function is correctly designed. The random agent set a clear floor: zero pipes in Flappy, a guaranteed loss in Pong, and zero apples in Snake.

RPPO was the weakest algorithm in all three games despite taking the longest to train. This is consistent across all three games and is likely explained by the environment design: all three games provide the agent with complete state information, so there is no hidden or partial state for the LSTM to reason over. RPPO's memory mechanism has nothing to work with that the other algorithms do not already have, making the added computational cost unjustified for this class of fully observable 2D games.

Human baseline comparison. The human baseline tells a different story in each game. In Flappy Bird, DRL and humans both failed on Hard, but on Easy and Normal the DRL agents ran indefinitely while humans averaged 194.3 and 25.3 pipes respectively, well below the 1,661-pipe cap the agents hit every episode. Pong was the exception: humans averaged +8.8, +8.2, and +8.6 across Easy, Normal, and Hard, outperforming A2C on Normal and Hard and finishing close to the strongest algorithms. In Snake, DRL agents dominated completely: the weakest DRL result (RPPO Easy at 26.1 apples) was still nearly four times the human Easy mean of 6.7, and humans averaged only 17.3 on Normal against a DRL ceiling above 50. Across the three games, the human-versus-DRL gap tracks directly with how much spatial planning and sustained precision the game demands: Flappy and Snake reward inhuman consistency, while Pong rewards reactive tracking that humans do naturally.

What the performance results actually reflect. The final evaluation scores reflect the best version of each agent after all reward function iterations were complete.

They do not reflect the time and effort required to reach that version. The results look clean in retrospect, but behind each row in the evaluation tables are many hours of training runs that did not produce useful agents. Performance in this sense is not just a property of the algorithm; it is also a measure of how much iteration the developer put in. A developer who runs only a single training attempt per game will likely see much weaker results than those reported here.

5.4.3 RQ3. Generalizability: To what extent can trained DRL agents generalize on modified versions of the same stages within the same game?

Generalisation across difficulty levels. Generalisation was strong on Easy and moderate on Hard across all three games. In Flappy Bird, all algorithms transferred perfectly to Easy (a wider gap), but three of four scored zero on Hard (a narrower gap). The pipe gap width is a harder parameter to generalise across than paddle size or snake speed, because it directly constrains the precision of the only action the agent has. In Pong, all algorithms maintained high performance on Easy, and three of four maintained strong performance on Hard, with only A2C collapsing. In Snake, mean scores remained stable across all three difficulties for every algorithm: the acceleration rate change did not meaningfully affect agent performance, which suggests the trained policies are robust to at least this dimension of variation.

A consistent pattern is that Easy difficulty reveals little: all algorithms perform well and the differences between them are small. Hard difficulty is where genuine generalisation is tested. Agents that learned robust strategies on Normal carry over; agents that overfit to Normal conditions fall apart. In Flappy Bird, three algorithms score zero on Hard while TRPO still manages a mean of 2.5. In Pong, three algorithms

stay above +6 on Hard while A2C crashes to −4.0. In Snake, the ordering is stable but the gap between the strongest and weakest algorithm widens on Hard. Easy tells you whether an agent learned the task at all. Hard tells you whether it learned it well enough to generalise.

Generalisation as a game design tool. Beyond the algorithmic comparison, these results point to a practical application. A developer can train a single agent on the default difficulty version of a game and then evaluate it across many alternative configurations without retraining. Each evaluation is a measurement: how well does the agent perform under this set of parameters? If the agent collapses under a configuration, the parameters are likely too extreme for a policy trained on Normal to handle, which is itself useful feedback about the gap between difficulty levels.

This enables a workflow where the developer adjusts a difficulty parameter (pipe gap width, snake acceleration, paddle size), runs an evaluation batch without retraining, and reads the score. Used this way, a trained agent effectively acts as an automated playthrough of every configuration the developer wants to test. With three games and three difficulty levels, this produced 12 evaluation conditions in this benchmark. But the same pipeline could scale to hundreds of parameter combinations with no additional training: vary the gap from 50 to 400 pixels in steps of 25, evaluate each one, and plot the score curve. The result is a performance profile across the full difficulty range that would take a human tester many hours to produce manually.

