TOWARDS GENERAL COMPUTER CONTROL: A MULTIMODAL AGENT FOR RED DEAD REDEMPTION II AS A CASE STUDY

Weihao Tan[2*], **Ziluo Ding**[1], **Wentao Zhang**[2], **Boyu Li**[1], **Bohan Zhou**[3*], **Junpeng Yue**[3*], **Haochong Xia**[2], **Jiechuan Jiang**[3], **Longtao Zheng**[2], **Xinrun Xu**[1], **Yifei Bi**[1], **Pengjie Gu**[2], **Xinrun Wang**[2], **Börje F. Karlsson**[1], **Bo An**[2†], **Zongqing Lu**[3, 1†]
[1] Beijing Academy of Artificial Intelligence (BAAI), China
[2] Nanyang Technological University, Singapore
[3] School of Computer Science, Peking University, China
weihao001@ntu.edu.sg boan@ntu.edu.sg zongqing.lu@pku.edu.cn
Project website: https://baai-agents.github.io/Cradle/

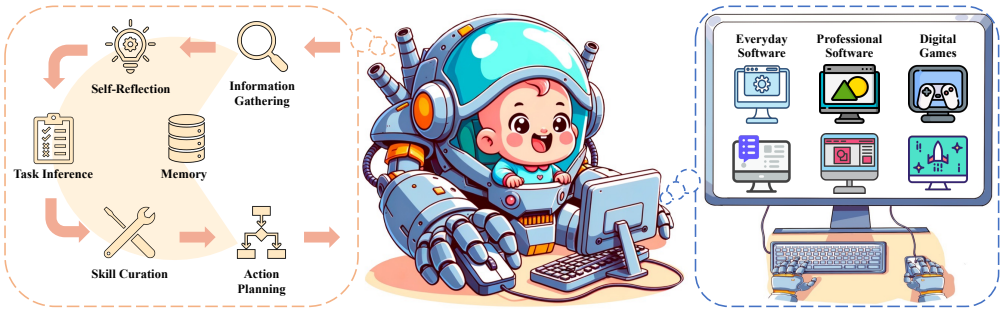


Figure 1: The **CRADLE** framework empowers nascent foundation models to perform complex computer tasks via the same general interface humans use: *e.g.,* screen as input and keyboard & mouse operations as output.

ABSTRACT

Despite the success in specific tasks and scenarios, existing foundation agents, empowered by large models (LMs) and advanced tools, still cannot generalize to different scenarios, mainly due to dramatic differences in the observations and actions across scenarios. In this work, we propose the General Computer Control (GCC) setting: building foundation agents that can master any computer task by taking only screen images (and possibly audio) of the computer as input, and producing keyboard and mouse operations as output, similar to human-computer interaction. The main challenges of achieving GCC are: 1) the multimodal observations for decision-making, 2) the requirements of accurate control of keyboard and mouse, 3) the need for long-term memory and reasoning, and 4) the abilities of efficient exploration and self-improvement. To target GCC, we introduce **CRADLE**, an agent framework with six main modules, including: 1) information gathering to extract multi-modality information, 2) self-reflection to rethink past experiences, 3) task inference to choose the best next task, 4) skill curation for generating and updating relevant skills for given tasks, 5) action planning to generate specific operations for keyboard and mouse control, and 6) memory for storage and retrieval of past experiences and known skills. To demonstrate the capabilities of generalization and self-improvement of **CRADLE**, we deploy it in the complex AAA game Red Dead Redemption II, serving as a preliminary attempt towards GCC with a challenging target. To our best knowledge, our work is the first to enable LMM-based agents to follow the main storyline and finish real missions in AAA games, with minimal reliance on prior knowledge or resources.

[*] Work performed while an intern at BAAI.
[†] Corresponding authors.

1 INTRODUCTION

Achieving Artificial General Intelligence (AGI) has long been a north-star goal for the AI community (Morris et al., 2023). Recent foundation agents, *i.e.,* agents empowered by large multimodal models (LMMs) and advanced tools, have been touted as a promising approach in pursuing AGI. While recent research has demonstrated the success of such agents in specific scenarios or tasks, including web browsing (Zhou et al., 2023; Deng et al., 2023; Gur et al., 2023; Zheng et al., 2024b; He et al., 2024), operating mobile applications (Yang et al., 2023), crafting and exploration in Minecraft (Wang et al., 2023c;a), and some robotics scenarios (Huang et al., 2022; Brohan et al., 2023; Driess et al., 2023); current foundation agents still cannot generalize across different targets, mainly due to different observation and action spaces, environment dynamics (*e.g.,* Minecraft and robotics), dependence on target-specific resources, or other semantic gaps (*e.g.,* lack of environment feedback for actions) (Xu et al., 2024).

