Optimizing Rational and Aesthetic Navigation Objectives via Stochastic Reward Shaping in Procedural 3D Unity Environments

Victor Tenneroni

December 5, 2025

1 Introduction

1.1 Problem Statement

This project addresses autonomous navigation across procedurally generated parkour environments where agents must balance multiple competing objectives: speed (reaching targets efficiently), energy management (stamina conservation), and aesthetic quality (stylistic movement). The agent must reach the target through human-preferred behaviors such as dynamic rolls and varied movement patterns.

This raises two fundamental questions. First, how do we train an AI to understand style? Style is inherently subjective—what one human finds aesthetically pleasing, another might not. In this work, we explore this question in the context of acrobatic parkour, where style manifests through dynamic rolls and varied movement patterns. Second, how do you integrate human preferences into RL when human reaction time is orders of magnitude slower than agent training time?

Reinforcement learning is necessary here for several reasons. The problem involves a high-dimensional state space (14 observations) and a complex action space (5 discrete actions). The randomized environment generates infinite variations through procedural platform generation, requiring the agent to generalize across variations instead of memorizing fixed sequences. The agent must make strategic tradeoffs between speed and stamina conservation, balancing immediate rewards against future resource availability. There is no closed-form solution for the combined dynamics of stamina management, randomized platform layouts, and aesthetic preference modeling.

Traditional approaches fail under these conditions. Rule-based systems cannot handle the randomization inherent in procedural generation. PID control lacks the strategic resource management needed for stamina optimization across varying platform configurations. Fixed environments would allow the agent to memorize sequences, defeating the goal of generalization.

We build both the RL agent and the environment simultaneously in Unity. This creates a moving target problem where environment changes during development break previously trained agents. The randomized environment (procedural platform generation with varying gaps, heights, and widths) presents a constantly changing training distribution that the agent must generalize across.

1.2 The Human Feedback Challenge

Reinforcement Learning from Human Feedback (RLHF) addresses preference learning by having humans directly label preferred trajectories during training. This approach captures nuanced aesthetic judgments that are difficult to encode in reward functions. Our training infrastructure operates

28 parallel agents at 20× time acceleration, generating ∼1,054 steps/second and approximately 30 complete episodes per minute.

As a first step toward preference learning under accelerated training constraints, we propose episodic stochastic reward modulation. We inject randomness at the episode level: 40% of training episodes provide enhanced rewards (+1.5 bonus) for high-cost stylistic actions (rolls), while the remaining 60% offer only base rewards (+0.5). This approach models the variance in human aesthetic preferences without requiring real-time feedback.

By creating episodes where certain behaviors are disproportionately rewarded, we force the agent to learn those behaviors remain viable strategies, preventing complete dismissal of high-cost actions that humans would find preferable.

1.3 Empirical Validation

Across multiple training configurations (2M steps each), we observe:

- Baseline (15% style frequency): 0.69% roll usage, +67.90 final reward
- Stochastic reward shaping (40% style frequency): 7.81% roll usage, +89.18 final reward
- Roll usage increased 11.3× (0.69% to 7.81%), with 239 rolls per episode on average
- Final performance: +89.18 average reward, 555.91 units mean distance traveled (range 29.89–603.56 units)

2 Background & Related Work

2.1 Reinforcement Learning from Human Feedback

Reinforcement Learning from Human Feedback (RLHF) (Christiano et al., 2017) addresses the fundamental challenge of communicating complex goals to RL systems when reward functions are difficult to specify. The approach learns a reward function from human preferences over trajectory segments, enabling agents to solve tasks without access to the true reward function.

Core Method:

RLHF maintains a policy $\pi : O \to A$ and a reward function estimate $\hat{r} : O \times A \to \mathbb{R}$, updated through three asynchronous processes:

1. **Policy Optimization:** The policy interacts with the environment, producing trajectories. Policy parameters are updated using standard RL algorithms (e.g., A2C, TRPO) to maximize predicted rewards $\hat{r}(o_t, a_t)$.
2. **Preference Elicitation:** Pairs of trajectory segments (σ^1, σ^2) are selected and presented to a human for comparison. The human indicates preference, equality, or inability to compare.
3. **Reward Function Fitting:** The reward function $\hat{r}$ is optimized via supervised learning to fit human comparisons using the Bradley-Terry model:

$$P[\sigma^1 \succ \sigma^2] = \frac{\exp(\sum_t \hat{r}(o_t^1, a_t^1))}{\exp(\sum_t \hat{r}(o_t^1, a_t^1)) + \exp(\sum_t \hat{r}(o_t^2, a_t^2))}$$

The reward function is optimized to minimize cross-entropy loss:

$$\text{loss}(\hat{r}) = -\sum_{(\sigma^1,\sigma^2,\mu)\in D} \left[\mu(1)\log P[\sigma^1 \succ \sigma^2] + \mu(2)\log P[\sigma^2 \succ \sigma^1]\right]$$

where D is the database of human comparisons and μ is the distribution over preferences.
Key Findings:

- **Efficiency:** RLHF reduces human feedback requirements by ∼3 orders of magnitude, requiring feedback on less than 1% of agent interactions
- **Performance:** With 700–5,500 human comparisons (15 minutes to 5 hours of human time), RLHF can solve complex RL tasks including Atari games and simulated robot locomotion, matching or exceeding performance of RL with true reward functions
- **Novel Behaviors:** Can learn complex novel behaviors (e.g., backflips, one-legged locomotion) from ∼1 hour of human feedback, even when no reward function can be hand-engineered
- **Online Feedback Critical:** Offline reward predictor training fails due to nonstationarity; human feedback must be intertwined with RL learning to prevent exploitation of learned reward function weaknesses

Limitations for Accelerated Training:
RLHF requires real-time human feedback during training, which becomes infeasible when:

- Training runs at 20× time acceleration (environment runs too fast for human perception)
- Training generates ∼1,054 steps/second across 28 parallel agents
- Episodes complete in ∼30 seconds (wall-clock time), requiring human evaluation every few seconds

This fundamental incompatibility motivates our approach: **offline preference approximation** through stochastic reward shaping, which models human preference variance without requiring real-time feedback.

2.2 Reward Shaping in Reinforcement Learning

Reward shaping modifies the reward function to guide learning while preserving optimal policies (Ng et al., 1999). Our work extends this concept by introducing **episodic stochastic reward modulation**, where reward structure varies probabilistically across episodes to approximate preference diversity.

3 Methodology

3.1 Training Infrastructure

3.1.1 Parallel Agent Setup

The training infrastructure employs **28 parallel agents** running simultaneously across 28 independent `TrainingArea` objects within a single Unity environment instance. This parallelization strategy enables efficient data collection and significantly accelerates the training process.

