LETSPLAY: A GAMEPLAY TRAJECTORY DATASET FOR VISUAL GAME AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Playing games through screenshots requires agents to interpret visual observations, remember past events, and execute actions toward a task objective. Training agents to acquire these capabilities requires gameplay trajectories that capture both what the agent observes and how it interacts with the environment. However, existing game benchmarks often rely on textual state descriptions or game-specific actions, while many visual game environments provide no gameplay trajectories for training. To address this, we introduce LETSPLAY, a gameplay trajectory dataset for training visual game agents. The dataset aligns screenshots with recorded device actions using a unified action schema and further annotates each action with its reasoning, immediate goal, and expected outcome. In total, LETSPLAY covers 25 games, comprising 2,036 gameplay trajectories and 315K action steps. Building on LETSPLAY, we develop PLAYAGENT, a multimodal agent for long-horizon gameplay. The agent conditions its decisions on recent actions and gameplay context. Extensive experiments demonstrate the usefulness of LETSPLAY for training visual game agents and the effectiveness of PLAYAGENT in completing game tasks.

1 INTRODUCTION

Video games provide practical environments for developing agents that connect visual understanding with sequential decision-making. Successful play requires an agent to interpret changing game screens, remember earlier events, and coordinate actions to make progress toward a task objective. Recent vision-language models (VLMs) make it possible to approach these tasks through game screenshots and device controls, connecting language-guided reasoning with actions in graphical user interfaces (GUIs) (Tan et al., 2024; Wang et al., 2025b). Training VLMs for these tasks requires gameplay trajectories comprising visual observations and their corresponding actions.

However, existing game benchmarks have limitations in visual input, action control, and training data. Text-based benchmarks such as SmartPlay (Wu et al., 2023) describe game states in words, eliminating the need for visual interpretation. Game-specific action interfaces also let agents execute predefined game commands without specifying device inputs. Visual benchmarks such as GameWorld (Ouyang et al., 2026) support screenshot-based interaction but lack corresponding gameplay trajectories for training. Among existing gameplay trajectory datasets, MineRL (Guss et al., 2019) is limited to Minecraft, while NitroGen (Magne et al., 2026) covers multiple games but focuses on gamepad control without language instructions.

To address this, we introduce **LETSPLAY**, a gameplay trajectory dataset for training visual game agents. We collect task-directed gameplay trajectories and align each GUI screenshot with the action performed at that step. To represent actions consistently across different input devices, we propose a unified action representation that defines a common action space for the model. For each step in a trajectory, we further annotate the reasoning behind the action, its immediate goal, and the expected outcome. These annotations provide language-based supervision for model training. Our dataset supports supervised fine-tuning, while the game environments and task evaluators enable online reinforcement learning with verifiable feedback.

Building on these trajectories, we develop PLAYAGENT by fine-tuning a vision-language model (VLM). Given the task instruction, current screenshot, input device state, and interaction history, the agent decides the next executable action. During training, we group trajectories by length within

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

each game and adopt a curriculum that gradually shifts from shorter to longer trajectories. To mitigate the imbalance caused by the large number of steps without device operations, we assign higher loss weights to tokens corresponding to device actions and task-completion signals. PLAYAGENT achieves a task success rate of 36.0%, outperforming the strongest baseline by 2.4%.

2 RELATED WORK

LM-based Game Agents. Existing LM-based game agents can be broadly divided into two categories: agents that plan through structured game interfaces and agents that interact directly with games through visual observations and device controls. The first category reads textual descriptions of game states and executes actions through game-specific APIs. Voyager (Wang et al., 2023) employs an automatic exploration curriculum to continually acquire new executable skills for playing Minecraft. GITM (Zhu et al., 2023) further leverages textual knowledge and memory to decompose high-level goals into structured action plans. While these abstractions enable effective long-horizon planning, they rely on game-specific state representations and control interfaces, thereby bypassing much of the visual perception and general-purpose device control required for direct interaction with real game environments. The second category operates directly on visual observations and controls games through native keyboard and mouse actions. Cradle (Tan et al., 2024) proposes a modular agent framework that coordinates planning, reflection, and memory for screenshot-based game interaction. Game-TARS (Wang et al., 2025b) learns native device actions together with sparse reasoning, while Lumine (Tan et al., 2025) learns gameplay control, instruction following, and reasoning capabilities from annotated trajectories. These studies show that VLMs can play games directly through screenshots and device controls.

Game Benchmarks. Existing game benchmarks evaluate the capabilities of LMs in gameplay, ranging from strategic reasoning to visual perception and control. Text-based benchmarks such as GAMA-Bench (Huang et al., 2024), GameBench (Costarelli et al., 2024), and SmartPlay (Wu et al., 2023) assess planning, decision-making, and reasoning through structured textual interactions, but avoid direct visual perception. LVLM-Playground (Wang et al., 2025a) and BALROG (Paglieri et al., 2025) introduce visual board and grid environments, yet still rely on environment-specific action spaces. Recent benchmarks move closer to native gameplay. V-MAGE (Zheng et al., 2026), LMGame-Bench (Hu et al., 2026), Orak (Park et al., 2026), and VideoGameBench (Zhang et al., 2025a) evaluate agents in visually rich game environments, while GameWorld (Ouyang et al., 2026) provides browser games with shared control interfaces and verifiable task outcomes. However, these benchmarks primarily focus on evaluation and provide limited gameplay trajectories for training.

VLA Learning. Predicting game actions from screenshots and instructions follows the vision-language-action (VLA) approach of mapping visual and language inputs to executable controls. SIMA (SIMA Team, 2024) and SIMA 2 (SIMA Team, 2025) learn keyboard and mouse control across virtual worlds, while JARVIS-VLA (Li et al., 2025) trains a VLM on Minecraft gameplay trajectories. In robotics, OpenVLA (Kim et al., 2024) learns arm and gripper control from recorded robot trajectories. RT-2 (Brohan et al., 2023) predicts robot movements as text tokens, while π_0 (Black et al., 2024) generates joint and gripper commands for multiple future steps through flow matching.

3 PLAYTRACE DATASET

Training visual game agents to follow task objectives requires gameplay trajectories that connect visual observations with executable actions and provide supervision for understanding why those actions are taken. Table 1 summarizes existing game benchmarks and datasets. Text-based benchmarks represent game states as textual descriptions, thereby bypassing perception of the game screen. Visual benchmarks such as GameWorld support screenshot-based interaction and verifiable task evaluation, but are designed primarily for evaluating agents rather than providing trajectories for training them (Ouyang et al., 2026). Existing visual gameplay datasets also remain limited in either game coverage or language supervision. MineRL pairs game screenshots with player actions, but is restricted to Minecraft (Guss et al., 2019). NitroGen covers a much broader range of games by pairing gameplay videos with gamepad actions, but does not provide explicit language supervision describing the task objective or the reasoning behind individual actions (Magne et al., 2026). To address

these limitations, we introduce **LETSPLAY**, a gameplay trajectory dataset that jointly provides game screenshots, executable device actions, and explicit language annotations describing the reasoning and goal associated with each action. This combination enables visual game agents to learn not only how to act from game screens, but also how individual actions contribute to accomplishing the task objective.

Table 1: **Comparison of Game Datasets and Benchmarks.** *Input Modality* specifies how game states are represented: GUI uses game screenshots, while Text uses textual descriptions. *Device Control* indicates whether agents interact through device inputs, such as keyboard, mouse, or controller actions, rather than game-specific semantic actions. *Training Set* indicates whether gameplay demonstrations are provided for model training. *Task Instruction* indicates whether explicit gameplay objectives are specified. *Action Annotation* indicates whether each action is paired with language supervision describing its rationale and intended goal. ✓: provided; ✗: not provided.