Computers[1] are the most important and universal interface in the increasingly digital world. Computer tasks cover a wide variety of scenarios, including complex software to create digital artifacts, *e.g.,* Photoshop, everyday productivity software, *e.g.,* spreadsheets and word processors, apps and websites with effects in the real world, *e.g.,* banking or travel booking apps, video games, *e.g.,* Red Dead Redemption II (RDR2), and control of external devices, *e.g.,* 3D printing devices and network routers. By providing standardized universal observation (*i.e.,* screen and audio) and abstract actions (*i.e.,* keyboard and mouse operations), computer control is an ideal testbed to develop foundation agents for varied complex tasks in dynamic scenarios and thus have the potential to unlock insights on the path to AGI. Therefore, we propose the *General Computer Control* (GCC) setting:

> *Building **foundation agents** that can master **ANY** computer task,* e.g., *software, games, etc., through only standard observation (*i.e., *screen and audio) and input device operations (*i.e., *keyboard and mouse).*

There are many challenges to achieve GCC: i) Observations in GCC are multimodal, which requires the alignment of the data in different modalities for better understanding and decision-making; ii) GCC requires accurate control of device operations (keyboard and mouse) to interact with the computer; iii) It requires long-term memory to store past experiences, due to the partial observability of GCC tasks, and reasoning ability over experiences to reuse the knowledge to solve novel tasks; and iv) GCC requires efficient exploration of the environments in a structured manner to discover better strategies and solutions autonomously, *i.e.,* self-improving, which allows agents to generalize across the myriad tasks in the digital world. We discuss GCC in more detail in Section 2.

In this work, we introduce **CRADLE**, a novel framework targeting GCC, and make a preliminary attempt towards it with a challenging target. As shown in Figure 1, distinctly from previous methods focused on controlling a web browser or software with easy access to their internal APIs and states (Furuta et al., 2023; Yang et al., 2023), **CRADLE** only takes screen images as input and outputs keyboard and mouse operations through a comprehensive reasoning process, enabling it to effectively understand the current situation and make reasonable decisions. Furthermore, we deploy **CRADLE** in the highly acclaimed AAA game[2], Red Dead Redemption II (RDR2). We select RDR2 for our case study due to its complex blackbox control system, which epitomizes the most demanding computer tasks and enables us to evaluate the performance boundaries of our framework in such virtual environments. RDR2 is characterized by its rich and diverse information, encompassing elements like dialogues, unique icons, and in-game prompts and instructions, thus requiring the capture and interpretation of various information forms. Additionally, the game requires a broader range of keyboard and mouse interactions than typical software, such as using the mouse for navigation and combining mouse buttons and keyboard keys interactively to realize actions in the game world (which necessitate not only precise key selection, but also determination of timing, and multi-iteration combinations). We argue that by demonstrating the feasibility of playing this complex game, **CRADLE** sheds light on its potential for GCC.

[1] Throughout the paper we use the term *computer* as a synonym for any user-focused computational device, *e.g.,* PC, smartphone, and tablet. While our description focuses on keyboard and mouse operations, it can be easily generalized to control handles and touchscreens.

[2] In the video game industry, AAA (Triple-A) is a term used to classify games with typically high development budgets and production values for the technology available in their time.

Our major contributions are summarized as follows:

- We propose the novel setting of General Computer Control (GCC), serving as a milestone towards AGI in the digital world, where agents take multimodal observations as inputs and output keyboard and mouse operations, similar to human-computer interactions.
- We propose a novel foundation agent framework (**CRADLE**) for the GCC setting[3], which has strong reasoning abilities, including self-reflection, task inference, and skill curation, to ensure its generalizability and self-improvement across various computer tasks.
- To demonstrate the capabilities of **CRADLE**, we incorporate the powerful Large Multimodal Model (LMM) GPT-4V into our framework and deploy it in the famous AAA game RDR2, serving as a preliminary attempt towards GCC. To our best knowledge, our work is the first to enable LMM-based agents to follow the main storyline and finish real missions in complex AAA games, with minimal reliance on prior knowledge.

2 GENERAL COMPUTER CONTROL

Definition. General Computer Control (GCC) is a setting where *an agent controls a computer through only standardized human-like interactions, i.e., using the screen and (optionally) audio as input and keyboard and mouse operations as output*. GCC provides a universal interface to any computer-based task without access to the application source code or APIs. Compared with robotics (Ma et al., 2023b), another promising direction toward AGI, GCC provides a more cost-effective and controllable experimental framework, also with significant impact in real scenarios. Due to its universality, achieving GCC can be seen as a significant milestone towards AGI, starting from the digital world, and can be further combined with robotics to address physical environments.

Figure 2: Taxonomy of GCC.

Taxonomy. As illustrated in Figure 2, there are three main categories of GCC targets: i) manipulating web software, *e.g.,* websites and cloud-based services (Deng et al., 2023), ii) manipulating traditional desktop software, *e.g.,* office suites (Wu et al., 2024), and iii) playing video games, *e.g.,* Minecraft (Wang et al., 2023a). Among them, digital video games are believed to be the most difficult tasks mainly due to i) environment complexity, *e.g.,* the open-ended world in RDR2, ii) the long-horizon decision making (not always linear), *e.g.,* 1K+ semantic steps in RDR2 tasks, and iii) the partial observability of tasks, *i.e.,* agents need to store their past experiences and reason upon them for better decision making. Digital games have been a testbed for research on decision making, *e.g.,* Atari games in reinforcement learning research (Bellemare et al., 2013). But complex more modern games present a much more challenging target for the development of foundation agents, due to their similarities to virtual reality applications and the real world.