Key Configuration:

- **Number of Agents:** 28 (one agent per `TrainingArea`)
- **Number of Environments:** 1 (single Unity instance)
- **Training Areas:** 28 `TrainingArea` objects in the scene
- **Time Scale:** 20× acceleration multiplier during training

Design Rationale: The choice of 28 agents represents a balance between:

1. **Data Collection Efficiency:** More agents provide more diverse experiences per unit time
2. **Computational Overhead:** Each agent requires physics simulation and observation collection
3. **Memory Constraints:** Unity scene complexity increases with more agents
4. **Practical Limits:** 28 agents was empirically determined to be the maximum stable configuration on the target hardware

3.1.2 Training Environment Configuration

Each of the 28 agents trains in an independent `TrainingArea` with procedurally generated platforms. At the start of each episode, platforms are regenerated with randomized parameters, ensuring all 28 agents experience different environments simultaneously.

Platform Generation Parameters:

- **Platform Count:** 20 platforms per episode
- **Gap Range:** 2.5–4.5 units (edge-to-edge, randomized per gap)
- **Platform Width Range:** 20–84 units (randomized per platform)
  - Base width: 20–28 units (randomized)
  - 80% of platforms are 3× longer: 60–84 units
  - 20% of platforms remain base size: 20–28 units
- **Height Variation:** -0.6 to +1.2 units change between consecutive platforms
- **Absolute Height Range:** -0.5 to 5.0 units

Randomization Strategy: Platforms are regenerated at the start of each episode for each agent independently. This means all 28 agents train on different platform layouts simultaneously, maximizing diversity in the collected experience. The randomization prevents memorization and forces the agent to use perception (raycasts) instead of sequence memory.

3.1.3 Training Duration and Scale

Training Scale:

- **Total Training Steps:** 2,000,000 steps
- **Wall-Clock Time:** ∼30–32 minutes (approximately 31.6 minutes for complete training)

- **Time Horizon:** 128 steps before value bootstrapping
- **Checkpoint Interval:** Every 500,000 steps

Training Efficiency Calculation:

- **Single Agent:** ∼8+ hours for 2M steps (at 20× time scale)
- **28 Parallel Agents:** ∼30 minutes for 2M steps
- **Speedup Factor:** ∼16× improvement (28 agents × 20× time scale = 560× theoretical, but limited by Python training loop and network updates)

3.1.4 Time Acceleration and Offline Preference Modeling

Time Acceleration Necessity: The 20× time acceleration factor is essential for practical training durations. However, this acceleration creates a fundamental constraint: **human feedback is incompatible with accelerated training**. Real-time human evaluation of agent behavior becomes infeasible when the environment runs 20× faster than normal speed.

The Fundamental Constraint:

- **Normal Speed:** Human can evaluate behavior in real-time
- **20× Accelerated:** Environment runs too fast for human perception and reaction
- **Implication:** Cannot use RLHF (Reinforcement Learning from Human Feedback) during training
- **Solution:** Design reward functions that approximate human preferences *a priori*

Implication for Reward Design: This constraint necessitates an **offline preference modeling approach**. Since real-time human feedback during training is infeasible (as in RLHF), we design reward functions that approximate human preferences *a priori*. The style reward system (Section 3.2) represents our approach to approximating human aesthetic preferences without requiring real-time human interaction.

3.1.5 PPO Algorithm and Training Strategy

Algorithm: Proximal Policy Optimization (PPO)

The training uses PPO with high exploration (beta = 0.1) and linear decay schedules to gradually shift from exploration to exploitation. The network architecture uses separate actor-critic networks: policy network (2×256 hidden layers → 5 action logits) and value network (2×128 hidden layers → 1 value estimate). The separate architecture enables independent learning rates for policy and value estimation, with advantage estimation providing credit assignment by separating "good action" from "already good state." Detailed hyperparameters, network architecture, and advantage estimation are provided in Section 4.2.

Why PPO Over Q-Learning/DQN?

PPO was selected over Q-learning/DQN for several reasons specific to this problem domain:

1. Continuous Observations with Discrete Actions: PPO handles the combination of continuous state space (14-dimensional observations) and discrete action space (5 actions) naturally. Q-learning/DQN struggles with continuous state spaces, requiring discretization or function

approximation that loses information. PPO's policy gradient approach directly models the action distribution over continuous observations.

2. Policy Stability with Clipping: PPO's clipping mechanism (epsilon = 0.2) prevents catastrophic policy updates that could cause the agent to "forget" previously learned behaviors. This is critical when training on a constantly changing randomized environment where the agent must maintain stable strategies across platform variations. Q-learning's deterministic action selection (always picks best Q-value) lacks this stability mechanism.

3. High Exploration Capability: The high entropy coefficient ($\beta = 0.1$, 6.7× higher than ML-Agents default of 0.015) enables sufficient exploration in the complex action space. This was empirically validated: increasing exploration led to a 603% reward improvement as the agent discovered roll usage patterns. Q-learning's ε-greedy exploration is less effective for discovering high-cost, high-reward actions like rolls that require strategic timing.

4. Separate Value Network: PPO uses independent actor (policy) and critic (value) networks, allowing policy and value estimation to learn at different rates. This separation is more stable than shared networks when rewards are sparse (target reach +10.0, fall -1.0), as the value network can learn long-term returns while the policy network focuses on action selection.

5. Episodic Style Bonus Compatibility: The episodic stochastic reward structure (40% of episodes with +1.5 roll bonus) requires an algorithm that can learn from variable reward distributions. PPO's on-policy learning naturally handles this episodic variation, while Q-learning's off-policy nature would struggle to adapt to episode-level reward changes.

6. Stable Learning with Sparse Rewards: The reward structure includes sparse terminal rewards (target reach, fall penalty) that occur infrequently. PPO's advantage estimation (GAE with $\lambda = 0.95$) effectively propagates these sparse signals, while Q-learning requires careful reward shaping to learn from sparse signals.

3.2 Reward Design

3.2.1 Design Philosophy and Workflow

Design Philosophy: The reward function must guide the agent toward both functional parkour (reaching targets efficiently) and aesthetic parkour (stylish movements). This dual objective creates a multi-objective optimization problem that requires careful reward shaping.

Key Design Principles:

- **Dense Rewards:** Provide learning signal at every step (progress, grounded)
- **Sparse Rewards:** Provide clear success/failure signals (target reach, fall)
- **Shaped Rewards:** Guide agent toward desired behaviors (style bonuses)
- **Magnitude Relationships:** Ensure rewards are properly scaled relative to each other

3.2.2 Multi-Objective Reward Structure

The reward function combines multiple objectives to guide the agent toward both functional and aesthetic parkour behavior:

1. **Progress Maximization** (Primary objective) — 79% of total reward
2. **Time Minimization** (Secondary objective) — Encourages speed

3. **Stamina Management** (Tertiary objective) — Encourages efficiency

4. **Style Actions** (Episodic bonus) — Encourages aesthetic behavior

3.2.3 Base Rewards: Design and Calibration

Dense Rewards (Per-Step):

Reward Component	Value	Condition	Design Rationale
Progress Reward	$+0.1 \times \Delta x$	Forward movement	Primary learning signal. 0.1/unit chosen to provide strong gradient while maintaining scale.
Grounded Reward	+0.001	Agent is grounded	Encourages staying on platforms. Small magnitude (0.1% of progress) prevents over-prioritization.
Time Penalty	−0.001	Per fixed update	Encourages speed. Magnitude matches grounded reward to balance.
Low Stamina Penalty	−0.002	Stamina $< 20\%$	Discourages keeping stamina at zero. 2× time penalty to emphasize importance.