Dataset / Benchmark	Games	Input Modality	Device Control	Training Set	Task Instruction	Action Annotation
MineRL (Guss et al., 2019)	1	GUI	✓	✓	✓	✗
FlashAdventure (Ahn et al., 2025)	34	GUI	✓	✗	✓	✗
JARVIS-VLA (Li et al., 2025)	1	GUI	✓	✓	✓	✗
MCU (Zheng et al., 2025)	1	GUI	✓	✗	✓	✗
StarBench (Zhang et al., 2025b)	1	GUI	✓	✗	✓	✗
VideoGameBench (Zhang et al., 2025a)	23	GUI	✓	✗	✓	✗
GameWorld (Ouyang et al., 2026)	34	GUI	✓	✗	✓	✗
LMGame-Bench (Hu et al., 2026)	6	Text	✗	✗	✓	✗
NitroGen (Magne et al., 2026)	1000+	GUI	✓	✓	✗	✗
Open P2P (Yue et al., 2026)	44	GUI	✓	✓	✓	✗
Orak (Park et al., 2026)	12	Text	✓	✓	✓	✗
V-MAGE (Zheng et al., 2026)	5	GUI	✗	✗	✓	✗
LETSPLAY (ours)	25	GUI	✓	✓	✓	✓

3.1 DATASET OVERVIEW

Dataset formulation. We formulate gameplay as a Partially Observable Decision Process (POMDP) conditioned on a task context D, which specifies the task objective, game rules, and legal controls. At each step t, the agent receives a game screenshot o_t as its visual observation, together with the current input device state s_t^{in}, which records held keys, held mouse buttons, and the cursor position. Let H_t denote the interaction history consisting of preceding observations and actions. The agent selects an executable device action according to

$$a_t \sim \pi_\theta\big(\cdot \mid D, H_t, s_t^{\text{in}}, o_t\big). \tag{1}$$

Executing a_t advances the game and produces the next observation o_{t+1}. The interaction continues until the task is completed, a terminal condition is reached, or the interaction budget is exhausted, yielding a gameplay trajectory.

Our curated dataset consists of trajectories that successfully complete their respective game tasks, each trajectory τ_i contains its task context D_i and an ordered sequence of T_i decision records:

$$\tau_i = \Big(D_i, \big(o_{i,t}, s_{i,t}^{\text{in}}, y_{i,t}^{\text{th}}, y_{i,t}^{\text{g}}, a_{i,t}, y_{i,t}^{\text{eo}}\big)_{t=1}^{T_i}\Big). \tag{2}$$

Here, $y_{i,t}^{\text{th}}$ gives a short rationale for the recorded action $a_{i,t}$, $y_{i,t}^{\text{g}}$ states the goal pursued by this action, and $y_{i,t}^{\text{eo}}$ describes its expected outcome. These fields provide additional supervision alongside the recorded actions, enabling agents to infer goals and plan actions from visual observations.

3.2 GAMEPLAY TRAJECTORY COLLECTION

LETSPLAY is built on 25 games spanning a diverse range of gameplay scenarios, such as reactive control, puzzle solving, and simulation games. Appendix A describes the gameplay of each game.

Expert Model-Based Trajectory Generation. We design a game-specific expert model for each game to generate successful gameplay trajectories. For each game, we first construct a discrete

action set from its legal inputs, where each action is assigned a unique identifier and mapped to a corresponding keyboard or mouse event. At each step, the expert observes the structured game state exposed by the game API, which contains information available to human players, and selects an action according to predefined game-specific heuristics. For example, in the `Chrome Dino` game, the expert jumps over approaching obstacles, ducks under low-flying birds, and waits when no collision is imminent. For games with multiple levels, we design a dedicated expert model for each level. Since the expert policies are rule-based, we vary the game seed across rollouts to induce diverse game states and increase trajectory diversity. During each rollout, the expert observes the current game state and selects an action for execution. The game state is queried every 50 ms, and the updated state is used to select the next action. We record the resulting action sequences and retain only trajectories for which the task evaluator confirms successful task completion. The full expert policies are provided in the appendix, and Table 2 summarizes their task coverage across game types.

Table 2: **Task Coverage of Expert Model-Based Trajectory Generation.** A task is completed when its expert produces at least one trajectory verified by the task evaluator. *Expert Success* is ✓ if the expert completes at least one task and ✗ otherwise. *Tasks Completed / Total* reports the number of distinct completed tasks and the total number of tasks for each game.

Genre	Game	Expert Success	Tasks Completed / Total
Arcade (3)	Core Ball	✓	5 / 5
	Google Snake	✓	5 / 5
	Pac-Man	✓	5 / 5
Platformer (5)	Another Gentleman's Adventure	✓	3 / 5
	Captain Callisto	✗	0 / 5
	Doodle Jump	✓	5 / 5
	GeoDash	✗	0 / 5
	Mario Game	✗	0 / 5
Puzzle (7)	2048	✓	5 / 5
	Astray	✓	5 / 5
	Hextris	✓	5 / 5
	Minesweeper	✓	5 / 5
	Stack	✓	5 / 5
	Tetris	✗	0 / 5
	Wordle	✓	4 / 5
Runner (8)	Boxel Rebound	✓	5 / 5
	Chrome Dino	✓	5 / 5
	Cubefield	✗	0 / 5
	Edge Surf	✓	5 / 5
	Flappy Bird	✓	5 / 5
	NS-Shaft	✗	0 / 5
	Run 3	✗	0 / 5
	Temple Run 2	✓	5 / 5
Simulation (2)	Monkey Mart	✗	0 / 5
	Wolf3D	✓	5 / 5
Total	**25 games**	—	**82 / 125**

Human-Based Trajectory Collection. For tasks that cannot be completed by the expert models, we additionally collect trajectories from human players. Each rollout starts by resetting the game with a random seed. Human players interact with the game in real time using the keyboard and mouse, while we record keystrokes, clicks, drags, and scrolls. The gameplay is continuously recorded to obtain the corresponding visual observations. We then convert the continuous human inputs into the same control step format as the expert trajectories. Inputs within the same sampled frame are merged into a single multi-channel action. Key presses shorter than 150 ms are treated as taps, while longer presses are represented by a hold followed by a release. Mouse movements are recorded as moves or drags according to the input state. We insert at most three wait actions between consecutive inputs to preserve their temporal gaps and capture the resulting game states. A trajectory is retained when the task evaluator confirms successful task completion.

We combine the successful trajectories generated by the expert models with those collected from human players to form LETSPLAY. The resulting dataset contains 2036 gameplay trajectories across 25 games, comprising 315K steps.

3.3 UNIFIED ACTION REPRESENTATION

We define a unified action representation for the device controls used by the visual game agent. At each interaction step, the agent encloses all device operations in a single block delimited by <act> and </act>. This block specifies the executable arguments passed to the device control interface. The complete action specification provided to the agent is listed in Appendix B.1.

Action Representation. Each action can combine keyboard input, mouse button operations, pointer movement, and scrolling. The keyboard channel uses <kb_hold>, <kb_tap>, and <kb_release> to specify holding, tapping, and releasing keys, respectively. Each operation is followed by one or more <key_NAME> tokens identifying the target keys. Text entry is specified by the literal text enclosed between <kb_text> and </kb_text>. The mouse button channel similarly uses <ms_hold>, <ms_tap>, and <ms_release> to specify holding, clicking, and releasing buttons. Their arguments are one or more tokens selected from <btn_left>, <btn_right>, and <btn_middle>, representing the left, right, and middle mouse buttons, respectively. The <drag> operation performs a complete left button drag within one step and takes four coordinate tokens in the order (x_0, y_0, x_1, y_1), where (x_0, y_0) and (x_1, y_1) denote the start and end positions. The pointer movement channel uses <move_abs> for movement to an absolute screen position and <move_rel> for movement by a relative displacement. Both operations take two coordinate tokens in horizontal and vertical order, specifying the target position or signed displacement, respectively. The scrolling channel uses <scroll> followed by two tokens specifying the signed horizontal and vertical mouse wheel displacements. When no new device input is needed, the action block contains only <noop>. To ensure a consistent representation, we serialize channels in the order of keyboard input, mouse buttons, pointer movement, and scrolling. Within the keyboard and mouse button channels, operations appear in the order of hold, tap, and release, followed by text entry or drag when present. Key and button arguments are deduplicated and sorted in ASCII order. For example, in a game where Shift enables running, w moves forward, and Space triggers a jump, the agent can start running forward while jumping once by emitting

```
<act> <kb_hold> <key_Shift> <key_w> <kb_tap> <key_Space> </act>
```

Coordinate Encoding. To represent spatial actions consistently across screen resolutions, we normalize coordinates and displacements by the corresponding screenshot dimensions. All spatial arguments use a shared vocabulary of 1,001 tokens from <bin_000> to <bin_1000>. Let L denote the screenshot width for the horizontal axis and its height for the vertical axis. We clip an absolute coordinate u to $[0, L]$ and a signed displacement δ to $[-L/2, L/2]$, then compute their token indices as

$$q_{\text{abs}}(u, L) = \text{round}\left(1000\frac{u}{L}\right), \qquad q_{\text{rel}}(\delta, L) = 500 + \text{round}\left(1000\frac{\delta}{L}\right). \tag{3}$$

Absolute pointer movement and drag endpoints use q_{abs}, with positions measured from the upper left corner of the screenshot. Along each axis, token indices 0 and 1000 correspond to the two image boundaries. Relative pointer movement and scrolling use q_{rel}, with <bin_500> representing zero displacement. Indices above 500 indicate movement right or down, while indices below 500 indicate movement left or up. Indices 0 and 1000 represent displacements of $-L/2$ and $L/2$, respectively. For execution, we convert a token index k to $kL/1000$ for an absolute position or $(k - 500)L/1000$ for a signed displacement. This encoding provides a shared token representation that can be mapped to pixel values at the current screen resolution. For example, the agent can turn the camera right and slightly up by emitting

```
<act> <move_rel> <bin_640> <bin_480> </act>
```

We convert the recorded action at every step in our collected trajectories into this unified representation. Device operations are mapped to the corresponding action tokens, and spatial coordinate arguments are quantized using the dimensions of the associated screenshot.