Challenges. There are several key challenges in targeting GCC. First, observations in GCC are multimodal, including images (video frames or screenshots), text (from the command line or included in images), and audio (*e.g.,* instructions, effects, and music). For example, when playing digital games, the agent commonly needs to follow instructions or dialogue (text/audio) and interpret the world (screen images) to complete tasks. Second, GCC requires the accurate control of keyboard and mouse to interact with the computer, where the action space of the mouse is continuous, *i.e.,* locations on the screen to move and varying speed of movement, and the action space of the keyboard control is discrete, but very large, considering combinations of keys, timing, and other attributes. Furthermore, the syntax contracts for the code generated by the agent to execute such IO operations introduce additional difficulties. Third, GCC tasks are usually partially observable. For example, the player in RDR2 cannot know the information of towns until he reaches them. Therefore, GCC requires the agent to have long-term memory to store past experiences, such as past actions, outcomes, and the knowledge learned from the past experiences, for fast and better decision making. Fourth, GCC requires efficient exploration of the environments in a structured manner and

[3]**CRADLE**'s codebase is open-sourced at: https://github.com/BAAI-Agents/Cradle

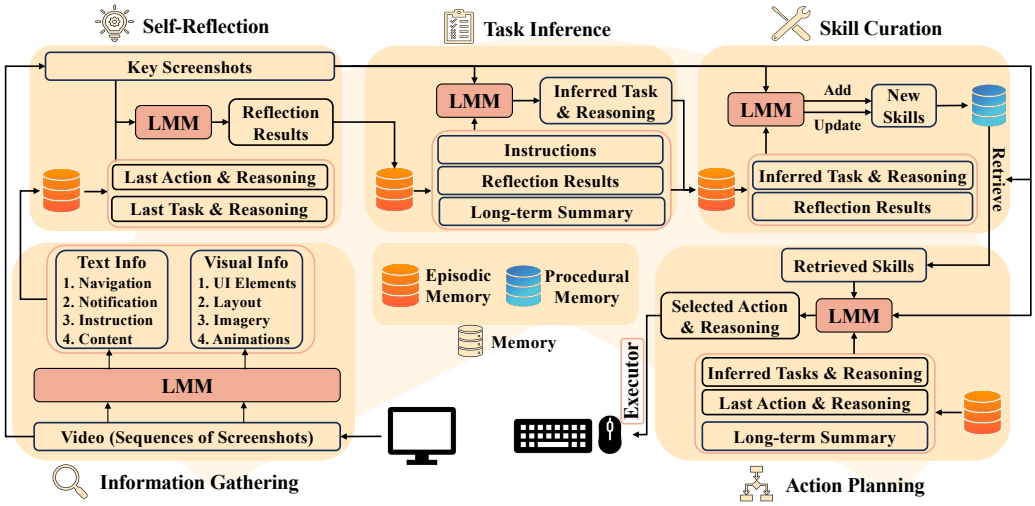


Figure 3: An overview of the **CRADLE** framework. **CRADLE** takes video from the computer screen as input and outputs computer keyboard and mouse control determined through inner reasoning.

discovering better strategies and solutions autonomously, *i.e.,* self-improving. The ability to explore and self-improve enables agents to generalize easily across various tasks in GCC. More details on such challenges are discussed in Appendices C.1 and C.2.

3 Related Work

The prosperity of LLMs has broadened the potential of deploying powerful foundation models as autonomous agents for completing complex tasks in various computer applications, such as web navigation (Zhou et al., 2023; Deng et al., 2023; Mialon et al., 2023), software manipulation (Rawles et al., 2023; Yang et al., 2023; Kapoor et al., 2024) and game playing (Wang et al., 2023c;a; Ma et al., 2023a; Xu et al., 2024). However, these web agents usually use raw HTML code and DOM tree as input and interact with the available element IDs, missing the rich visual patterns with key information, like icons, images, and spatial relations. The game agents usually adopt textual observations obtained from internal APIs and pre-defined semantic actions. These domain-specific settings make them struggle with generalizing to other games, let alone other types of applications. These methods fail to generalize across various tasks, due to the inconsistent observation and action spaces. This indicates the necessity of the GCC setting, which provides a unified representation of the observation and action spaces for enormous challenging computer tasks. We provide more discussions in the extended related work in Appendix A.

4 The **CRADLE** Framework

To pursue GCC, we propose **CRADLE**, illustrated in Figure 3, an agent framework that can properly handle the challenges GCC presents, *i.e.,* observing and interacting with any environment and dealing with corresponding information and semantic gaps, without relying on any special API that is not available to a typical computer user. A framework for GCC should have the ability to understand and interpret computer screens and dynamic changes between consecutive frames (and possibly audio) from arbitrary software and generate reasonable computer control actions to be executed reliably. This suggests a multimodal model with powerful vision and reasoning capabilities, in addition to rich knowledge of computer UI and control, is a key requirement. In this work, we leverage GPT-4V, one of the most capable current LMM models, as the framework's backbone model.

4.1 Environment IO

Although the input from the environment may cross multiple modalities (*e.g.,* visual, audio), in this work, we cast **CRADLE** to consume videos (sequences of screenshots) of the screen as input

and produce keyboard and mouse control commands as output. As this matches the most common setting faced by users, it is a pragmatic scenario in a preliminary attempt towards GCC.

Information Gathering. As shown in Figure 3, to capture all information relevant to understanding the recent situation and perform further reasoning, **CRADLE** takes as input a video recording (since the last executed action) and needs to make sure information is properly extracted from it, which includes both textual and visual information. Textual information usually includes content (headings and paragraphs), navigation labels (menus and links), notifications, and instructions to convey messages and guide users, which usually depend on the OCR ability of LMM models. On the other hand, visual information includes layout, imagery (visual contents of the screenshot itself and icons), animations, and UI elements to enhance user experience and interface design, which poses high requirements for the spatial perception and visual understanding of LMM models.

Skill and Action Generation Under the GCC setting, the only way to interact with the environment is through keyboard and mouse operations. To bridge the gap between actions outputted by the framework and the OS-level executable actions, **CRADLE** uses the LMM to generate code function as semantic-level skills which encapsulate lower-level keyboard and mouse control, *e.g.,* `def move_forward(duration): key_hold('W', duration)`. We then let the LMM instantiate these skill functions into executable actions by specifying any necessary parametric aspects, such as duration, position, and speed, *e.g.,* `move_forward(duration = 2)`.