Table 1: Dense reward components

Sparse Rewards (Episode-Level):

Reward Component	Value	Condition	Design Rationale
Target Reach	+10.0	Distance < 2.0 units	Clear success signal. Equivalent to 100 units of progress.
Fall Penalty	−1.0	Agent falls or timeout	Clear failure signal. Magnitude chosen to be significant but not overwhelming (10% of target reach).

Table 2: Sparse reward components

3.2.4 Reward Scaling and Context

Target Definition and Success Condition:

The target position is calculated dynamically based on the procedurally generated platform layout:

- **Target X Position:** targetX = lastPlatformEndX + targetOffset
  - lastPlatformEndX = right edge of the 20th (last) platform
  - targetOffset = 5.0 units (target is positioned 5 units beyond the last platform)

- **Target Y Position:** Matches agent spawn height (ensures target is at agent level)
- **Success Condition:** $|\text{agent.x} - \text{target.x}| < 2.0$ units (X-axis distance only, not 3D distance)
  - Uses X-axis only to avoid issues when agent passes target at different Y height
  - When reached: episode ends immediately with `EndEpisode()`

Target Reward:

- targetReachReward = +10.0 (one-time, sparse reward given only when target is reached)
- This is a sparse reward—only given once per episode when successful
- Represents ∼11% of total episode reward in successful episodes

Typical Episode Reward Breakdown:
For a successful episode reaching the target (∼700 units of progress, ∼850 steps):

- **Progress Reward:** ∼70.0 (79% of total) — $700 \times 0.1 = +70.0$
  - Primary learning signal: most reward comes from progress, not target reach
- **Target Reach:** +10.0 (11% of total)
  - Sparse success signal: serves as the success condition, but progress reward is the primary learning signal
- **Grounded Reward:** ∼0.85 (1% of total) — $850 \times 0.001 = +0.85$
- **Time Penalty:** ∼-0.85 (-1% of total) — $850 \times -0.001 = -0.85$
- **Roll Rewards:** Variable
  - Base: +0.5 per roll (always given)
  - Style: +1.5 per roll (40% of episodes)
  - Typical: ∼239 rolls/episode ×0.5 = +119.5 base
  - In style episodes: additional +358.5 from style bonuses
- **Low Stamina Penalty:** Variable — −0.002 per step when stamina < 20%
- **Total Episode Reward:** ∼80.0 (typical successful episode, matches mean of 80.06)

Reward Range Interpretation:
The observed reward range (3.05–88.82) reflects episode outcomes:

- **Minimum (3.05):** Episodes that fail early (timeout/fall) — minimal progress reward, no target reach reward
- **Maximum (88.82):** Successful episodes with optimal behavior — full progress reward + target reach + efficient action usage
- **Mean (80.06):** Represents the typical successful episode reward breakdown above

3.2.5 Iterative Reward Calibration: Design Evolution

The reward structure evolved through iterative problem-solving, addressing emergent behaviors that deviated from desired parkour style:

Problem 1: Sprint Bashing

- **Observed Behavior:** Agent learned to hold sprint 38% of the time, keeping stamina at zero
- **Root Cause:** No penalty for depleting stamina; sprint provided speed advantage with no downside
- **Fix:** Added low stamina penalty (-0.002 per step when stamina $< 20\%$) and reduced sprint consumption rate from 33.33/sec to 20/sec
- **Result:** Agent learned to manage stamina strategically instead of depleting it completely

Problem 2: Roll Ignored

- **Observed Behavior:** Roll usage remained at 0.69% despite being the fastest action (18 units/sec)
- **Root Cause:** Roll cost was too high (150 stamina = 7.5 seconds to regenerate at 20/sec regen rate)
- **Fix:** Reduced roll cost from 150 to 60 stamina (2 seconds to regenerate at 30/sec regen rate)
- **Result:** Roll became more accessible, but usage remained low

Problem 3: Still No Rolls

- **Observed Behavior:** Even with lower cost (60 stamina), agent rarely used rolls
- **Root Cause:** No positive incentive; roll was merely "not bad" but provided no reward signal
- **Fix:** Added base roll reward (+0.5 always given) so rolls are never "bad" actions
- **Result:** Roll usage increased slightly, but still insufficient

Final Breakthrough: Dual Reward Structure

- **Solution:** Dual reward structure combining base reward (+0.5 always) with episodic style bonus (+1.5 in 40% of episodes)
- **Rationale:** Base reward ensures rolls are always valuable, while style bonus creates strategic variety and prevents complete dismissal of high-cost actions
- **Result:** 31% reward improvement ($+67.90 \rightarrow +89.18$), rolls used strategically (7.81% usage, 239 rolls/episode average)

This iterative process demonstrates the importance of empirical observation and reward calibration in RL systems, where theoretical reward design often requires refinement based on emergent agent behavior.

3.2.6 Style Reward Approximation: Design Process

Stochastic Reward Shaping as Preference Approximation:

The style reward system approximates human aesthetic preferences through **stochastic reward injection** instead of real-time human feedback. This design addresses the fundamental constraint that human feedback is incompatible with accelerated training (Section 3.1). The final dual reward structure (base + style bonus) emerged from the iterative calibration process described above.

Roll Reward Structure:

Reward Component	Value	Condition	Design Rationale
Roll Base Reward	+0.5	Roll action executed	Ensures rolls are always valuable. Prevents agent from ignoring rolls in non-style episodes.
Roll Style Bonus	+1.5	Roll in style episode	Provides additional incentive in 40% of episodes. Creates behavioral variety.

Table 3: Roll reward structure

Total Roll Reward:

- **In style episodes (40%):** +0.5 base +1.5 style = +2.0 per roll (20× progress per unit)
- **In non-style episodes (60%):** +0.5 base per roll (5× progress per unit)

Episode-Level Style Flag:

- **Probability:** 40% (`styleEpisodeFrequency = 0.4`)
- **Assignment:** Randomly determined at episode start
- **Scope:** Affects all roll actions within that episode
- **Rationale:** Stochastic injection mimics preference diversity across different human evaluators

3.2.7 Rationale: Stochastic Injection Mimics Preference Diversity

Why Stochastic Episode-Level Flags?

The episode-level style flag approximates preference diversity across different human evaluators. The stochastic assignment (40% probability) replaces a fixed reward structure, creating behavioral variety that mimics how different humans might value style vs. efficiency differently.

Why Base Reward + Style Bonus?

- **Base reward (always given):** Ensures rolls are always valuable, not just in style episodes. This prevents the agent from completely ignoring rolls in non-style episodes.
- **Style bonus (conditional):** Provides additional incentive in 40% of episodes, creating behavioral variety and encouraging occasional stylish movement.