In Flappy Bird, this pattern is visible even with only three points: the agent scores are flat and maxed out at 150 and 300 pixels, then drop sharply to near zero at 75 pixels. That transition point is precise and reproducible, and it tells the designer exactly where the Normal-trained policy breaks down. In Pong, the paddle-size profile shows a similarly sharp threshold: three algorithms hold up through 45 pixels but

A2C fails, which a developer could use to decide whether A2C is a useful proxy agent for balancing that game at all. Snake's profile is the most informative for this purpose: because performance is stable across all three levels, a developer can conclude that the acceleration parameter within this range does not meaningfully change how difficult the game is for a trained agent, and might need to adjust a different parameter to produce the desired difficulty spread.

The key constraint on this workflow is that the agent must actually be competent before evaluation results are meaningful. A poorly trained agent will score low on every configuration, making it impossible to distinguish between a well-balanced difficulty curve and a broken reward function. This is where the iteration cost from RQ2 becomes relevant again: the measurement quality is bounded by the quality of the agent, and producing a good agent is the hard part.

5.4.4 RQ4. Developer Perspective: What challenges, trade-offs, and benefits emerge when smaller developers attempt to adopt DRL-based testing?

The biggest recurring challenge was reward function design. Flappy Bird needed moderate iteration, but the gap-centring fix (that clearing near the edge causes an immediate crash on the next flap) was not obvious and only became clear from watching the agent fail. Pong was the most straightforward and only needed a few iterations to balance the CPU. Snake needed the most iterations by far; suppressing the orbiting behaviour without breaking the learning gradient took a lot of work. In every game, the observation space was the second hardest thing to get right.

One big advantage of the DRL approach is the feedback loop: training is fully automated and logs to CSV, so comparing the effect of each change is easy. The

downside is training time, which runs from 30 minutes to 7.5 hours depending on the algorithm. Using the lighter algorithms (A2C or TRPO) makes iterating much more practical.

5.4.5 Overall Algorithm Comparison

Looking across all three games together, a clear ranking emerges. TRPO was the most consistent performer overall. It ranked first or joint-first in Snake, was the only algorithm to clear pipes on Flappy Hard, and had the best mean on Pong Hard at +9.6. It never had a catastrophic result on any game or difficulty. PPO was the most reliable all-round choice: it never collapsed on any difficulty, hit a perfect +10 on both Pong Easy and Normal, and performed solidly across Snake and Flappy. A2C was the best value by training time, finishing in 30 to 47 minutes across all three games and producing competitive results on Easy and Normal. Its Pong Hard result of -4.0 is the one place where it clearly broke down. RPPO was the weakest return on investment across the entire benchmark: the longest training times (4 to 7.5 hours) paired with the weakest evaluation scores in every game. Its LSTM architecture provided no benefit in environments where the agent already receives complete state information.

The strongest trend across all three games is that TRPO and PPO produce the most robust policies. They may not always converge the fastest or cheapest, but their results hold up across difficulty levels better than A2C or RPPO. A2C is the algorithm to use when iteration speed matters more than peak performance, such as early in development when the reward function is still being tuned. RPPO is hard to justify for this class of game given both its training cost and its results.

A second trend is the relationship between algorithm robustness and difficulty. On Easy, all four DRL algorithms perform well and the differences between them

are small. On Hard, the gaps open up significantly. The agents that learned general strategies on Normal generalise to Hard; the ones that overfit to the training conditions fall apart. This makes Hard difficulty the most useful setting for benchmarking how well an algorithm actually learned the task rather than just learned to pass the training configuration.