3.4 TRAJECTORY STEP ANNOTATION

Each step in the trajectories pairs a game screenshot with the corresponding executed action. To train game agents to reason about visual observations and plan their actions, we augment each step

with three language annotation fields describing the action's rationale, goal, and expected outcome. The <think> field explains the action in the context of the visible scene, the <goal> field states the objective pursued by the action, and the <eo> field describes its expected immediate effect.

Specifically, we prompt Qwen-3.8-Max to reconstruct the rationale, goal, and expected outcome for each step t. The system prompt provides the game rules and controls, the task objective, and the definition of device actions, followed by annotation instructions. The user message presents up to four preceding steps with their screenshots and recorded actions, the current input device state, the current screenshot, and the executed action, together with the screenshot captured immediately after the action. For a step that declares task completion, the model additionally explains why the task objective is satisfied. Based on this input, the model produces a structured response with three fields, i.e., `thought`, `goal`, and `expected_outcome`. The full annotation prompt is provided in Appendix C. Each annotation is written from the player's perspective. The rationale is grounded in the current observation and the preceding steps, while the expected outcome describes the immediate effect of the executed action. Although the subsequent screenshot informs the expected outcome, the annotation instructions require it to be phrased as a prediction made before the action. We then attach these annotations to the corresponding recorded step and construct the training target in the order of <think>, <goal>, <act>, and <eo>.

4 PLAYAGENT

4.1 VISUAL GAME AGENT

Building on LETSPLAY, we develop PLAYAGENT, a visual game agent that follows task instructions and executes device inputs to complete game objectives. Effective gameplay requires the agent to interpret ongoing visual changes while tracking task progress over time. A single screenshot is often insufficient to infer temporal information such as motion direction or prior progress, while retaining the entire interaction history would lead to increasingly long inputs. To balance temporal context and input efficiency, we fine-tune a pretrained vision-language model with two complementary history windows, i.e., a recent window that captures short-term visual dynamics and a long-term window that retains earlier control decisions relevant to task progress.

The model comprises a vision encoder, a multimodal projector, and a language backbone. It combines screenshots with the task context, input device state, and interaction history to generate a rationale, an immediate goal, an executable action, and an expected outcome during training. After each action, the agent updates its observation and history for the next decision.

Observation and history. Each decision is conditioned on the task context D, current screenshot o_t, input device state s_t^{in}, and a bounded interaction history. We construct this history using two windows with complementary roles. The recent window H_t^{recent} contains up to n_r consecutive preceding steps, including <noop> steps, to capture local visual changes. The long-term window H_t^{long} retains up to n_l of the most recent steps containing device operations before the recent window, preserving earlier control decisions without duplicating recent entries. Together, the two windows provide complementary context for understanding local dynamics and tracking task progress.

Each history entry h_j consists of the game screenshot, immediate goal, and executed action at step j. With entries arranged chronologically within each window, the complete model input is

$$x_t = \big(D, H_t^{\text{long}}, H_t^{\text{recent}}, s_t^{\text{in}}, o_t\big), \qquad h_j = \big(o_j, y_j^{\text{g}}, a_j\big), \quad j < t, \tag{4}$$

Both prompts are provided in Appendix B.1 and Appendix B.2, respectively.

Structured decision generation. Given x_t, the model autoregressively generates the four target segments:

$$y_t = y_t^{\text{th}} \oplus y_t^{\text{g}} \oplus a_t \oplus y_t^{\text{eo}}, \qquad p_\theta(y_t \mid x_t) = \prod_{j=1}^{|y_t|} p_\theta(y_{t,j} \mid x_t, y_{t,<j}), \tag{5}$$

where $\oplus$ denotes concatenation. Each segment is enclosed by its corresponding tags, and the target ends with an end-of-turn token. The complete response format is provided in Appendix B.3.

Closed-loop execution. The agent parses each generated action, executes the corresponding device operations in the game, and updates the device input state s_t^{in}, which tracks held keys, held mouse

buttons, and the cursor position. It then incorporates the next screenshot and the updated input state into its history to determine the next action. When the agent judges that the task objective has been achieved, it emits <complete> within the <act> block to signal completion, which is then verified by the task evaluator.

4.2 Curriculum Learning with Action Class Loss Shaping

Gameplay trajectories vary substantially in length and contain many more <noop> steps than device operations or task completion signals. These properties create two challenges for supervised training. Sampling decisions uniformly from the full dataset gives longer trajectories more influence from the beginning of training, without a gradual progression from shorter to longer task processes. At the same time, frequent <noop> steps can dominate action supervision, leaving less emphasis on the decisions that change device inputs or end a task. To address these challenges, we combine a trajectory length curriculum with action class loss shaping. The curriculum gradually shifts training toward longer trajectories, while loss shaping gives greater weight to device operations and task completion signals throughout training.

Trajectory length curriculum. We organize training to learn from shorter trajectories first and progressively increase exposure to decisions from longer trajectories. This curriculum allows the model to first acquire control behaviors over relatively short task processes before placing greater emphasis on decision-making over extended gameplay. Because typical episode lengths vary across games, we group trajectories by length within each game. Let $\mathcal{T}_g$ denote the set of training trajectories for game g, and let T_i denote the number of steps in trajectory τ_i. We partition $\mathcal{T}_g$ into K ordered length buckets defined by

$$\mathcal{B}_{g,k} = \{\tau_i \in \mathcal{T}_g : \ell_{g,k-1} < T_i \le \ell_{g,k}\}, \qquad k = 1, \dots, K, \tag{6}$$

where $0 = \ell_{g,0} \le \cdots \le \ell_{g,K} = \infty$ are length boundaries determined from that game's training trajectories. We use $K = 4$ buckets and set the three internal boundaries to the empirical 25th, 50th, and 75th percentiles of trajectory length within each game.

Training proceeds in three stages $r \in \{0, 1, 2\}$, each covering one third of the total optimization steps. For each game g, we select a nonempty bucket with probability $p_{g,k}^{(r)}$ and then sample a trajectory uniformly from that bucket. Let $k_g^\star$ index the shortest nonempty bucket. The bucket sampling probability is

$$p_{g,k}^{(r)} = (1 - \beta_r)\mathbf{1}[k = k_g^\star] + \beta_r \frac{|\mathcal{B}_{g,k}|}{|\mathcal{T}_g|}, \tag{7}$$

where $\mathbf{1}[\cdot]$ is the indicator function and $\beta_r = 0.2 + 0.4r$, giving mixing coefficients of 0.2, 0.6, and 1.0. Empty buckets receive zero probability. The initial stage allocates 80% of the probability to the shortest nonempty bucket and distributes the remaining 20% in proportion to bucket sizes. Subsequent stages reduce this preference for short trajectories, reaching uniform trajectory sampling within each game in the final stage. Each decision step in a selected trajectory forms a training sample with its original input context and target output. We combine these samples from all games for supervised training.

Action class loss shaping. To address the predominance of <noop> steps in action supervision, we increase the loss weights of action tokens for less frequent device operations and task completion signals. We classify each decision step as $c \in \{\mathrm{noop}, \mathrm{action}, \mathrm{complete}\}$, where action denotes a step containing device operations and complete denotes a step that emits <complete>. Let m_j denote the segment containing output token y_j. We define its loss weight as

$$w_j = \begin{cases} \alpha_c, & m_j = \mathrm{act}, \\ 1, & \text{otherwise}, \end{cases} \qquad (\alpha_{\mathrm{noop}}, \alpha_{\mathrm{action}}, \alpha_{\mathrm{complete}}) = (1, 2, 3). \tag{8}$$

This weighting emphasizes device control and task termination while retaining unit weights for rationale, goal, and expected outcome tokens.

At curriculum stage r, we train on batches $\mathcal{M}_r$ of decision samples using the weighted token cross-entropy normalized by the total token weight

$$\mathcal{L}(\theta; \mathcal{M}_r) = -\frac{\sum_{b \in \mathcal{M}_r} \sum_{j=1}^{|y_b|} w_{b,j} \log p_\theta(y_{b,j} \mid x_b, y_{b,<j})}{\sum_{b \in \mathcal{M}_r} \sum_{j=1}^{|y_b|} w_{b,j}}. \tag{9}$$

5 EXPERIMENTS

We evaluate PLAYAGENT through online gameplay, where each agent plays the games directly from screenshots and its performance is verified by task evaluators. Section 5.1 describes the evaluation environment, the baselines, and the implementation details.