3.3 MDP Formulation

The parkour navigation problem is formalized as a Markov Decision Process (MDP) defined by the tuple (S, A, R, P, γ):

State Space ($S \subseteq \mathbb{R}^{14}$): The fully observable state space consists of 14 continuous values encoding agent position, velocity, environment perception, and internal state (detailed in Section 3.3.2).

Action Space ($A = \{0, 1, 2, 3, 4\}$): Discrete action space with 5 actions: Idle (0), Jump (1), Jog (2), Sprint (3), Roll (4). Actions are subject to constraints based on stamina, grounded state, and cooldowns (detailed in Section 3.3.4).

Reward Function ($R : S \times A \times S' \to \mathbb{R}$): The reward function combines dense per-step rewards, sparse terminal rewards, and episodic style bonuses:

- **Dense rewards:** Progress ($+0.1 \times \Delta x$), grounded ($+0.001$), time penalty (-0.001), low stamina penalty (-0.002)
- **Sparse rewards:** Target reach ($+10.0$), fall/timeout (-1.0)
- **Style rewards:** Roll base ($+0.5$ always), roll style bonus ($+1.5$ in 40% of episodes)

Transition Dynamics ($P(s'|s, a)$): State transitions are governed by deterministic physics and stochastic environmental elements:

- **Deterministic physics:** Position updates via $p' = p + v\Delta t$, gravity, and stamina dynamics
- **Stochastic elements:** Platform randomization (gaps 2.5–4.5 units, widths 20–84 units) regenerated at each episode start; style flag assignment (40% probability) determined at episode start

Discount Factor ($\gamma = 0.99$): High discount factor emphasizes long-term rewards, appropriate for episodes lasting $\sim$100 seconds with strategic stamina management requirements.

3.4 State/Action Space

3.4.1 State Space Design Philosophy

Design Goals:

1. **Sufficient Information:** Agent must have enough information to make good decisions
2. **Minimal Dimensionality:** Smaller state space = faster learning
3. **Generalization:** State space must work across different platform layouts
4. **Interpretability:** State components should have clear semantic meaning

What We Exclude Matters: The state space deliberately excludes information that would hinder generalization:

- **No absolute position:** Since platforms randomize each episode, absolute coordinates are meaningless. The agent observes relative target position instead.
- **No action history:** The current state (velocity, stamina, raycasts) contains all necessary information for decision-making. Adding action history would increase dimensionality without providing additional signal.
- **No platform sequence memory:** The agent must use perception (raycasts) instead of memorizing platform patterns, forcing generalization across infinite environment variations.

3.4.2 State Space (Observations)

Total Observations: 14 floats

The state space is fully observable and consists of the following components:

Observation Component	Size	Description	Range/Normalization
Target Relative Position	3 floats	$(target.position - agent.position)$	Raw 3D vector (units)
Velocity	3 floats	$controller.velocity$	Raw 3D vector (units/sec)
Grounded State	1 float	1.0 if grounded, 0.0 if not	Binary (0.0 or 1.0)
Platform Raycasts	5 floats	Downward raycasts at [2, 4, 6, 8, 10] units ahead	Normalized (0.0–1.0)
Obstacle Distance	1 float	Forward obstacle raycast distance	Normalized (0.0–1.0)
Stamina	1 float	$currentStamina/maxStamina$	Normalized (0.0–1.0)

Table 4: State space components

State Space Properties:

- **Dimensionality:** 14 ($S \subseteq \mathbb{R}^{14}$)
- **Observability:** Fully observable (no hidden information)
- **Normalization:** Applied where applicable (raycasts, stamina)
- **Completeness:** Contains all information needed for parkour decisions

3.4.3 Platform Detection Raycasts: Critical Design Decision

Purpose: Detect gaps and platform edges ahead of the agent to enable gap detection and jump timing.

Implementation Details:

- **5 downward raycasts** at forward distances: $[2, 4, 6, 8, 10]$ units ahead
- **Ray origin:** $agent.position + forward \times distance + Vector3.up \times 0.5$
- **Ray direction:** $Vector3.down$
- **Max ray distance:** 10 (normalization factor)
- **Output encoding:**
  - Platform detected: $hit.distance/maxRayDist$ (0.0–1.0, where 0.0 = platform at ray origin)
  - No platform (gap): 1.0 (normalized max distance)

Critical Design: Perception for Generalization
Empirical Evidence:

- **Experiment:** test_v9 (no raycasts) vs. test_v10 (5 raycasts) in randomized environment
- **Result:** +3.43 vs. +9.85 reward (187% improvement, ∼60% performance drop without raycasts)
- **Interpretation:** Without raycasts, agent cannot adapt to randomized gap spacing (2.5–4.5 units)
- **Conclusion:** Platform raycasts are **essential** for generalization to randomized environments

Critical Insight: Raycasts enable the agent to "see ahead" and detect gaps dynamically. Without them, the agent attempts to memorize platform patterns, which fails catastrophically when platforms are randomized each episode. The 60% performance drop demonstrates that perception-based state representation is non-negotiable for procedural environments.

3.4.4 Action Space Design

Type: Discrete, single branch, 5 actions

Action	ID	Description	Speed	Stamina Cost	Constraints
Idle	0	No movement	0 units/sec	0	Always available
Jump	1	Vertical jump with forward boost	Instant	20 per jump	Requires: $isGrounded \wedge stamina \geq 20.0$
Jog	2	Forward movement	6 units/sec	0	Always available
Sprint	3	Forward movement	12 units/sec	20/sec	Requires: $stamina > 0 \wedge \neg cooldown$ (cooldown: 0.5s)
Roll	4	Forward roll	18 units/sec	60 per roll	Requires: $stamina \geq 60.0 \wedge \neg isRolling$ (duration: 0.6s)

Table 5: Action space

Action Space Properties:

- **Type:** Discrete ($A = \{0, 1, 2, 3, 4\}$)
- **Branch Count:** 1 (single decision branch)
- **Action Count:** 5

- **Constraints:** Enforced by environment (stamina, cooldown, grounded state)

Action Timing and Constraints:

- **Sprint Cooldown:** 0.5 seconds after sprint ends before sprint can be used again
- **Roll Duration:** 0.6 seconds (roll is a timed action, cannot chain rolls)
- **Stamina System:** Max stamina 100.0, regeneration 30.0/sec when not sprinting/jumping/rolling

Risk/Reward Trade-off: Roll Action Roll is the fastest action (18 units/sec, 1.5× sprint speed) but carries the highest stamina cost (60 per roll, 3× jump cost). This creates a strategic decision: the agent must balance speed gains against stamina depletion. The high cost prevents indiscriminate roll usage while the speed advantage rewards strategic timing (e.g., crossing gaps efficiently). This risk/reward structure naturally emerges from the action design instead of being explicitly encoded in rewards.