Action Execution. After the LMM generates actions and decides to execute them in the environment, an ***Executor*** is then triggered to map these semantic actions to the final OS-level keyboard and mouse commands to interact with the target environment.

4.2 REASONING

Based on the extracted information from the video observations and relevant information from its memory, **CRADLE** needs to reason, taking into account incomplete information and semantic gaps, and then make the next decision. This process is analogous to "**reflect on the past, summarize the present, and plan for the future**", which is broken down into the following modules.

Self-Reflection. The reflection module initially evaluates whether the last executed action was successfully carried out and whether the task was completed. Sequential key screenshots from the last video observation, along with the previous context for action planning and task inference are fed to the LMM for reasoning. Additionally, we also request the LMM to provide an analysis of any failure. This valuable information enables **CRADLE** to try and remedy inappropriate decisions or less-than-ideal actions. Furthermore, reflection can also be leveraged to inform re-planning of the task and bring the agent closer to target task completion, better understand the factors that led to previous successes, or suggest how to update or improve specific skills.

Task Inference. After reflecting on the outcome of the last step, **CRADLE** needs to analyze the current situation to infer the most suitable task for the current moment. We let the LMM estimate the highest priority task to perform and also determine whether it is time to stop an ongoing task and start a new one, prompting it with the key screenshots, the long-term summary of past experiences, and the latest reflection results.

Skill Curation. As a new task is selected, **CRADLE** needs to prepare the tactics to accomplish it, by retrieving useful skills from the procedural memory, updating skills, or generating new ones. Similarly to Liang et al. (2023); Wang et al. (2023a), skills in **CRADLE** are represented as code functions; a form both flexible and interpretable, and easy for LMMs to understand and maintain. Atomic skills are usually made up of simple calls to keyboard and mouse control, *e.g.,* `key_press()` to press a given key, which can then be extended and rewritten into more complex composite skills.

Action Planning. To complete a given task, the LMM needs to select the appropriate skills from the curated skill set and instantiate these skills into a sequence of executable actions by specifying any necessary parametric aspects (*e.g.,* duration, position, and target) according to the inferred task, last action, and long-term summary. The action sequence is then fed to the ***Executor*** for interaction with the environment. It is important to note that in complex digital games, the effective action space is composed not only of keyboard and mouse function calls per se, but also involves timing and cross-action interaction, semantic mapping of term usage on screen to action-specific details, among other factors. Moreover, performing actions in the environment is non-trivial also as mapping code

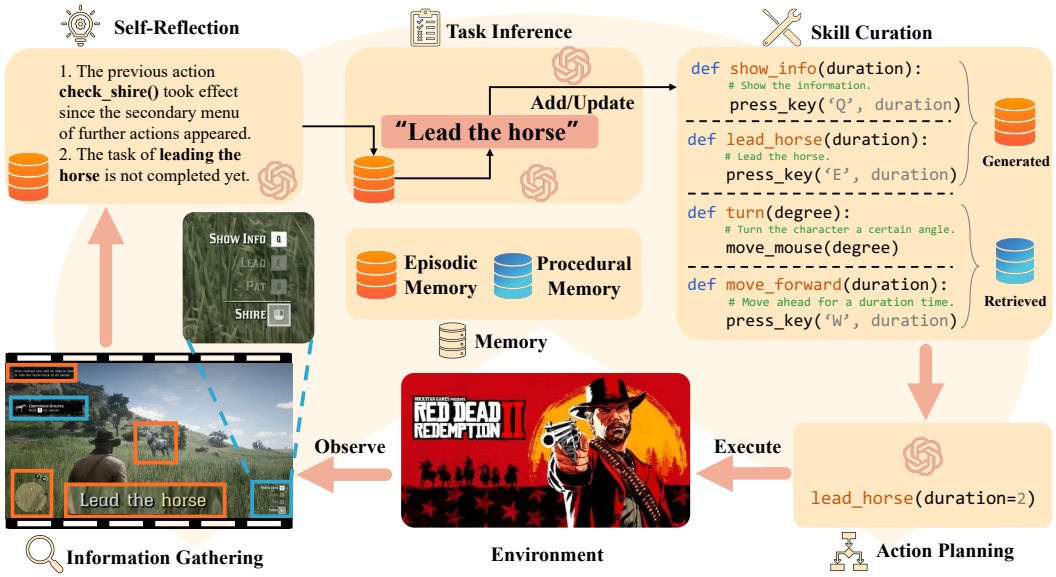


Figure 4: The detailed illustration of how **CRADLE** is instantiated as a game agent to play RDR2.

execution to its effects is not always explicit. That is, action execution can result in no error from the code or game, but still be incorrect or ineffective.

4.3 MEMORY

CRADLE stores and maintains all the useful information provided by the environment and the LMM's output through a memory module, consisting of episodic memory and procedural memory.

Episodic Memory. Episodic memory is used to maintain current and past experiences, including key screenshots from each video observation, and all LMM output, *e.g.,* textual and visual information, actions, tasks, and reasoning from each module. To facilitate retrieval and storage, periodical summarization is conducted to abstract recently added multimodal information into long-term summaries. The incorporation of episodic memory enables **CRADLE** to effectively retain crucial information over extended periods, thereby enhancing its decision-making capabilities.