5.1 EXPERIMENTAL SETUP

Games and tasks. We evaluate all agents online on the games included in LETSPLAY. These games span five genres, namely arcade, platformer, puzzle, runner, and simulation games. Each game provides five tasks, giving 125 tasks in total. Each task specifies a natural language objective, such as clearing a level or reaching a target score, together with a maximum number of control steps. Each game runs in a headless Chromium browser with a viewport of 1280×800 pixels. The agent perceives the game only through screenshots of this viewport and controls it only through keyboard and mouse inputs.

Evaluation metric. We report the task success rate. Each task is attempted in 3 independent episodes, and each episode resets the game with a different seed, so that the agent starts from a different game state in each attempt. An episode ends when the agent emits `<complete>`, when the game reaches a terminal state, or when the step budget of the task is exhausted. The task evaluator then checks the final game state against the task objective. An episode is successful when the objective is met and the game has not ended in a loss. A task counts as completed when at least one of its episodes is successful. The success rate of a genre is the fraction of completed tasks among all tasks in that genre. The overall success rate is the fraction of completed tasks among all 125 tasks.

Baselines. We compare PLAYAGENT with three vision-language models, i.e., Qwen-3.7-Plus, Qwen-3.8-Max, and Qwen3-VL-8B-Instruct (Bai et al., 2025). Qwen3-VL-8B-Instruct is the base model of PLAYAGENT, so the comparison with it directly measures the effect of training on LETSPLAY. All baselines use the same system prompt, input device state, history windows, and action interface as PLAYAGENT (Appendix B), and follow the same evaluation protocol.

Implementation details. We build PLAYAGENT on Qwen3-VL-8B-Instruct (Bai et al., 2025) and fine-tune it with LlamaFactory (Zheng et al., 2024). We freeze the vision encoder and fully fine-tune all parameters of the multimodal projector and the language backbone. The history windows contain 8 recent steps and 8 long-term steps, and all screenshots are resized to 640×400 pixels. Training uses the action class loss weights in Equation 8 and runs for 3 epochs on 8 NVIDIA H200 GPUs with an effective batch size of 64 and a learning rate of 5×10^{-5}. At inference, the agent uses greedy decoding and stops generation at `</act>`. The complete dataset and source code will be publicly released upon acceptance.

5.2 MAIN RESULTS

Table 3 reports the task success rates of all agents in each game genre and over all tasks.

Overall performance. PLAYAGENT achieves the highest overall success rate of 36.0%, which is 2.4% higher than the strongest baseline, Qwen-3.8-Max (33.6%). Compared with its base model Qwen3-VL-8B-Instruct, PLAYAGENT improves the overall success rate from 11.2% to 36.0%, a gain of 24.8%. This result shows that the trajectories and step annotations in LETSPLAY teach the model to ground task objectives in game screens and to translate them into executable device actions.

Table 3: **Task Success Rates across Game Genres.** Each task is attempted in 3 episodes with a different game seed and counts as completed when at least one episode is successful.

Agent	Arcade	Platformer	Puzzle	Runner	Simulation	Overall
Proprietary models						
Qwen-3.7-Plus	20.0% (3 / 15)	8.0% (2 / 25)	14.3% (5 / 35)	7.5% (3 / 40)	0.0% (0 / 10)	10.4% (13 / 125)
Qwen-3.8-Max	**53.3% (8 / 15)**	**24.0% (6 / 25)**	48.6% (17 / 35)	17.5% (7 / 40)	**40.0% (4 / 10)**	33.6% (42 / 125)
Open-source models						
Qwen3-VL-8B-Instruct	6.7% (1 / 15)	12.0% (3 / 25)	14.3% (5 / 35)	12.5% (5 / 40)	0.0% (0 / 10)	11.2% (14 / 125)
PLAYAGENT (ours)	**53.3% (8 / 15)**	16.0% (4 / 25)	**54.3% (19 / 35)**	**35.0% (14 / 40)**	0.0% (0 / 10)	**36.0% (45 / 125)**

It also suggests that LETSPLAY provides an effective cold start for training game agents, on which subsequent reinforcement learning can build.

Performance across genres. Training on LETSPLAY brings the largest gains on arcade, puzzle, and runner games. Compared with its base model, PLAYAGENT raises the success rate from 6.7% to 53.3% on arcade games, from 14.3% to 54.3% on puzzle games, and from 12.5% to 35.0% on runner games. The advantage is most pronounced on runner games, where PLAYAGENT reaches 35.0%, twice the success rate of the strongest baseline in this genre, Qwen-3.8-Max (17.5%). These games require the agent to react to approaching obstacles at the right moment. The recent history window of PLAYAGENT captures the motion of objects across consecutive frames, and the successful trajectories in LETSPLAY teach the model when to act. In contrast, platformer and simulation games remain the most challenging genres for PLAYAGENT, with success rates of 16.0% and 0.0%, while Qwen-3.8-Max achieves the best results in these genres with 24.0% and 40.0%. These games require long sequences of precise movements or navigation through large scenes, where an early mistake often cannot be recovered later in the episode.

6 DISCUSSION

Conclusion. In this paper, we introduce LETSPLAY, a gameplay trajectory dataset built on GameWorld for training visual game agents. LETSPLAY combines task-directed trajectories from game-specific experts and human players, pairing screenshots with device actions in a unified representation and annotating each step with the action's rationale, immediate goal, and expected outcome. These annotations connect visual observations and device actions with task objectives, providing structured supervision for agent training. Building on LETSPLAY, we develop PLAYAGENT, a visual game agent that decides actions based on the current game screenshot, input device state, and interaction history. We train this agent using a curriculum that groups trajectories into length-based buckets and progressively introduces longer trajectories, together with action class loss shaping that emphasizes less frequent device control actions and termination decisions. Experiments show that PLAYAGENT achieves a task success rate of 36.0%, outperforming baseline methods.

Limitations. First, LETSPLAY is limited to the games and tasks in GameWorld, and generalization to unseen games and control interfaces remains to be evaluated. Game-specific expert heuristics may also limit behavioral diversity. Nevertheless, its paired visual observations, device actions, and language annotations provide rich supervision for the initial training of visual game agents. Second, the agent's bounded history may omit events needed for long-horizon decisions, while the unified action representation remains sensitive to execution timing. Future work can broaden game and trajectory coverage and explore adaptive memory and timing-aware control.

7 AI USE STATEMENT

We used generative AI tools only to proofread and refine the language of this manuscript. As part of our method, Qwen-3.8-Max generates the step annotations in LETSPLAY (Section 3.4). All research ideas, experimental designs, and final content were conceived, verified, and approved by the authors, who take full responsibility for this paper.

REFERENCES

Jaewoo Ahn, Junseo Kim, Heeseung Yun, Jaehyeon Son, Dongmin Park, Jaewoong Cho, and Gunhee Kim. FlashAdventure: A benchmark for GUI agents solving full story arcs in diverse adventure games. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 23354–23384, 2025. doi: 10.18653/v1/2025.emnlp-main.1192. URL https://aclanthology.org/2025.emnlp-main.1192/.

Shuai Bai et al. Qwen3-VL technical report. *arXiv preprint arXiv:2511.21631*, 2025. URL https://arxiv.org/abs/2511.21631.

Kevin Black, Noah Brown, Danny Driess, Adnan Esmail, Michael Equi, Chelsea Finn, Niccolo Fusai, Lachy Groom, Karol Hausman, Brian Ichter, et al. π_0: A Vision-Language-Action Flow Model for General Robot Control. *arXiv preprint arXiv:2410.24164*, 2024. URL https://arxiv.org/abs/2410.24164.

Anthony Brohan, Noah Brown, Justice Carbajal, Yevgen Chebotar, Xi Chen, Krzysztof Choromanski, Tianli Ding, Danny Driess, Avinava Dubey, Chelsea Finn, et al. RT-2: Vision-Language-Action Models Transfer Web Knowledge to Robotic Control. *arXiv preprint arXiv:2307.15818*, 2023. URL https://arxiv.org/abs/2307.15818.

Anthony Costarelli, Mat Allen, Roman Hauksson, Grace Sodunke, Suhas Hariharan, Carlson Cheng, Wenjie Li, Joshua Clymer, and Arjun Yadav. GameBench: Evaluating Strategic Reasoning Abilities of LLM Agents. *arXiv preprint arXiv:2406.06613*, 2024. URL https://arxiv.org/abs/2406.06613.