4 Implementation

4.1 Unity ML-Agents Setup

4.1.1 Environment Configuration

The training environment is built using **Unity 2022.3 LTS** with the **ML-Agents Toolkit (version 1.1.0)**. The implementation follows the standard ML-Agents architecture with custom extensions for parkour-specific behaviors.

Core Components:

- **Agent Script:** `ParkourAgent.cs` — Inherits from `Unity.MLAgents.Agent`
- **Training Areas:** 28 `TrainingArea` objects in the scene (one per parallel agent)
- **Character Controller:** Unity's built-in `CharacterController` component for physics-based movement
- **Configuration System:** `CharacterConfig` ScriptableObject for centralized parameter management

ML-Agents Integration:

- **Package Version:** `com.unity.ml-agents` 3.0.0+ (Unity Package Manager)
- **Python Package:** `mlagents` 1.1.0 (via conda/pip)
- **Communication:** Unity ↔ Python via gRPC on port 5004 (default)
- **Behavior Name:** `ParkourRunner` (must match in config and Unity)

4.1.2 Training Workflow: Detailed Process

Training Command:

```
cd src
conda activate mlagents
python train_with_progress.py parkour_config.yaml --run-id=training_<
    timestamp> --force
```

Step-by-Step Training Process:
Phase 1: Initialization (0–5 seconds)

1. **Python Script Execution:**
   - `train_with_progress.py` reads `parkour_config.yaml`
   - Parses max_steps (2,000,000) and behavior name (`ParkourRunner`)
   - Auto-generates run-id: `training_YYYYMMDD_HHMMSS`
   - Launches `mlagents-learn` subprocess with config file
2. **ML-Agents Trainer Startup:**
   - Python trainer initializes PyTorch model (actor + critic networks)
   - Opens gRPC server on port 5004
   - Waits for Unity connection
3. **Unity Editor Connection:**
   - User opens `TrainingScene.unity` in Unity Editor
   - Scene contains 28 `TrainingArea` objects, each with a `ParkourAgent`
   - User presses **Play** button in Unity Editor
   - Unity ML-Agents connects to Python trainer on port 5004

Phase 2: Training Loop (30 minutes)

4. **Experience Collection:**
   - Each of 28 agents collects experiences simultaneously
   - At 50Hz (20ms per step), each agent:
     - Collects observations (14 floats)
     - Receives action from policy network
     - Executes action (with constraints)
     - Calculates rewards
     - Stores experience tuple: $(state, action, reward, next\_state)$
5. **Batch Processing:**
   - When buffer reaches `time_horizon` (128 steps) $\times$ 28 agents = 3,584 experiences:
     - Trainer samples `batch_size` (1024) experiences
     - Computes advantages using GAE ($\lambda = 0.95$, $\gamma = 0.99$)
     - Trains policy network for `num_epoch` (5) epochs

- Updates value network (critic)
- Clears buffer, continues collection

6. **Progress Tracking:**
   - `train_with_progress.py` intercepts ML-Agents output
   - Parses step count from log lines: `[INFO] ParkourRunner. Step: 680000.`
   - Calculates percentage: $(current\_steps/max\_steps) \times 100$
   - Displays: `[34.0%] Time Elapsed: 735.333 s. Mean Reward: 9.899.`
7. **Checkpointing:**
   - Every 500,000 steps: Model saved to `src/results/training_*/ParkourRunner.onnx`
   - Summary logs saved to `src/results/training_*/run_logs/`
   - TensorBoard logs updated for visualization

Phase 3: Completion (2M steps)

8. **Training Completion:**
   - Final model saved at 2,000,000 steps
   - Training statistics written to `timers.json` and `training_status.json`
   - Python script exits
   - Unity Editor can be stopped

4.2 Training Hyperparameters

4.2.1 PPO Configuration

The training uses Proximal Policy Optimization (PPO) with the following hyperparameters defined in `parkour_config.yaml`:

Hyperparameters:

- **Learning Rate:** 3.0×10^{-4} (linear decay schedule)
- **Batch Size:** 1024 experiences per training batch
- **Buffer Size:** 10240 (10× batch size for experience replay)
- **Beta (Entropy):** 0.1 (linear decay) — High exploration coefficient
- **Epsilon (Clipping):** 0.2 (linear decay) — PPO clipping parameter
- **Lambda (GAE):** 0.95 — Generalized Advantage Estimation lambda
- **Gamma (Discount):** 0.99 — Discount factor for future rewards
- **Num Epochs:** 5 — Training epochs per batch
- **Time Horizon:** 128 steps before value bootstrapping

Network Architecture:

- **Actor Network:** 2 hidden layers $\times$ 256 units $\rightarrow$ 5 action logits, input normalization enabled
- **Critic Network:** 2 hidden layers $\times$ 128 units $\rightarrow$ 1 value estimate, separate from actor (not shared)
- **Activation:** ReLU (default ML-Agents)
- **Initialization:** Xavier/Glorot uniform (ML-Agents default)

Actor-Critic Architecture and Advantage Estimation:
The actor-critic architecture uses two separate networks with distinct roles:

- **Policy Network (Actor):** Takes the 14-dimensional state and outputs a probability distribution over 5 discrete actions. The network learns which actions to take in each state.
- **Value Network (Critic):** Takes the same state and outputs a scalar value estimate representing the expected future return from that state. The network learns to evaluate how "good" a state is.

Key Insight: Advantage Estimation for Credit Assignment
The critical mechanism is **advantage estimation**, which separates "good action" from "already good state":

1. **Value network evaluates state:** "You're in a good situation worth 45 reward"
2. **Agent takes action and receives reward:** Actual return is 50
3. **Advantage calculation:** $A(s, a) = Q(s, a) - V(s) = 50 - 45 = +5$
4. **Policy update:** Reinforce the action because it performed better than expected

This separation is crucial for credit assignment. Without it, the agent might attribute high rewards to being in a good state instead of taking a good action. The advantage signal tells the policy network: "This action was better than what the value network expected, so increase its probability."

Why Separate Networks (Not Shared)?
The actor and critic use separate networks instead of shared layers because:

- **Policy needs stability:** The actor requires stable updates to maintain consistent behavior across the randomized environment
- **Value needs flexibility:** The critic must adapt quickly to changing reward distributions (especially with episodic style bonuses)
- **Different learning rates:** Policy and value estimation benefit from independent learning dynamics, preventing one from interfering with the other

4.2.2 Hyperparameter Selection Rationale

High Beta (0.1): Increased from default 0.015 to encourage exploration in the complex parkour environment. The linear decay schedule allows gradual shift from exploration to exploitation.

Selection Process:

- **Initial Value:** 0.015 (ML-Agents default)
- **Problem:** Agent converged too quickly, missed optimal strategies
- **Experimentation:** Tested 0.05, 0.1, 0.2
- **Result:** 0.1 provided best balance (high exploration, still learns effectively)
- **Decay:** Linear from 0.1 $\rightarrow$ $\sim$0.00074 over 2M steps

Time Horizon 128: Balanced between shorter horizons (64) that may miss long-term dependencies and longer horizons (192) that slow training. Appropriate for 100-second episodes.