Procedural Memory. This memory is specific to storing and retrieving skills in code form, which can be learned from scratch or pre-defined in the procedural memory. Upon skill curation, skills can be added, updated, or composed in the memory. The most relevant skills for a given task and situation will be retrieved to support action planning, so, as **CRADLE** continuously acquires new skills during interaction, it is critical that this memory can effectively calculate skill relevance.

5 EMPIRICAL STUDIES

As shown in Figure 4, we deploy **CRADLE** as a game-playing agent in the renowned AAA game RDR2, showcasing our framework for GCC with a complex target. To the best of our knowledge, this is the first work to explore such a challenging game under the GCC setting, without access to any internal game state or API (*i.e.,* the agent has to interact with the game in a human-like manner). RDR2 is a typical 3D RPG-like game with classical keyboard and mouse controls and movie-like realistic graphics. The game's progression unveils a mix of story content, dialogues and instructions, and helpful tips for interactive game mechanisms, which are leveraged by our agent to independently expand the skills in the procedural memory from scratch. Demonstrating our agent's capability to navigate the world and complete tasks following the main storyline in RDR2 underscores the significant potential of our framework in advancing towards GCC. We provide a more detailed introduction of the game in Appendix B.1. The implementation details of the framework and all LMM prompts can be found in Appendices C.1 and C.3.

Objective. To emulate a new human player learning to play the game from scratch, including in-game beginner tutorials and hints, we mainly focus on the first two missions of the main storyline

in Chapter I, *Explore Shelter* and *Rescue John*, including horse riding, NPC following, house exploration, weapons selection, combat with enemies and wolves, etc. Typically, it takes a novice human player around 40 minutes of the gameplay duration to finish these missions. Seldom previous work attempted this kind of challenge with super long-horizon tasks and rich semantic environments. Besides the missions in the main storyline, we also evaluate **CRADLE** in the open-ended world in Chapter II with a mission, *Buy supply*, where the agent is instructed to go from the camp to the General Store in the town of Valentine for supplies. In such open-ended tasks, in-game guidance will seldom appear and the agent needs to analyze its situation and propose feasible solutions to complete the mission. We provide a brief introduction to these tasks in Appendix B.2.

Observations and Action Space. Strictly following the GCC setting, our agent takes the video of the screen as input and outputs keyboard and mouse operations to play the game. To lower the frequency of interaction with the backbone model, the video recorder takes a game screenshot every 0.5 seconds, which proves to be sufficient for information gathering without missing any important information. For the action space, we categorize the keyboard and mouse actions into 4 key categories: `press(key, duration)`, `hold(key, duration)`, `release(key)`, and `pointer_move(x, y)`, which can be combined in different ways to form combos, use keys in fast sequence, or coordinate timings. Skill code needs to be generated by the agent in order to utilize such functions and affordances so executed actions take effect. More details on the action space are provided in Appendix C.1.

5.1 Case Studies

Here we present a few case studies for a more in-depth discussion of the framework capabilities. More challenges of the GCC setting are discussed in Appendix C.2. We also provide an analysis of the limitations of GPT-4V in Appendix D.

5.1.1 Self-Reflection

Self-reflection is an essential component in **CRADLE** as it allows our framework reasoning to correct previous mistakes or address ineffective actions taken in-game. Figure 5 provides an example of the self-reflection module. The task requires the agent to select a weapon to equip, in the context of the "Protect Dutch" task. Initially, the agent selects a knife as its weapon by chance, but since the game requires a gun to be chosen, this is incorrect and the game still prompts the player to re-open the weapon wheel. The self-reflection module is able to determine that the previous action was incorrect and on a subsequent iteration the agent successfully opts for the gun, correctly fulfilling the task requirement and advancing to the next stage in the story.

5.1.2 Skill Curation

For skill curation, we first provide GPT-4V with examples of general mouse and keyboard control APIs, *e.g.*, io_env.key_press and io_env.mouse_click. Figure 6 shows that GPT-4V can capture and understand the prompts appearing on screenshots, *i.e.,* icons and text, and strictly follow the provided skill examples using our IO interface to generate correct skill code. Moreover, GPT-4V also generates comments in the code to demonstrate the functionality of this skill, which are essential for computing similarity and relevance with a given task during skill retrieval. The quality of the generated comment directly determines the results of skill retrieval, and further impacts reasoning to action planning. Curation can also re-generate code for a given skill, which is useful if GPT-4V wrongly recognized a key or mouse button in a previous iteration.

5.2 Quantitative Evaluation

To illustrate the effectiveness and importance of different modules in **CRADLE** to its overall performance, we evaluate the framework on seven representative tasks from the main storyline and open-ended missions, compared with two ablation-like baselines: **CRADLE** without Self-Reflection and **CRADLE** without Task Inference. Except for the task of *Protect Dutch* which involves a fast-paced gun battle, *Search house*, which requires the agent to explore a complex indoor environment, and the open-ended task with long-horizon *Buy supply*, **CRADLE** can complete all tasks in the main storyline consistently. Moreover, even for these complex tasks, **CRADLE** also achieves a significantly