Chapter 6

Conclusions

This thesis evaluated the feasibility of deep reinforcement learning as a playtesting tool for small 2D games. We constructed a benchmark of three games (Flappy Bird, Pong, and Snake), trained four DRL algorithms (PPO, A2C, TRPO, RPPO) on each, and evaluated performance across three difficulty levels designed around a doubling-and-halving principle. The benchmark ran entirely on a consumer CPU without a discrete GPU. We addressed four research questions spanning feasibility, algorithm performance, generalization, and the developer experience.

RQ1: Feasibility

DRL-based playtesting is not readily feasible for small studios without a significant upfront investment. All 12 agent configurations were trained on a single consumer machine in a reasonable time, which confirms that the hardware barrier is low. The dominant engineering cost was the design and iteration of the observation space and reward function; each game required numerous training cycles before the agent behaved correctly. The dominant calendar cost was evaluation: Flappy Bird evalua-

tion alone required approximately 82 hours because well-trained agents consistently reached the 30-minute episode cap on Easy and Normal, and evaluation time cannot be shortened without discarding results. Snake and Pong each completed evaluation within a single working day. This asymmetry is a direct consequence of agent success; the better the policy, the longer each evaluation run takes. Snake alone required more reward engineering effort than the other two games combined. The workflow also requires familiarity with both game development and machine learning; there is no off-the-shelf component that removes either requirement. For a developer working on a single-release project, the investment is difficult to justify. For a live-service game with ongoing balance updates, the cost is more plausibly recovered over time.

RQ2: Algorithm Performance

Performance varied across games and algorithms, but a consistent ranking emerged. TRPO was the strongest overall, ranking first or joint-first in all three games and being the only algorithm to clear any pipes on Flappy Hard. PPO was the most reliable, never collapsing on any game or difficulty. A2C offered the best trade-off between training speed and performance on Easy and Normal difficulties, finishing in 30 to 47 minutes per game, but broke down on Pong Hard with a mean point differential of -4.0. RPPO was the weakest algorithm in every game despite requiring the longest training times (4 to 7.5 hours per game); its LSTM memory provided no benefit in environments that already supply complete state information.

The human baseline results differed by game. In Snake, every DRL algorithm substantially exceeded human performance at all three difficulties: the weakest DRL result was still nearly four times the human mean on Easy. In Flappy Bird, DRL agents ran indefinitely on Easy and Normal, while the human averaged 194.3 and

25.3 pipes, respectively, but both the human and three of four DRL algorithms scored zero on Hard. Pong was the exception: the human averaged above +8 on all three difficulties, outperforming A2C on Normal and Hard and remaining competitive with the strongest algorithms throughout. Pong is the only game in the benchmark where a human player remained competitive with the best DRL agents; the reactive ball-tracking it requires is a task humans perform naturally, whereas the long-horizon spatial planning required by Snake is not.

RQ3: Generalisation

Generalization to easier difficulties was reliable across all games: policies trained on Normal transferred without meaningful loss to Easy in every case. Hard difficulty was more informative. In Flappy Bird, three of four algorithms scored zero on Hard; only TRPO cleared any pipes, and TRPO's mean was still only 2.5. In Pong, three of four algorithms maintained strong performance on Hard, while A2C collapsed, indicating that A2C overfitted to the paddle size seen during training. Snake agents remained stable across all three levels, suggesting that speed within the tested range is not a limiting factor for trained policies.

These results support the use of difficulty variation as a lightweight evaluation tool. A single agent trained on Normal can be evaluated across a range of parameter configurations without retraining, and the resulting score profile identifies the point at which the trained policy breaks down. In Flappy Bird, this transition is sharp and precise: performance is flat and maxed out at 150 and 300-pixel gaps, then collapses to near zero at 75 pixels. In Snake, the flat profile across all three levels tells the developer that the acceleration parameter within this range does not meaningfully change how hard the game is for a trained agent, which is itself actionable design

feedback. This evaluation workflow scales naturally to many more configurations with no additional training cost.