Decay Formulas:

$$lr(t) = 3.0 \times 10^{-4} \times \left(1 - \frac{t}{2,000,000}\right) \tag{1}$$

$$beta(t) = 0.1 \times \left(1 - \frac{t}{2,000,000}\right) \tag{2}$$

$$epsilon(t) = 0.2 \times \left(1 - \frac{t}{2,000,000}\right) \tag{3}$$

4.3 Style Episode Frequency: Why 40%?

4.3.1 Empirical Evolution

The style episode frequency was **increased from 15% to 40%** during development based on empirical observations:

Initial Design (15%):

- Original implementation used `styleEpisodeFrequency = 0.15`
- Rationale: Provide occasional style incentives without overwhelming functional objectives
- Result: Roll usage remained low ($\sim$0.69% of actions, 28.1 rolls/episode)
- **Training Run:** `training_20251207_171550` (previous run before frequency increase)

Current Design (40%):

- Increased to `styleEpisodeFrequency = 0.4` (40% of episodes)
- Rationale: Provide more opportunities for style actions to be learned and expressed
- Result: Significantly increased roll usage (exact percentage from training logs)
- **Training Run:** `training_20251207_210205` (current run with 40% frequency)

Empirical Evidence:

- **15% Frequency:** Roll usage $\sim$0.69% of actions, agent rarely used rolls
- **40% Frequency:** Roll usage significantly increased (user confirmed "more rolls")
- **Reward Improvement:** +31% improvement (+67.90 $\rightarrow$ +89.18) with 40% frequency

4.3.2 Selection Criteria

The 40% frequency was chosen to balance three competing objectives:

1. **Sufficient Style Incentive:** High enough frequency to ensure the agent learns roll usage patterns
2. **Functional Behavior Preservation:** Low enough that 60% of episodes focus purely on functional objectives (progress, speed, efficiency)
3. **Behavioral Variety:** Creates diversity in agent behavior across episodes

4.3.3 Acknowledgment of Arbitrariness

We acknowledge that the 40% value is somewhat arbitrary and was selected through empirical tuning instead of theoretical optimization. The choice represents a practical balance point that:

- Provides sufficient style signal for learning
- Maintains functional behavior in majority of episodes
- Approximates preference diversity (different human evaluators might prefer different style/-efficiency trade-offs)

Alternative Frequencies Considered:

- **10–20%:** Too infrequent, agent rarely learns style actions (observed in 15% experiment)
- **50–60%:** Too frequent, risks prioritizing style over function (not tested, but theoretical concern)
- **40%:** Empirical sweet spot observed in training experiments

Theoretical Justification (Post-Hoc): While 40% was chosen empirically, we can justify it post-hoc:

- **Majority Functional (60%):** Ensures agent prioritizes reaching target
- **Substantial Style (40%):** Provides enough style signal for learning
- **Preference Diversity:** Mimics scenario where 40% of human evaluators prefer style, 60% prefer efficiency

4.3.4 Implementation Details

Code Location: `src/Assets/Scripts/CharacterConfig.cs`

```
[Tooltip("Probability that an episode will have style bonuses enabled (0.1
    = 10%, 0.2 = 20%)")]
[Range(0f, 1f)]
public float styleEpisodeFrequency = 0.4f; // Increased from 15% to 40%
```

Assignment Logic: `src/Assets/Scripts/ParkourAgent.cs`

```
public override void OnEpisodeBegin()
{
    // ... other reset logic ...

    // Randomly assign style bonus flag at episode start
    styleBonusEnabled = Random.Range(0f, 1f) < config.
        styleEpisodeFrequency;

    // ... rest of reset logic ...
}
```

Impact: The flag is assigned once per episode and affects all roll actions within that episode. This episodic-level assignment ensures consistent reward structure throughout each episode, making it easier for the agent to learn the relationship between style episodes and roll rewards.

4.4 Configuration System

4.4.1 Dual Configuration Architecture

The implementation uses a **dual configuration system** to separate training hyperparameters from environment/gameplay parameters:

1. ML-Agents Config (`parkour_config.yaml`):

- PPO hyperparameters (learning rate, batch size, etc.)
- Network architecture settings
- Training schedule (max steps, checkpoints)
- **Location:** `src/parkour_config.yaml`
- **Format:** YAML
- **Edited:** Text editor (no Unity required)

2. Unity ScriptableObject (`CharacterConfig.cs`):

- Movement parameters (speeds, jump force, gravity)
- Stamina system (max, consumption, regen rates)
- Reward values (progress multiplier, target reach, roll rewards)
- Environment settings (episode timeout, raycast distances)
- Style system (roll base reward, style bonus, frequency)
- **Location:** Unity Project (`Assets/Settings/CharacterConfig.asset`)
- **Format:** Unity ScriptableObject (serialized as `.asset` file)
- **Edited:** Unity Inspector (visual editor)

Rationale: This separation allows:

- **Training hyperparameters** to be adjusted without Unity recompilation
- **Gameplay parameters** to be tuned in Unity Editor with immediate visual feedback
- **Version control** of both configuration types independently
- **Team Collaboration:** RL researchers can edit YAML, game designers can edit ScriptableObject

4.4.2 Key Configuration Values

Movement:

- Jog speed: 6 units/sec
- Sprint speed: 12 units/sec
- Roll speed: 18 units/sec (1.5× sprint)

Stamina System:

- Max stamina: 100.0
- Sprint consumption: 20.0/sec
- Jump cost: 20.0 per jump
- Roll cost: 60.0 per roll
- Regen rate: 30.0/sec (when not sprinting/jumping/rolling)

Rewards:

- Progress: 0.1 per unit forward
- Target reach: +10.0
- Roll base: +0.5 (always given)
- Roll style bonus: +1.5 (in 40% of episodes)

4.5 Constraint Enforcement: Learning Through Environment Feedback

Environment-Enforced Constraints: Action constraints are enforced by the environment instead of through explicit penalties, allowing the agent to learn limits through experience:

- **Stamina Depletion:** When stamina reaches zero during sprint, the environment automatically switches to jog. The agent learns stamina management by experiencing action failures, not through explicit penalties.
- **Action Blocking:** Jump and roll actions are blocked when constraints are violated (insufficient stamina, not grounded, cooldown active). The agent receives zero reward for blocked actions, learning that certain state-action combinations are invalid.
- **Cooldown Enforcement:** Sprint cooldown (0.5s) prevents immediate re-sprint. The agent learns timing constraints through failed sprint attempts during cooldown periods.

Design Rationale: This approach is more robust than reward penalties because:

1. **Hard Constraints:** The environment physically prevents invalid actions, ensuring the agent cannot exploit reward functions to bypass constraints.
2. **Natural Learning:** The agent discovers constraints through exploration and failure, similar to how humans learn physical limitations.
3. **Generalization:** Environment-enforced constraints work across all reward configurations, making the system more modular and maintainable.