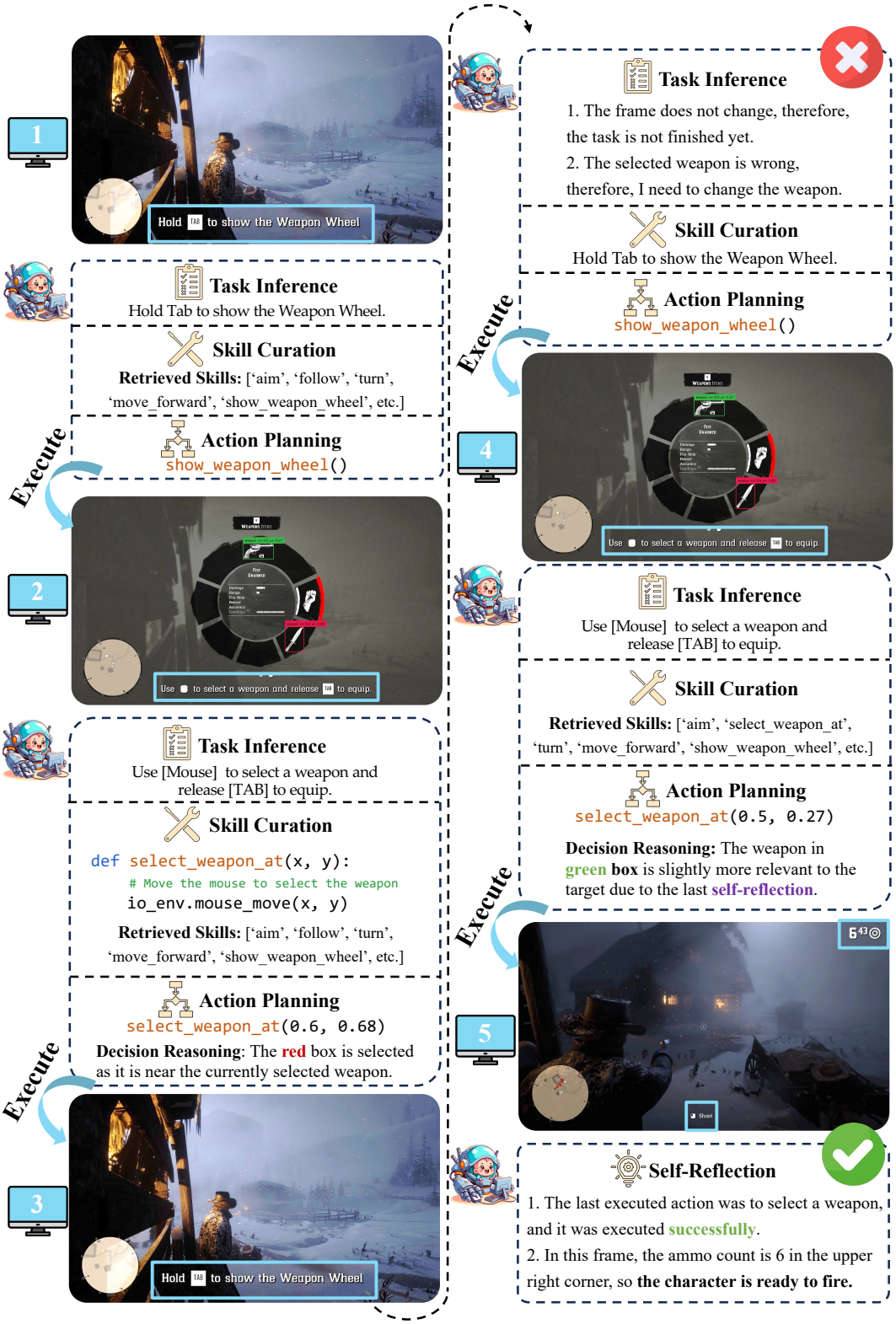


Figure 5: Case study of self-reflection on re-trying a failed task. Task instruction and context require the agent to equip the gun. A wrong weapon (knife) is first selected, but the agent equips the gun after self-reflection. Only relevant modules are shown for better readability, though all modules (Figure 3) are executed per iteration.


```
def take_cover(duration = 2):
    """
    press "Q" to take cover
    """
    io_env.key_press('q', duration)
```

```
def focus_on_horse():
    """
    hold "right mouse button" to focus on horse
    """
    io_env.mouse_hold('right button')
```

```
def view_stored_weapons():
    """
    press "tab" to view your stored weapons
    """
    io_env.key_hold('tab')
```

Figure 6: Skill code generation based on in-game instructions. As the storyline progresses, the game will continually provide prompts on how to use a new skill via keystrokes or utilizing the mouse.

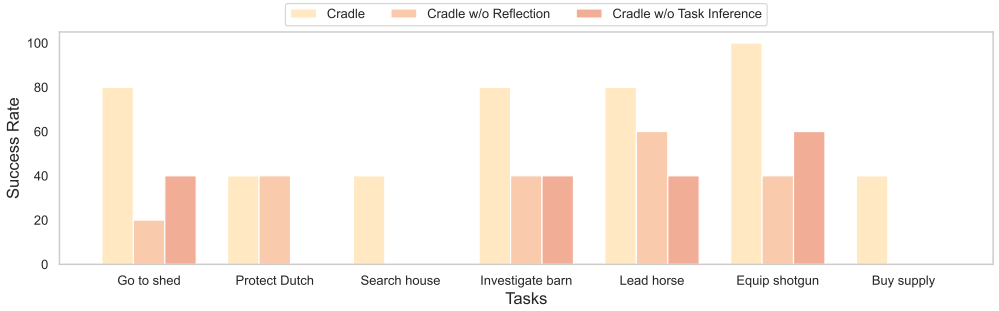


Figure 7: **CRADLE** performance on seven representative tasks in RDR2. The first six tasks are from the two missions in the main storyline and the last one is one of the open-ended missions. Every task is tested five times with a maximum of ten minutes in-game time. The test will also terminate if the mission fails.

higher success rate than disabling either Self-Reflection or Task Inference. The most common failure for the *Buy supply* open-world complex task happens when the agent riding a horse in the lively Valentine town crashes into a passerby or a carriage.