William H Guss, Brandon Houghton, Nicholay Topin, Phillip Wang, Cayden Codel, Manuela Veloso, and Ruslan Salakhutdinov. Minerl: A large-scale dataset of minecraft demonstrations. *arXiv preprint arXiv:1907.13440*, 2019.

Lanxiang Hu, Mingjia Huo, Yuxuan Zhang, Haoyang Yu, Eric P. Xing, Ion Stoica, Tajana Rosing, Haojian Jin, and Hao Zhang. LMGame-Bench: How good are LLMs at playing games? In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=qeziG97WUZ.

Jen-tse Huang, Eric John Li, Man Ho Lam, Tian Liang, Wenxuan Wang, Youliang Yuan, Wenxiang Jiao, Xing Wang, Zhaopeng Tu, and Michael R. Lyu. How Far Are We on the Decision-Making of LLMs? Evaluating LLMs' Gaming Ability in Multi-Agent Environments. *arXiv preprint arXiv:2403.11807*, 2024. URL https://arxiv.org/abs/2403.11807.

Moo Jin Kim, Karl Pertsch, Siddharth Karamcheti, Ted Xiao, Ashwin Balakrishna, Suraj Nair, Rafael Rafailov, Ethan Foster, Grace Lam, Pannag Sanketi, Quan Vuong, Thomas Kollar, Benjamin Burchfiel, Russ Tedrake, Dorsa Sadigh, Sergey Levine, Percy Liang, and Chelsea Finn. OpenVLA: An Open-Source Vision-Language-Action Model. *arXiv preprint arXiv:2406.09246*, 2024. URL https://arxiv.org/abs/2406.09246.

Muyao Li, Zihao Wang, Kaichen He, Xiaojian Ma, and Yitao Liang. JARVIS-VLA: Post-Training Large-Scale Vision Language Models to Play Visual Games with Keyboards and Mouse. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 17878–17899, 2025. doi: 10.18653/v1/2025.findings-acl.920. URL https://aclanthology.org/2025.findings-acl.920/.

Loïc Magne, Anas Awadalla, Guanzhi Wang, Yinzhen Xu, Joshua Belofsky, Fengyuan Hu, Joohwan Kim, Ludwig Schmidt, Georgia Gkioxari, Jan Kautz, Yisong Yue, Yejin Choi, Yuke Zhu, and Linxi "Jim" Fan. NitroGen: An Open Foundation Model for Generalist Gaming Agents. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 21511–21521, 2026. URL https://arxiv.org/abs/2601.02427.

Mingyu Ouyang, Siyuan Hu, Kevin Qinghong Lin, Hwee Tou Ng, and Mike Zheng Shou. GameWorld: Towards standardized and verifiable evaluation of multimodal game agents. *arXiv preprint arXiv:2604.07429*, 2026. URL https://arxiv.org/abs/2604.07429.

Davide Paglieri, Bartłomiej Cupiał, Samuel Coward, Ulyana Piterbarg, Maciej Wolczyk, Akbir Khan, Eduardo Pignatelli, Łukasz Kuciński, Lerrel Pinto, Rob Fergus, Jakob Nicolaus Foerster, Jack Parker-Holder, and Tim Rocktäschel. BALROG: Benchmarking Agentic LLM and VLM Reasoning On Games. In *International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=fp6t3F669F.

Dongmin Park, Minkyu Kim, Beongjun Choi, Junhyuck Kim, Keon Lee, Jonghyun Lee, Inkyu Park, Byeong-Uk Lee, Jaeyoung Hwang, Jaewoo Ahn, Ameya S. Mahabaleshwarkar, Bilal Kartal, Pritam Biswas, Yoshi Suhara, Kangwook Lee, and Jaewoong Cho. Orak: A foundational benchmark for training and evaluating LLM agents on diverse video games. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=H1ncX6O6Yh.

SIMA Team. Scaling Instructable Agents Across Many Simulated Worlds. *arXiv preprint arXiv:2404.10179*, 2024. URL https://arxiv.org/abs/2404.10179.

SIMA Team. SIMA 2: A Generalist Embodied Agent for Virtual Worlds. *arXiv preprint arXiv:2512.04797*, 2025. URL https://arxiv.org/abs/2512.04797.

Weihao Tan, Wentao Zhang, Xinrun Xu, Haochong Xia, Ziluo Ding, Boyu Li, Bohan Zhou, Junpeng Yue, Jiechuan Jiang, Yewen Li, et al. Cradle: Empowering Foundation Agents Towards General Computer Control. *arXiv preprint arXiv:2403.03186*, 2024. URL https://arxiv.org/abs/2403.03186.

Weihao Tan, Xiangyang Li, Yunhao Fang, Heyuan Yao, Shi Yan, Hao Luo, Tenglong Ao, Huihui Li, Hongbin Ren, Bairen Yi, Yujia Qin, Bo An, Libin Liu, and Guang Shi. Lumine: An Open Recipe for Building Generalist Agents in 3D Open Worlds. *arXiv preprint arXiv:2511.08892*, 2025. URL https://arxiv.org/abs/2511.08892.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An Open-Ended Embodied Agent with Large Language Models. *arXiv preprint arXiv:2305.16291*, 2023. URL https://arxiv.org/abs/2305.16291.

Xinyu Wang, Bohan Zhuang, and Qi Wu. Are Large Vision Language Models Good Game Players? *arXiv preprint arXiv:2503.02358*, 2025a. URL https://arxiv.org/abs/2503.02358.

Zihao Wang, Xujing Li, Yining Ye, Junjie Fang, Haoming Wang, Longxiang Liu, Shihao Liang, Junting Lu, Zhiyong Wu, Jiazhan Feng, et al. Game-TARS: Pretrained Foundation Models for Scalable Generalist Multimodal Game Agents. *arXiv preprint arXiv:2510.23691*, 2025b. URL https://arxiv.org/abs/2510.23691.

Yue Wu, Xuan Tang, Tom M. Mitchell, and Yuanzhi Li. SmartPlay: A Benchmark for LLMs as Intelligent Agents. *arXiv preprint arXiv:2310.01557*, 2023. URL https://arxiv.org/abs/2310.01557.

Yuguang Yue, Irakli Salia, Samuel Hunt, Chris Green, Wenzhe Shi, and Jonathan J. Hunt. Scaling behavior cloning improves causal reasoning: An open model for real-time video game playing. *arXiv preprint arXiv:2601.04575*, 2026. URL https://arxiv.org/abs/2601.04575.

Alex L. Zhang, Thomas L. Griffiths, Karthik R. Narasimhan, and Ofir Press. VideoGameBench: Can Vision-Language Models complete popular video games? *arXiv preprint arXiv:2505.18134*, 2025a. URL https://arxiv.org/abs/2505.18134.

Haoran Zhang, Chenhao Zhu, Sicong Guo, Hanzhe Guo, Haiming Li, and Donglin Yu. StarBench: A turn-based RPG benchmark for agentic multimodal decision-making and information seeking. *arXiv preprint arXiv:2510.18483*, 2025b. URL https://arxiv.org/abs/2510.18483.

Xiangxi Zheng, Linjie Li, Zhengyuan Yang, Ping Yu, Alex Jinpeng Wang, Rui Yan, Yuan Yao, and Lijuan Wang. V-MAGE: A Game Evaluation Framework for Assessing Vision-Centric Capabilities in Multimodal Large Language Models. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 17707–17758, 2026. doi: 10.18653/v1/2026.findings-acl.878. URL https://aclanthology.org/2026.findings-acl.878/.

Xinyue Zheng, Haowei Lin, Kaichen He, Zihao Wang, Qiang Fu, Haobo Fu, Zilong Zheng, and Yitao Liang. MCU: An evaluation framework for open-ended game agents. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 78221–78259, 2025. URL https://proceedings.mlr.press/v267/zheng25j.html.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. LlamaFactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pp. 400–410, 2024. URL https://aclanthology.org/2024.acl-demos.38.

Xizhou Zhu, Yuntao Chen, Hao Tian, Chenxin Tao, Weijie Su, Chenyu Yang, Gao Huang, Bin Li, Lewei Lu, Xiaogang Wang, Yu Qiao, Zhaoxiang Zhang, and Jifeng Dai. Ghost in the Minecraft: Generally Capable Agents for Open-World Environments via Large Language Models with Text-based Knowledge and Memory. *arXiv preprint arXiv:2305.17144*, 2023. URL https://arxiv.org/abs/2305.17144.