5 Results & Analysis

5.1 Training Performance

Final Performance Metrics (2M steps):

- **Final Reward:** +89.18 (at 2M steps)
- **Previous Best:** +78.32 (run28, sprint-only configuration)
- **Improvement:** +14% over previous best, +31% over roll system v1 (+67.90)

Training Progression:

Checkpoint	Reward	Improvement from 500k
500k steps	+26.67	Baseline
1.0M steps	+45.25	+69.5%
1.5M steps	+81.60	+205.8%
2.0M steps	+89.18	+234.3%

Table 6: Training progression

Training Metrics:

- **Policy Loss:** 0.0233 (mean, range 0.0175–0.0312) — Stable, converged
- **Value Loss:** 0.985 (mean, range 0.400–1.808) — Reasonable estimation error
- **Policy Entropy:** 0.657 (mean, range 0.657–1.605) — High exploration maintained
- **Learning Rate (final):** 8.36×10^{-7} (decayed from 3.0×10^{-4})
- **Epsilon (final):** 0.100 (decayed from 0.2)
- **Beta (final):** 0.000289 (decayed from 0.1)

5.2 Episode Statistics

Mean Episode Performance:

- **Mean Episode Reward:** 80.06 (range 3.05–88.82)
- **Mean Episode Length:** 61.07 steps (range 4.90–68.50)
- **Mean Max Distance:** 555.91 units (range 29.89–603.56)
- **Mean Episode Duration:** 609.64 environment steps

Reward Range Interpretation:

- **Minimum (3.05):** Episodes that fail early (timeout/fall) — minimal progress, no target reach
- **Maximum (88.82):** Successful episodes with optimal behavior — full progress + target + efficient action usage
- **Mean (80.06):** Typical successful episode (matches reward breakdown in Section 3.2.4)

5.3 Action Distribution and Behavior

Action Distribution (Percentage of Total Actions):

- **Jog:** 67.61% (primary movement mode)
- **Sprint:** 14.00% (strategic speed bursts)
- **Roll:** 7.81% (increased from 0.69% in previous run — 11.3× improvement)
- **Jump:** 3.53% (gap crossing)
- **Idle:** 7.04% (minimal, efficient)

Action Counts (Per Episode, Mean):

- **Jog:** 2,072 actions
- **Sprint:** 424 actions
- **Roll:** 239 actions
- **Jump:** 102 actions
- **Idle:** 216 actions

Agent Behavior Analysis:

- **Roll Usage:** 7.81% of actions (vs 0.69% in previous run with 15% style frequency)
- **Roll Count:** 239 rolls per episode (mean)
- **Roll Improvement:** 11.3× increase over previous run (0.69% → 7.81%)
- **Strategic Roll Usage:** Rolls used at 7.81% despite high cost (60 stamina), indicating learned strategic value

5.4 Behavioral Emergence: Style Bonus Impact

Comparative Analysis:

The stochastic reward shaping (40% style frequency) significantly increased roll usage compared to baseline configurations:

Configuration	Style Frequency	Roll Usage	Final Reward	Notes
Baseline (run28)	0% (no rolls)	0%	+78.32	Sprint-only, no roll action
Roll System v1	15%	0.69%	+67.90	Roll cost 150, insufficient incentive
Current (training_21)	**40%**	**7.81%**	**+89.18**	Dual reward structure, strategic usage

Table 7: Comparative analysis of style bonus impact

Key Behavioral Changes:

1. **Roll Integration:** Agent learned to use rolls strategically (7.81% usage) despite high stamina cost (60 per roll)
2. **Stamina Management:** Agent balances sprint (14%) and roll (7.81%) usage, maintaining stamina for critical actions
3. **Movement Diversity:** Primary movement is jog (67.61%), with strategic use of sprint and roll for speed and style

5.5 Training Dynamics

Convergence Analysis:

- **Reward Curve:** Monotonically increasing from 500k to 2M steps, no catastrophic forgetting
- **Policy Convergence:** Policy loss stabilized at 0.0233, indicating converged policy
- **Value Estimation:** Value loss at 0.985 reflects reasonable estimation error for 850-step episodes
- **Exploration:** Policy entropy maintained at 0.657, indicating continued exploration even at convergence

Learning Rate Decay: The linear decay schedule successfully shifted from exploration to exploitation:

- Initial learning rate: 3.0×10^{-4}
- Final learning rate: 8.36×10^{-7} (99.7% decay)
- Beta decay: $0.1 \rightarrow 0.000289$ (99.7% decay)
- Epsilon decay: $0.2 \rightarrow 0.100$ (50% decay)

Style Bonus Impact on Learning: The episodic style bonus (40% frequency) created behavioral variety without destabilizing learning:

- Consistent reward structure within episodes (style flag assigned at episode start)
- Agent learned to adapt behavior based on episode type
- No evidence of reward confusion or learning instability

6 Discussion & Future Work

6.1 RLHF Integration

The fundamental constraint identified in Section 3.1.3—that human feedback is incompatible with 20× accelerated training—motivates exploring offline RLHF approaches that decouple human evaluation from real-time training. Several integration strategies are viable with the existing environment:

6.1.1 Option 1: Asynchronous Preference Collection (Most Viable)

Approach: Decouple training from human feedback by collecting preferences between training runs.

Workflow:

1. **Normal Training Phase:** Run standard training (28 agents, 30 minutes, 2M steps) with current reward model
2. **Trajectory Extraction:** Automatically sample trajectory pairs from replay buffer and save as video clips (∼1 minute)
3. **Human Labeling:** Human evaluates comparison pairs between training runs (10–15 minutes of human time per iteration)
4. **Reward Model Update:** Train reward model on accumulated preferences (∼5 minutes)
5. **Iterate:** Next training run uses updated reward model

Total Time per Iteration: 30 min (training) + 15 min (human) + 5 min (model update) = 50 minutes

Advantages:

- Maintains accelerated training efficiency (30 min per 2M steps)
- Human feedback occurs at human pace, not constrained by training speed
- Can accumulate preferences across multiple training runs
- Compatible with existing infrastructure (Unity recorder, replay buffer)

Why Not Implemented: This approach requires implementing trajectory recording, video generation, and reward model training infrastructure, which was beyond the scope of the initial proof-of-concept for stochastic reward shaping.

6.1.2 Option 2: Synchronous Feedback with Checkpointing (Partially Viable)

Approach: Hybrid training with alternating slow (observable) and fast (accelerated) phases.

Workflow:

1. **Phase A - Observable Training:** Train at normal speed (1× time scale) with 4 agents for 5–10 minutes, human provides real-time keyboard feedback
2. **Phase B - Accelerated Training:** Switch to accelerated mode (10× time scale) with 28 agents for 20 minutes using current reward model
3. **Phase C - Model Update:** Train reward model on accumulated preferences (∼5 minutes)
4. **Repeat:** Cycle through phases

Advantages:

- Allows real-time human feedback during observable phases
- Maintains training efficiency through accelerated phases
- Human can provide immediate corrections during slow phases

Limitations:

- Requires manual intervention during observable phases
- Less efficient than fully asynchronous approach
- Human must be present during training sessions

6.1.3 Option 3: Pre-train Reward Model, Then RL (Most Pragmatic)

Approach: One-time offline preference collection before RL training begins.