RQ4: Developer Perspective

Reward function design was the dominant source of effort across all three games. In Snake, suppressing the orbiting behaviour required multiple shaping terms added in separate iterations, each requiring a full training run to evaluate. In Flappy Bird, the gap-centring issue was not apparent from training curves and only became clear from watching the agent play in real time. Pong required the fewest iterations overall, but still needed careful tuning of the CPU opponent's reaction delay to produce a useful learning signal.

The observation space was the second hardest component to get right in each game; the action space was straightforward in all three cases. The CSV training logs made it easy to compare the effect of each change. During the iteration phase, training time was the main constraint on how quickly a developer could test a change; using A2C or TRPO brings each cycle within a working day. Once a final configuration was ready, evaluation time became the dominant cost, and unlike training time it does not decrease by choosing a faster algorithm. Reward function iteration is not automatable and is captured in neither figure. A developer must watch the agent play, identify what is going wrong, and redesign the reward; this cycle can repeat many times before the agent behaves correctly.

Limitations

The benchmark covers three games, all of which are clones of classic arcade titles with relatively simple mechanics and a fully observable state. The results may not generalize to games with richer state spaces, larger action sets, or fundamentally different objectives. Each game was built and evaluated by a single developer, which limits the generalisability of the RQ4 conclusions; a different developer with different prior experience would likely face a different set of challenges and arrive at different design choices. The human baseline was collected from a single player and reflects that individual's skill level and play style, which may not represent the broader range of human performance on these games. All agents were trained using Stable-Baselines3 default hyperparameters without tuning; the reported results represent a floor rather than the maximum achievable by each algorithm, and tuned configurations would likely produce stronger performance.

Future Work

The most direct extension of this work is adding more games to the benchmark. A Donkey Kong clone was in active development during this project, but did not reach a state suitable for inclusion before the submission deadline. Donkey Kong would contribute a qualitatively different challenge: a multi-level platformer with climbing mechanics, falling hazards, and moving enemies, which would test whether the reward design patterns and generalization results observed here extend to a more complex environment.

Beyond additional games, several directions are worth pursuing. Systematic hyperparameter tuning for each algorithm on each game would establish whether the default configurations used here are close to optimal or whether tuned versions pro-

duce substantially different results. A human baseline collected from multiple players would give clearer insight into where DRL performance genuinely exceeds human play rather than reflecting one individual. Extending the difficulty evaluation to a continuous parameter sweep, varying the pipe gap or paddle size across many values rather than just three, would sharpen the performance profiles and more precisely identify where trained policies break down. Finally, applying the workflow to a game under active development, rather than a finished clone, would give a more realistic picture of how the iteration costs change when both the game and the agent are being modified simultaneously.

Bibliography

[1] Andersen, P.-A., Goodwin, M., and Granmo, O.-C. Deep RTS: A Game Environment for Deep Reinforcement Learning in Real-Time Strategy Games. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)* (Aug. 2018).

[2] Beeching, E., Debangoye, J., Simonin, O., and Wolf, C. Godot reinforcement learning agents, 2021.

[3] Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research 47* (June 2013), 253–279.

[4] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. Openai gym, 2016.

[5] Bycer, J. *Game Design Deep Dive: Platformers*, 1 ed. CRC Press, 2020.

[6] Cilan, A., and Özgövde, A. Learning controllable and diverse player behaviors in multi-agent environments, 2025.

[7] Cobbe, K., Hesse, C., Hilton, J., and Schulman, J. Leveraging procedural generation to benchmark reinforcement learning. In *Proceedings of the 37th*

International Conference on Machine Learning (2020), vol. 119 of *Proceedings of Machine Learning Research*, PMLR.

[8] Ferdous, R., Kifetew, F., Prandi, D., and Susi, A. Curiosity driven multi-agent reinforcement learning for 3d game testing. In *2025 IEEE/ACM International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* (2025).

[9] Gillberg, J., Bergdahl, J., Sestini, A., Eakins, A., and Gisslén, L. Technical challenges of deploying reinforcement learning agents for game testing in AAA games. In *2023 IEEE Conference on Games (CoG)* (2023).