A GAME DESCRIPTIONS

This section briefly describes the 25 games in LETSPLAY, grouped by the five genres in Table 2. The games and their genres follow GameWorld (Ouyang et al., 2026). Each game is played through keyboard and mouse inputs, and each game is associated with five tasks that specify concrete gameplay objectives.

Arcade. Arcade games require fast closed loop control with frequent reactions to moving entities. In `Core Ball`, the player presses `Space` to shoot numbered balls into a rotating core, and a level is cleared once all balls are attached without any two balls colliding. `Google Snake` is a version of the classic snake game, in which the player steers a continuously moving snake to eat apples and grow longer while avoiding the walls and its own body. In `Pac-Man`, the player guides Pac-Man through a maze to eat all pellets and avoid the ghosts, which become edible for a short time after Pac-Man eats a power pellet.

Platformer. Platformer games require precise movement and jumping across structured terrain while avoiding hazards. In `Another Gentleman's Adventure`, the player moves a gentleman across platforms to collect coins, hits question blocks from below to reveal items, and defeats enemies by jumping on them. `Captain Callisto` asks the player to reach the exit of each area by running and jumping across platforms, and collected fuel powers a jetpack that lifts the character upward. In `Doodle Jump`, the character keeps bouncing upward and the player steers it left or right to land on higher platforms, some of which move, break, or vanish after one use. `GeoDash` is an automatically running game in the style of Geometry Dash, where the only control is a jump that must be timed precisely to clear spikes and gaps. `Mario Game` follows Super Mario Bros., in which the player runs and jumps through side scrolling levels to reach the flagpole, stomps on enemies, and collects coins and power ups from question blocks.

Puzzle. Puzzle games emphasize reasoning over discrete states and planning over long horizons. In `2048`, each arrow key slides all tiles on a 4×4 board in one direction and merges two tiles of the same value into their sum, and the player aims to build larger tiles before the board fills up. `Astray` is a maze in which the player rolls a ball with the arrow keys to find the path to the exit. `Hextris` surrounds a central hexagon with falling colored blocks, and the player rotates the hexagon so that three or more blocks of the same color meet and clear before the stacks overflow its boundary. In `Minesweeper`, the player reveals cells with left clicks and flags mines with right clicks, using the number in each revealed cell, which counts its adjacent mines, to locate every mine without clicking on one. `Stack` asks the player to drop a block that slides back and forth above a tower, and any overhanging part is cut off, so imprecise timing shrinks the next block until no overlap remains. In `Tetris`, the player moves, rotates, and drops falling pieces to complete and clear horizontal lines before the stack reaches the top. `Wordle` requires the player to guess a hidden word of five letters within six tries, and the color of each letter in a guess indicates whether it is in the correct position, elsewhere in the word, or absent.

Runner. Runner games advance the scene automatically and require timely reactions to approaching obstacles. In `Boxel Rebound`, the character moves forward on its own and the player can only jump, which must be timed to avoid spikes and pits until the end of the level. `Chrome Dino` is the endless runner of the Chrome browser, where the dinosaur jumps over cacti and ducks under birds flying low as its speed increases. `Cubefield` places the player in a triangular ship that flies forward through a 3D field of cubes, and the player steers left or right because hitting any cube ends the run. In `Edge Surf`, the player steers a surfer to avoid obstacles and collect items, where boosts grant temporary speed, shields grant temporary invincibility, and each collision costs one of three lives. `Flappy Bird` is controlled by a single button, where each flap gives the bird an upward impulse against gravity and the player times the flaps to pass through the gaps between pipes. In `NS-Shaft`, the character descends through a shaft of shifting platforms, and the player moves it left or right to land safely, since touching hazards reduces its health and falling to the bottom ends the game. `Run 3` takes place in a tunnel in space, where the player strafes and jumps to cross gaps, and moving onto a side wall rotates the tunnel so that the wall becomes the new floor. `Temple Run 2` is a 3D endless runner at high speed, in which the player switches lanes, turns at intersections, jumps over obstacles, and slides under them, and any missed action ends the run.

Simulation. Simulation games feature open environments with multiple concurrent objectives. In `Monkey Mart`, the player controls a monkey that runs a supermarket by harvesting bananas and corn, stocking shelves, and serving customers at the counter, and the earned money unlocks new stations and upgrades. `Wolf3D` is the first person shooter Wolfenstein 3D, in which the player navigates a 3D maze, opens doors, and shoots guards while managing health and ammunition.

B AGENT PROMPTS

This section presents the prompts through which PLAYAGENT interacts with the games, as introduced in Section 4.1. Each decision step is formulated as a single conversational turn consisting of a system prompt, an instruction prompt, and the model response. The system prompt carries the task context D and remains fixed throughout an episode. The instruction prompt is rebuilt at every step from the two history windows, the input device state, and the current screenshot. The response contains the four target segments in Equation 5. We construct the training samples and the online evaluation requests with the same prompt builders, so that PLAYAGENT receives identical prompt text during training and inference. All baselines in Section 5.1 are evaluated with the same prompts.

B.1 SYSTEM PROMPT

The system prompt is instantiated from the following template.

```
# GUI Game Agent

You are an autonomous agent that plays a GUI game by driving its keyboard
     and mouse, exactly as a human player would. At each step you receive
     the current screenshot and the input state that reports which keys
    and buttons are physically held right now. Your job is to decide the
    next action that best advances the task objective.

## Game

{game}

## Task

{task_prompt}

## Game Rules and Controls

{guidance}

{action_reference}

{response_format}
```

The placeholders {game} and {task_prompt} are filled with the game identifier and the natural language objective of the current task, respectively. The placeholder {guidance} is filled with a game-specific guide that describes the game objective, game rules, and player controls, and enumerates the legal inputs of the game. The placeholders {action_reference} and {response_format} are shared across all games and specify the action space and the output format, respectively. For example, the system prompt of a `Flappy Bird` task instantiates the game-dependent placeholders as follows.

```
## Game

flappy-bird

## Task

Pass at least 3 pipes in one run.

## Game Rules and Controls

#### Flappy Bird

You are playing **Flappy Bird**, a one-button flying game. The bird
    travels
through a sequence of vertical green pipes while gravity continuously
    changes
its altitude.

##### Game Objective

- Guide the bird through the gap in every pipe pair without touching a
    pipe,
  the ground, or the upper boundary.
- Each pipe pair passed adds one point. The run ends on the first
    collision.

##### Game Rules

- A flap gives the bird an immediate upward velocity impulse; it is not a
  sustained lift. Between flaps, gravity pulls the bird downward and its
  vertical speed changes continuously.
- Pipe gaps vary in height and position. A flap that is too early can
    leave the
  bird above the gap, while a late flap may not provide enough altitude.
- The title screen starts on the first input. After a collision the score
  screen is shown; Space or a click starts another run.

##### Player Controls

You control only the bird's vertical impulse. Prefer single edge presses
    and
  short wait frames so the next observation shows the resulting altitude.

ACTION SPACE (ONLY LEGAL ACTIONS):

- Space: Flap upward; on the title/score screen it starts or restarts the
     run.
- Mouse left click: Flap in play, or start the run from the title screen.
- Wait: Do not flap and let gravity pull the bird downward.
```