Without task inference, the agent can still leverage the extracted instructional text information and long-term summary to infer both task and goal in the action planning process, which can lead to some successful task completions. However, over time the instructional text disappears from the screen and the long-term summary will be diluted as relevant information will be forgotten due to the recently added entries, which explains why it fails across all tasks, but still has a reasonable success rate in the *Equip shotgun* task, where the task has a shorter horizon and its instructional text appears at the bottom of the screen. For the open-ended mission, *Buy supply*, without self-reflection to propose short-horizon goals, the agent even struggles with leaving the camp.

On the other hand, without self-reflection, **CRADLE** struggles with movement, especially when the agent is blocked by obstacles, which is difficult for action planning to notice and address. Without an independent critic to reflect on the current situation, **CRADLE** tends to still trust its previous reasoning and believe that the actions were carried out successfully. As *Search house* is a complex indoor task with various pieces of furniture as obstacles, we can see **CRADLE** without Self-Reflection exhibits extremely poor performance, as expected. Without self-reflection, the agent also struggles with entering the store and is always confused about whether the task is finished.

6 CONCLUSIONS

In this work, we introduce GCC, a general, challenging setting aimed to pave the way towards more general foundation agents across computer tasks. Moreover, we propose **CRADLE**, a novel framework that enables LMM-based agents to work in such an impactful setting, and we further showcase its effectiveness in the famous AAA game, RDR2. **CRADLE** exhibits strong performance in learning skills, following the storyline, and finishing the real missions in the game. **CRADLE** serves as a pioneering work to develop more powerful LMM-based general agents across computer control tasks, combining both further framework enhancements and new advances in LMMs.

REFERENCES

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. GPT-4 Technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Bowen Baker, Ilge Akkaya, Peter Zhokov, Joost Huizinga, Jie Tang, Adrien Ecoffet, Brandon Houghton, Raul Sampedro, and Jeff Clune. Video pretraining (VPT): Learning to act by watching unlabeled online videos. *Advances in Neural Information Processing Systems*, 35:24639–24654, 2022.

Anton Bakhtin, Noam Brown, Emily Dinan, Gabriele Farina, Colin Flaherty, Daniel Fried, Andrew Goff, Jonathan Gray, Hengyuan Hu, et al. Human-level play in the game of diplomacy by combining language models with strategic reasoning. *Science*, 378(6624):1067–1074, 2022.

Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The Arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

Anthony Brohan, Yevgen Chebotar, Chelsea Finn, Karol Hausman, Alexander Herzog, Daniel Ho, Julian Ibarz, Alex Irpan, Eric Jang, Ryan Julian, et al. Do as I can, not as I say: Grounding language in robotic affordances. In *Conference on Robot Learning*, pp. 287–318. PMLR, 2023.

Roberto Brunelli. *Template matching techniques in computer vision: theory and practice*. John Wiley & Sons, 2009.

Kanzhi Cheng, Qiushi Sun, Yougang Chu, Fangzhi Xu, Yantao Li, Jianbing Zhang, and Zhiyong Wu. SeeClick: Harnessing GUI grounding for advanced visual GUI agents. *arXiv preprint arXiv:2401.10935*, 2024.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2Web: Towards a generalist agent for the web. *arXiv preprint arXiv:2306.06070*, 2023.

Danny Driess, Fei Xia, Mehdi SM Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, et al. Palm-e: An embodied multimodal language model. *arXiv preprint arXiv:2303.03378*, 2023.

Linxi Fan, Guanzhi Wang, Yunfan Jiang, Ajay Mandlekar, Yuncong Yang, Haoyi Zhu, Andrew Tang, De-An Huang, Yuke Zhu, and Anima Anandkumar. Minedojo: Building open-ended embodied agents with internet-scale knowledge. *Advances in Neural Information Processing Systems*, 35: 18343–18362, 2022.

Hiroki Furuta, Ofir Nachum, Kuang-Huei Lee, Yutaka Matsuo, Shixiang Shane Gu, and Izzeddin Gur. Multimodal web navigation with instruction-finetuned foundation models. *arXiv preprint arXiv:2305.11854*, 2023.

Difei Gao, Lei Ji, Zechen Bai, Mingyu Ouyang, Peiran Li, Dongxing Mao, Qinchen Wu, Weichen Zhang, Peiyi Wang, Xiangwu Guo, et al. ASSISTGUI: Task-oriented desktop graphical user interface automation. *arXiv preprint arXiv:2312.13108*, 2023.

Izzeddin Gur, Hiroki Furuta, Austin Huang, Mustafa Safdari, Yutaka Matsuo, Douglas Eck, and Aleksandra Faust. A real-world webagent with planning, long context understanding, and program synthesis. *arXiv preprint arXiv:2307.12856*, 2023.

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.

Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. WebVoyager: Building an end-to-end web agent with large multimodal models. *arXiv preprint arXiv:2401.13919*, 2024.

Wenyi Hong, Weihan Wang, Qingsong Lv, Jiazheng Xu, Wenmeng Yu, Junhui Ji, Yan Wang, Zihan Wang, Yuxiao Dong, Ming Ding, et al. CogAgent: A visual language model for GUI agents. *arXiv preprint arXiv:2312.08914*, 2023.

Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, et al. Inner monologue: Embodied reasoning through planning with language models. *arXiv preprint arXiv:2207.05608*, 2022.