[10] Gomes, G., Vidal, C. A., Cavalcante-Neto, J. B., and Nogueira, Y. L. A modeling environment for reinforcement learning in games. *Entertainment Computing 43* (Aug. 2022).

[11] Gomes, G., Vidal, C. A., Cavalcante-Neto, J. B., and Nogueira, Y. L. B. AI4U: A Tool for Game Reinforcement Learning Experiments. In *2020 19th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)* (Nov. 2020).

[12] Gulcehre, C., Wang, Z., Novikov, A., Paine, T. L., Colmenarejo, S. G., Zolna, K., Agarwal, R., Merel, J., Mankowitz, D., Paduraru, C., Dulac-Arnold, G., Li, J., Norouzi, M., Hoffman, M., Nachum, O., Tucker, G., Heess, N., and de Freitas, N. RL Unplugged: A Suite of Benchmarks for Offline Reinforcement Learning.

[13] Hausknecht, M., and Stone, P. Deep recurrent q-learning for partially observable mdps. In *2015 AAAI Fall Symposium Series* (2015).

[14] Karakovskiy, S., and Togelius, J. The mario AI benchmark and competitions. *IEEE Transactions on Computational Intelligence and AI in Games 4*, 1 (2012).

[15] Kempka, M., Wydmuch, M., Runc, G., Toczek, J., and Jaśkowski, W. ViZDoom: A Doom-based AI research platform for visual reinforcement learning. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)* (Sept. 2016).

[16] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning, 2016.

[17] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. Human-level control through deep reinforcement learning. *Nature 518*, 7540 (2015), 529–533.

[18] Nichol, A., Pfau, V., Hesse, C., Klimov, O., and Schulman, J. Gotta learn fast: A new benchmark for generalization in rl, 2018.

[19] Pleines, M., Pallasch, M., Zimmer, F., and Preuss, M. Generalization, mayhems and limits in recurrent proximal policy optimization, 2022.

[20] Politowski, C., Petrillo, F., and Guéhéneuc, Y. A survey of video game testing. In *2nd IEEE/ACM International Conference on Automation of Software Test, AST@ICSE 2021, Madrid, Spain, May 20-21, 2021* (2021), IEEE.

[21] Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., and Dormann, N. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research 22*, 268 (2021).

[22] Rani, G., Pandey, U., Wagde, A. A., and Dhaka, V. S. A deep reinforcement learning technique for bug detection in video games. *International Journal of Information Technology 15*, 1 (Jan. 2023).

[23] Romac, C., Portelas, R., Hofmann, K., and Oudeyer, P.-Y. Teachmyagent: a benchmark for automatic curriculum learning in deep RL. In *Proceedings of the 38th International Conference on Machine Learning* (2021), vol. 139 of *Proceedings of Machine Learning Research*, PMLR.

[24] Rupp, F., Eberhardinger, M., and Eckert, K. Simulation-driven balancing of competitive game levels with reinforcement learning. *IEEE Transactions on Games 16*, 4 (2024).

[25] Sapio, F., and Ratini, R. Developing and Testing a New Reinforcement Learning Toolkit with Unreal Engine. In *Artificial Intelligence in HCI*, H. Degen and S. Ntoa, Eds., vol. 13336. Springer International Publishing, Cham, 2022.

[26] Schulman, J., Levine, S., Moritz, P., Jordan, M. I., and Abbeel, P. Trust region policy optimization, 2015.

[27] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms, 2017.

[28] Togelius, J., Karakovskiy, S., and Baumgarten, R. The 2009 mario AI competition. In *IEEE Congress on Evolutionary Computation (CEC 2010)* (July 2010).

[29] Towers, M., Kwiatkowski, A., Terry, J., Balis, J. U., De Cola, G., Deleu, T., Goulão, M., Kallinteris, A., Krimmel, M., KG, A., Perez-Vicente, R., Pierré, A., Schulhoff, S., Tai, J. J., Tan, H., and Younis, O. G. Gymnasium: A standard interface for reinforcement learning environments, 2024.