The action reference specifies the unified action representation described in Section 3.3. It defines the standalone control tokens, the syntax of the four device channels, the coordinate encoding, the persistence of held inputs, and the canonical token order, and concludes with examples of complete action blocks.

```
## Available Actions
```

```
Every step you emit actions wrapped in a single `<act> ... </act>`.

### Action Shape

An action is either a single standalone control token or one or more
    device channels combined together.

| Token | Meaning |
| --- | --- |
| `<noop>` | Advance one step without changing any held input. Must stand
     alone. |
| `<complete>` | End the episode. Should be used only once the task
    objective is met. Must stand alone. |

`<noop>` and `<complete>` never carry device channels. Every other action
     is built from the four channels below. Include only the channels you
     need. The channels always appear in a fixed order. Keyboard comes
    first, then mouse, then move, then scroll.

### Keyboard Channel

| Syntax | Meaning |
| --- | --- |
| `<kb_hold> <key_NAME> ...` | Press the listed keys down and keep them
    held across later steps until released. |
| `<kb_tap> <key_NAME> ...` | Tap the listed keys once (down, then up);
    does not change the held set. |
| `<kb_release> <key_NAME> ...` | Release the listed keys. |
| `<kb_text>text</kb_text>` | Type the literal text character by
    character. |

Each key is a `<key_NAME>` token, e.g. `<key_w>`, `<key_1>`, `<key_Space
    >`, `<key_Enter>`, `<key_Escape>`, `<key_Shift>`, `<key_ArrowUp>`.
    Key names follow the game controls. A given key may appear in only
    one of `<kb_hold>`, `<kb_tap>` or `<kb_release>` within the same
    action.

### Mouse Channel

| Syntax | Meaning |
| --- | --- |
| `<ms_hold> <btn_NAME> ...` | Press the listed buttons down and keep
    them held until released. |
| `<ms_tap> <btn_NAME> ...` | Click the listed buttons once (down, then
    up); does not change the held set. |
| `<ms_release> <btn_NAME> ...` | Release the listed buttons. |
| `<drag> x1 y1 x2 y2` | Press the left button at `(x1, y1)`, glide to `(
    x2, y2)`, and release, all within this step. |

Button names are `<btn_left>`, `<btn_right>` and `<btn_middle>`. A given
    button may appear in only one of `<ms_hold>`, `<ms_tap>` or `<
    ms_release>`.

### Move Channel

| Syntax | Meaning |
| --- | --- |
| `<move_abs> x y` | Move the cursor to an absolute screen position. |
| `<move_rel> dx dy` | Move or turn by a displacement relative to the
    current position. |

Use `<move_abs>` for point-and-click targets such as buttons, menus,
    icons and board cells. Use `<move_rel>` for pointer-lock camera
```

```
    control such as aiming or looking around. Use at most one of the two
    per action.

### Scroll Channel

| Syntax | Meaning |
| --- | --- |
| `<scroll> dx dy` | Scroll the mouse wheel by a signed displacement. |

Positive `dy` scrolls down and negative `dy` scrolls up; positive `dx`
    scrolls right and negative `dx` scrolls left.

### Coordinates

Every position and displacement is a coordinate token from `<bin_000>` to
     `<bin_1000>`. One bin is 1/1000 of the current image width on the x
    axis and 1/1000 of its height on the y axis.

Absolute channels (`<move_abs>` and `<drag>`) give a position measured
    from the top-left corner of the image.

- On the x axis, `<bin_000>` is the left edge and `<bin_1000>` is the
    right edge. The value increases from left to right.
- On the y axis, `<bin_000>` is the top edge and `<bin_1000>` is the
    bottom edge. The value increases from top to bottom.
- Therefore `<bin_000> <bin_000>` is the top-left corner, `<bin_1000> <
    bin_1000>` is the bottom-right corner, and `<bin_500> <bin_500>` is
    the center of the image.

Signed channels (`<move_rel>` and `<scroll>`) give a displacement
    measured from `<bin_500>`, which means no movement.

- A value below `<bin_500>` moves or scrolls left on the x axis, and a
    value above `<bin_500>` moves or scrolls right.
- A value below `<bin_500>` moves or scrolls up on the y axis, and a
    value above `<bin_500>` moves or scrolls down.
- The displacement can reach half an image dimension in either direction.

### Held State

- A hold stays down across later steps until you release it. Inputs you
    do not mention keep their current state.
- A tap is a one-off event and never changes what is held.
- To hold something for a while, hold it once, wait for as many steps as
    you want, then release it.

### Canonical Order

Every action has exactly one valid serialization. Order the tokens by
    these rules.

1. Channels come first in this order: keyboard, then mouse, then move,
    then scroll.
2. Inside a channel the operations come in this order: hold, then tap,
    then release, then text or drag.
3. Each structure token appears at most once.
4. The names after a structure token are listed once each in ASCII order:
     digits first, then capitalized names such as `<key_ArrowUp>` `<
    key_Shift>` `<key_Space>`, then lowercase letters.

### Examples

| Intent | Action |
| --- | --- |
| Wait one step, keeping every held input | `<act> <noop> </act>` |
```

```
| Start running forward while jumping once | `<act> <kb_hold> <key_Shift>
    <key_w> <kb_tap> <key_Space> </act>` |
| Aim with the right button held and fire a single shot | `<act> <ms_hold
   > <btn_right> <ms_tap> <btn_left> </act>` |
| Move the cursor to a button and click it | `<act> <ms_tap> <btn_left> <
   move_abs> <bin_842> <bin_512> </act>` |
| Turn the camera right and slightly up | `<act> <move_rel> <bin_640> <
   bin_480> </act>` |
| Drag an item across the board | `<act> <drag> <bin_200> <bin_300> <
   bin_700> <bin_300> </act>` |
| Type a word into a text field | `<act> <kb_text>crane</kb_text> </act>`
    |
| Combine several channels in one step | `<act> <kb_hold> <key_Shift> <
   key_d> <kb_tap> <key_Space> <ms_tap> <btn_left> <move_abs> <bin_516>
   <bin_275> </act>` |
| Finish once the objective is met | `<act> <complete> </act>` |
```

The response format that concludes the system prompt is presented in Appendix B.3.

B.2 Instruction Prompt

The instruction prompt forms the user message of each decision step. Together with the task context D in the system prompt, it realizes the model input x_t in Equation 4. It consists of the long-term window H_t^{long}, the recent window H_t^{recent}, and a decision instruction, in that order. Each window begins with a heading and a short description of its role, followed by its entries in chronological order. Each entry h_j is introduced by a step header, followed by the screenshot o_j, the immediate goal y_j^{g}, and the action a_j of step j. The rationale and expected outcome of earlier steps are not included in the history. The decision instruction then reports the input device state s_t^{in} and precedes the current screenshot o_t. The complete template is shown below, where `[screenshot of step j]` marks the position at which the corresponding image is inserted.

```
# Long-term History

This section holds earlier meaningful steps from before the recent window
    , keeping only actions that changed the game state. It summarizes
    what has already been accomplished and gives you the long-range
    context behind the current situation.
Entries are listed oldest first.

## Step {j}

[screenshot of step j]
<goal>{goal of step j}</goal><act> {action tokens of step j} </act>

...(one entry per step in the long-term window)

# Recent History

This section holds the most recent consecutive steps leading up to now,
    so you can see exactly what just happened. It provides the immediate
    context that connects your latest actions to the current screen.
Entries are listed oldest first.

## Step {j}

[screenshot of step j]
<goal>{goal of step j}</goal><act> {action tokens of step j} </act>

...(one entry per step in the recent window)

# Decision

Choose the next action that best advances the task objective.
```

````
## Current Input State

```json
{input_state}
```

## Current Screen (step {t})

[screenshot of step t]
````

When a window contains no entries, as at the beginning of an episode, its entries are replaced by the sentence "No steps have been recorded for this part of the history yet." The placeholder {input_state} is filled with a JSON object that reports the cursor position, the input focus, the held keys, the held mouse buttons, and the active touch contacts. For example, at the first step of the Flappy Bird task above, no input is held and the input state is

```
{"cursor": {"x": 640, "y": 400}, "focus": "game", "keyboard": {"held":
    []}, "mouse": {"held": []}, "touch": {"active_contacts": []}}
```

B.3 Response Format

The system prompt concludes with the following response format, which requires the model to generate the rationale, immediate goal, action, and expected outcome in the order of Equation 5.

````
## Response Format

Your reply has four sections. `<think>` holds a brief analysis of the
    observation and history, `<goal>` states the current objective, `<act
    >` carries the Action tokens, and `<eo>` describes the immediate
    expected result. Emit them in this exact order, with no JSON,
    headings or Markdown fences:

```text
<think>Brief analysis of the observation and history.</think>
<goal>One immediate objective.</goal>
<act>Action-DSL tokens.</act>
<eo>One immediate expected result.</eo>
```
````

During training, the target response contains all four segments and is followed by an end-of-turn token. At inference, decoding stops at </act>, and the tokens inside the <act> block are parsed into device operations for execution, as described in Section 4.1. The following two target responses are taken from a successful trajectory of the Flappy Bird task above. In the first response, the agent taps Space to lift the bird toward the gap of the approaching pipe pair. In the second response, the agent emits <complete> after the score reaches the target of three pipes. Line breaks between segments are added for readability.

```
<think>The bird is at roughly y=420 and sinking toward the bottom pipe of
     the first pair, whose gap spans about y=300 to y=440 just ahead at x
    =380; a single flap now gives the upward impulse needed to center on
    the gap before reaching it.</think>
<goal>Flap to rise into the gap of the approaching first pipe pair</goal>
<act> <kb_tap> <key_Space> </act>
<eo>The bird gets an upward velocity boost and climbs toward the middle
    of the pipe gap as the pipes draw nearer.</eo>
```

```
<think>The score counter on screen now reads 3, meaning the bird has
    already cleared three pipe pairs in this single run without colliding
    , which fulfills the task objective of passing at least 3 pipes.</
    think>
```

```
<goal>Finish the episode after passing 3 pipes in one run</goal>
<act> <complete> </act>
<eo>The episode ends with the objective satisfied at a score of 3.</eo>
```

C TRAJECTORY ANNOTATION PROMPT

This section presents the prompt used to annotate each trajectory step with a rationale, a goal, and an expected outcome, as described in Section 3.4. Annotation operates on the recorded trajectories before their actions are converted into the unified action representation of Section 3.3. In these recordings, each action is stored as a JSON object that groups device operations by channel, with coordinates given in screenshot pixels. The annotation prompt therefore describes actions in this JSON form rather than with action tokens. Each annotation request consists of a system prompt and a user message that presents the step to be annotated.