Workflow:

1. **Phase 1 - Offline Collection:** Generate trajectories from random or hand-crafted policies, collect 200–500 human preference pairs (2–3 hours of human time, one-time cost)
2. **Phase 2 - Reward Model Training:** Train initial reward model using Bradley-Terry model on collected preferences
3. **Phase 3 - RL Training:** Use learned reward model for standard RL training (current 30-minute setup)

Advantages:

- Minimal infrastructure changes (one-time preference collection)
- No ongoing human time commitment during training
- Can bootstrap from human preferences before any RL training
- Compatible with existing accelerated training setup

Limitations:

- Reward model is static (doesn't adapt as policy improves)
- May not capture preferences for behaviors that only emerge during training
- Requires substantial upfront human time investment

6.1.4 Option 4: Minimal Viable RLHF (Simplest Implementation)

Approach: Post-training clip rating with simple regression model.

Workflow:

1. **Training:** Run standard training (30 minutes, 2M steps)
2. **Auto-extraction:** Unity automatically saves 10 "style moment" clips (rolls, jumps, acrobatic sequences)
3. **Human Rating:** Human rates each clip 1–5 stars ($\sim$2 minutes per iteration)
4. **Simple Reward Model:** Fit linear regression model predicting star rating from state features (height, velocity, rotation)
5. **Next Iteration:** Use predicted rating as style reward in subsequent training

Advantages:

- Minimal implementation complexity
- Very low human time commitment (2 minutes per iteration)
- Can iterate quickly (10 iterations = 20 minutes human time)
- Directly addresses style evaluation without full RLHF infrastructure

Limitations:

- Simpler reward model (linear regression vs. neural network)
- Only evaluates style moments, not full trajectory preferences
- May not capture complex preference relationships

Comparison to Current Approach:

The current stochastic reward shaping (40% style frequency) can be viewed as a zero-human-time approximation of Option 4, where the "human rating" is replaced by a stochastic binary signal (style episode flag). Future work could validate whether any of these RLHF approaches provide better preference approximation than the current stochastic method, or whether the approximation quality is sufficient for the target application.

6.2 Training Optimization

6.2.1 Hyperparameter Optimization

Current hyperparameters use linear decay schedules, which may be suboptimal for the learning dynamics observed in this domain. Current configuration: Beta 0.1 (linear decay), entropy 0.635 (indicating over-exploration), learning rate 3×10^{-4}.

Proposed Improvements:

- **Beta:** Exponential decay from $0.05 \rightarrow 0.001$ (vs. current linear 0.1)
- **Learning Rate:** Exponential decay from $5 \times 10^{-4} \rightarrow 1 \times 10^{-5}$ (vs. current linear 3×10^{-4})
- **Epsilon:** Exponential decay from $0.15 \rightarrow 0.05$ (vs. current linear 0.2)

- **Lambda (GAE):** Increase to 0.98 (vs. current 0.95)
- **Epochs:** Reduce to 3 (vs. current 5)

Rationale: Exponential decay schedules better match observed learning curves, where policy convergence accelerates in later training stages. Current linear decay is too slow, maintaining high exploration (entropy 0.635) when exploitation should dominate.

Expected Outcomes: Final reward +95–100 (vs. current +89.18), faster convergence, lower final entropy ∼0.2–0.3 (vs. current 0.657).

6.2.2 Movement Smoothing

Issue: Agent exhibits sprint stuttering behavior, lacking realistic visual flow. This is unrelated to roll usage but indicates suboptimal reward structure for continuous movement.

Proposed Solution: Momentum-based movement system with flow penalties:

- **Continuous Sprint:** Increase speed from 12 → 14 units/sec when sprint is maintained
- **Interruption Penalty:** -0.005 per sprint interruption to encourage sustained movement
- **Momentum Reward:** Small positive reward for maintaining consistent movement direction

This addresses the visual realism gap by incentivizing smooth, continuous movement patterns instead of rapid start-stop behavior.

6.3 Environment and Action Space Extensions

6.3.1 Full 3D Movement

Current implementation uses 2.5D movement (forward/backward, up/down, but limited turning). Extending to full 3D navigation would enable:

- **Multi-axis platforms:** Platforms at arbitrary orientations, not just horizontal
- **Turning mechanics:** Agent must learn to orient toward targets in 3D space
- **Spatial reasoning:** More complex state space requiring 3D spatial awareness

This extension would test generalization to more complex navigation problems while maintaining the core preference learning challenge.

6.3.2 Expanded Action Space

Current action space has 5 discrete actions (Idle, Jump, Jog, Sprint, Roll). Proposed expansion:

- **Additional Actions:** Slide, wall jump, vault (expanding from 5 → 9+ actions)
- **Alternative:** Continuous control for movement direction and intensity
- **Complexity:** Horizontal expansion of problem complexity, testing whether stochastic reward shaping scales to larger action spaces

This would explore whether the episodic style bonus mechanism generalizes when more stylistic actions are available, and whether the agent can learn strategic selection among a larger action set.

6.3.3 Dynamic Obstacles

Current environment uses static, procedurally generated platforms. Adding dynamic elements:

- **Moving Platforms:** Platforms that translate or rotate over time
- **Time-Dependent Physics:** Environmental changes that require temporal reasoning
- **Partial Observability:** Some obstacles may be occluded or only partially visible

This extension would test whether the learned policy generalizes to non-stationary environments and whether the current fully observable state space (14 observations) remains sufficient.

6.4 Workflow and Algorithmic Improvements

6.4.1 LLM-Assisted Hyperparameter Tuning

Proposed: Automated hyperparameter optimization using LLM-based reasoning to adapt hyperparameters when environment changes occur.

Rationale: Current manual hyperparameter tuning is time-consuming and requires domain expertise. LLM assistance could:

- Analyze training curves and suggest hyperparameter adjustments
- Adapt hyperparameters when environment parameters change (e.g., platform count, gap ranges)
- Reduce iteration time for reward calibration experiments

This workflow optimization would accelerate the iterative design process demonstrated in Section 3.2.4 (Iterative Reward Calibration).

6.4.2 Q-Learning and DQN Benchmark

Proposed: Implement Q-learning and DQN algorithms on the same environment to benchmark against PPO.

Rationale: Section 3.1.4 (Why PPO Over Q-Learning/DQN) provides theoretical arguments for PPO's superiority, but empirical comparison would validate these claims. This would test:

- Whether Q-learning/DQN can handle the continuous state space (14 observations) with discrete actions
- Whether deterministic action selection (Q-learning) vs. stochastic policy (PPO) affects style action discovery
- Whether value-based methods can learn from sparse rewards (target reach +10.0, fall -1.0) as effectively as policy gradient methods

This experimental comparison would provide empirical evidence for the algorithm selection rationale.