C.1 ANNOTATION SYSTEM PROMPT

The system prompt consists of a game prompt followed by annotation instructions. The game prompt describes how to play the current game and how to express actions, and is organized into seven sections.

- **Role** describes the model as a computer-use agent that plays games. It asks the model to inspect the play area, the heads-up display, and any pop-ups on the screenshot, and to combine them with the reported input state and recent actions when choosing the next action.
- **Coordinate System** states the screenshot resolution of 1280×800 pixels and defines absolute positions and relative displacements in pixels, measured from the upper left corner.
- **Action Space** defines an action as a JSON object with four optional device channels, i.e., `keyboard`, `mouse`, `move`, and `scroll`, and a `complete` flag. These channels correspond to the keyboard, mouse button, pointer movement, and scrolling channels of Section 3.3. An empty object $\{\}$ advances the game by one step without changing any input, which corresponds to `<noop>`. This section also explains that held inputs persist until released and specifies how the environment resolves redundant or conflicting requests.
- **Response Format** specifies the JSON reply of a decision step and defines the `goal`, `thought`, and `expected_outcome` fields, which the annotation instructions refer to. We reproduce it below.
- **Examples** provides seven worked replies, covering point-and-click interaction, multi-channel aiming and firing, waiting, holding one key while tapping another, holding a button across steps, firing while aiming with a held button, and task completion.
- **Game-specific Guidance** contains the same game guide as the agent system prompt in Appendix B.1.
- **Task Objective** states the objective of the current task and the completion rules, as reproduced below.

```
### Response Format

Your entire response must consist of exactly one valid JSON object
    representing a single `Action`.

Output nothing except the JSON object. Do not include explanatory prose,
    Markdown code fences, comments, headings, or text before or after the
     object.

Every field is optional unless required by a condition described below.
    Include only the fields needed for the current step.
Do not output placeholder values such as `"..."`, literal `OR`
    expressions, comments, or trailing commas. Do not output any duration
    , time or wait field: the game time per step is fixed by the
    environment.
```

````
The `Action` may contain the following fields:

```text
{
  "keyboard": <keyboard command>,
  "mouse": <mouse command>,
  "move": <cursor movement command>,
  "scroll": <scroll command>,
  "complete": <boolean>,
  "reason": <string>,
  "goal": <string>,
  "thought": <string>,
  "expected_outcome": <string>
}
```

#### Allowed Field Forms

- `keyboard` is an object with any subset of these fields:

  - `"hold": ["w", ...]`
  - `"tap": ["w", ...]`
  - `"release": ["w", ...]`
  - `"text": "..."`

  A key must not appear in more than one of `hold`, `tap` and `release`.

- `mouse` is an object with any subset of these fields:

  - `"hold": ["left", ...]`
  - `"tap": ["left", ...]`
  - `"release": ["left", ...]`
  - `"drag": {"x1": 0, "y1": 0, "x2": 0, "y2": 0}`

  A button must not appear in more than one of `hold`, `tap` and `release`. `drag` must not be combined with `"left"` in `hold` or `tap`, nor with a `move` channel.

- `move` must use exactly one of the following forms:

  - `{"mode": "absolute", "x": 0, "y": 0}`
  - `{"mode": "relative", "dx": 0, "dy": 0}`

- `scroll` must use the following form:

  - `{"dx": 0, "dy": 0}`

- `complete` indicates whether the task has finished:

  - `"complete": true`
  - `"complete": false`

- `reason` briefly explains why the task ended. Include it only when `complete` is `true`. When `complete` is `true`, `reason` is required and must contain exactly one concise sentence.

- `goal` states the short, high-level objective served by the current action.

- `thought` gives a concise rationale for choosing the current action based on the visible game state, the reported input state and recent action history. Keep it brief and do not provide lengthy step-by-step reasoning.
````

```
- `expected_outcome` briefly describes the immediate result expected from
    the action.

#### Completion Rules

If the task is still in progress:

- Omit `reason`.
- Either omit `complete` or set it to `false`.

If the task is finished:

- Set `complete` to `true`.
- Include a one-sentence `reason`.
- Omit all device channels because they are ignored after completion.
    Every held input is released automatically when the episode ends.

Preferred completion format:

`{"complete": true, "reason": "The objective has been successfully
    completed."}`
```

```
### Task Objective

You must accomplish the following task objective:

{task_prompt}

#### How this task ends

- The episode ends ONLY when YOU reply with `"complete": true`, or when
    the step budget is exhausted.
- Reply with `"complete": true` as soon as the objective above is
    satisfied. Do NOT declare completion before it is met.
```

The following annotation instructions are appended to the game prompt. They change the role of the model from choosing an action to explaining an action that the player has already executed. They also require the model to write from the perspective of the player and to phrase the expected outcome as a prediction, without revealing that the subsequent screenshot has been observed.

```
### Annotation Mode

In this session you are NOT choosing the action. A human expert already
    played this step, and your job is to reconstruct the first-person
    rationale an agent would have written for the action the human took.
    Besides the usual context (action history, current input state,
    current game screen) you receive two extra sections: `Executed Action
    `, the exact device-channel JSON the player produced on the current
    screen, and `Game Screen After The Action`, what the screen looked
    like right after it, so that your `expected_outcome` is accurate.

Rules:

- Write `goal`, `thought` and `expected_outcome` in the first person and
    present tense, as the player deciding this action from the current
    screen. `goal` is a short high-level objective; `thought` is a
    concise rationale grounded in what is visible on the current screen,
    the reported input state and the recent history (1-2 sentences); `
    expected_outcome` is the immediate result the player expects (1
    sentence).
- Never mention that a human played, that you are annotating, or that you
     have seen a later frame. Do not write "the next frame shows"; phrase
     `expected_outcome` as an expectation.
```

```
- An empty executed action `{}` means the player deliberately waited and
    let the game advance one step; explain why waiting was the right call
     on this screen.
- Do not repeat, add or alter device channels. Reply with exactly one
    JSON object with the keys `goal`, `thought` and `expected_outcome` (
    plus a one-sentence `reason` when the executed action is `{"complete
    ": true}`), and nothing else.
```

C.2 ANNOTATION REQUEST

The user message first presents the context of step t in the same layout as a decision request under the game prompt. It contains up to four preceding steps in chronological order, each with its screenshot and recorded action, followed by the input device state and the screenshot of step t. The action history is omitted at the first step of a trajectory.

```
## Action History

Here are your last {n} steps, oldest first. For each step you can see the
     game screen you were looking at and the action you took for it.

### Step {j}

Game screen:
[screenshot of step j]
Action: {recorded action of step j}

...(one entry per preceding step)

## Current Input State

This is what is physically held right now, as confirmed by the
    environment. Base your `hold` / `release` decisions on it: held
    inputs stay down until you release them.
{input_state}

## Current Game Screen

This is the game screen right now. Decide your next action based on it,
    the current input state and your action history.
[screenshot of step t]
```

The executed action is then supplied using the following template, where {`action`} is the recorded action of step t in JSON form.

```
## Executed Action

On the current game screen above, the player performed exactly this
    action:
{action}
```

For a step that does not declare completion, the screenshot captured immediately after the action is appended with the following heading, so that the expected outcome can reflect the actual effect of the action.

```
## Game Screen After The Action

This is the game screen right after that action was executed:
[screenshot of step t+1]
```

A completion step, whose recorded action is {`"complete": true`}, has no subsequent screenshot. Instead, the following note is appended to the executed action to request an additional justification.

```
The player declared the task complete here. Also include a one-sentence `
    reason` explaining why the objective is satisfied on the current
    screen.
```

The request ends with the following output instruction. For a completion step, the JSON object in this instruction additionally contains the key `"reason"`.

```
## Your Reply

Reply with exactly one JSON object: {"goal": "...", "thought": "...", "
    expected_outcome": "..."}. No other text.
```

We parse the reply as a JSON object and accept it when it contains nonempty `goal` and `thought` fields, together with a nonempty `reason` field for a completion step. Otherwise, the request is repeated, up to five attempts in total. The accepted `thought`, `goal`, and `expected_outcome` fields form the `<think>`, `<goal>`, and `<eo>` segments of the training target, respectively.