Max Jaderberg, Wojciech M Czarnecki, Iain Dunning, Luke Marris, Guy Lever, Antonio Garcia Castaneda, Charles Beattie, Neil C Rabinowitz, Ari S Morcos, Avraham Ruderman, et al. Human-level performance in 3D multiplayer games with population-based reinforcement learning. *Science*, 364(6443):859–865, 2019.

Matthew Johnson, Katja Hofmann, Tim Hutton, and David Bignell. The Malmo platform for artificial intelligence experimentation. In *Ijcai*, pp. 4246–4247, 2016.

Raghav Kapoor, Yash Parag Butala, Melisa Russak, Jing Yu Koh, Kiran Kamble, Waseem Alshikh, and Ruslan Salakhutdinov. OmniACT: A dataset and benchmark for enabling multimodal generalist autonomous agents for desktop and web, 2024.

Christian Kauten. Super Mario Bros for OpenAI Gym. GitHub, 2018. URL https://github.com/Kautenja/gym-super-mario-bros.

Karol Kurach, Anton Raichuk, Piotr Stańczyk, Michał Zając, Olivier Bachem, Lasse Espeholt, Carlos Riquelme, Damien Vincent, Marcin Michalski, Olivier Bousquet, et al. Google research football: A novel reinforcement learning environment. In *Proceedings of the AAAI conference on artificial intelligence*, pp. 4501–4510, 2020.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 9493–9500. IEEE, 2023.

Shilong Liu, Zhaoyang Zeng, Tianhe Ren, Feng Li, Hao Zhang, Jie Yang, Chunyuan Li, Jianwei Yang, Hang Su, Jun Zhu, et al. Grounding Dino: Marrying dino with grounded pre-training for open-set object detection. *arXiv preprint arXiv:2303.05499*, 2023.

Weiyu Ma, Qirui Mi, Xue Yan, Yuqiao Wu, Runji Lin, Haifeng Zhang, and Jun Wang. Large language models play StarCraft II: Benchmarks and a chain of summarization approach. *arXiv preprint arXiv:2312.11865*, 2023a.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. *arXiv preprint arXiv:2310.12931*, 2023b.

Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. GAIA: a benchmark for general AI assistants. *arXiv preprint arXiv:2311.12983*, 2023.

Meredith Ringel Morris, Jascha Sohl-dickstein, Noah Fiedel, Tris Warkentin, Allan Dafoe, Aleksandra Faust, Clement Farabet, and Shane Legg. Levels of AGI: Operationalizing progress on the path to AGI. *arXiv preprint arXiv:2311.02462*, 2023.

Runliang Niu, Jindong Li, Shiqi Wang, Yali Fu, Xiyu Hu, Xueyuan Leng, He Kong, Yi Chang, and Qi Wang. ScreenAgent: A vision language model-driven computer control agent. *arXiv preprint arXiv:2402.07945*, 2024.

Siyuan Qi, Shuo Chen, Yexin Li, Xiangyu Kong, Junqi Wang, Bangcheng Yang, Pring Wong, Yifan Zhong, Xiaoyuan Zhang, Zhaowei Zhang, et al. CivRealm: A learning and reasoning odyssey in Civilization for decision-making agents. *arXiv preprint arXiv:2401.10568*, 2024.

Christopher Rawles, Alice Li, Daniel Rodriguez, Oriana Riva, and Timothy Lillicrap. Android in the wild: A large-scale dataset for Android device control. *arXiv preprint arXiv:2307.10088*, 2023.

Mikayel Samvelyan, Tabish Rashid, Christian Schroeder De Witt, Gregory Farquhar, Nantas Nardelli, Tim GJ Rudner, Chia-Man Hung, Philip HS Torr, Jakob Foerster, and Shimon Whiteson. The Starcraft multi-agent challenge. *arXiv preprint arXiv:1902.04043*, 2019.

Oriol Vinyals, Igor Babuschkin, Junyoung Chung, Michael Mathieu, Max Jaderberg, Wojciech M Czarnecki, Andrew Dudzik, Aja Huang, Petko Georgiev, Richard Powell, et al. AlphaStar: Mastering the real-time strategy game Starcraft II. *DeepMind blog*, 2:20, 2019.

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*, 2023a.

Junyang Wang, Haiyang Xu, Jiabo Ye, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. Mobile-Agent: Autonomous multi-modal mobile device agent with visual perception. *arXiv preprint arXiv:2401.16158*, 2024.

Zihao Wang, Shaofei Cai, Anji Liu, Yonggang Jin, Jinbing Hou, Bowei Zhang, Haowei Lin, Zhaofeng He, Zilong Zheng, Yaodong Yang, et al. Jarvis-1: Open-world multi-task agents with memory-augmented multimodal language models. *arXiv preprint arXiv:2311.05997*, 2023b.

Zihao Wang, Shaofei Cai, Anji Liu, Xiaojian Ma, and Yitao Liang. Describe, explain, plan and select: Interactive planning with large language models enables open-world multi-task agents. *arXiv preprint arXiv:2302.01560*, 2023c.

Zhiyong Wu, Chengcheng Han, Zichen Ding, Zhenmin Weng, Zhoumianze Liu, Shunyu Yao, Tao Yu, and Lingpeng Kong. OS-copilot: Towards generalist computer agents with self-improvement. *arXiv preprint arXiv:2402.07456*, 2024.

Peter R. Wurman, Samuel Barrett, Kenta Kawamoto, James MacGlashan, Kaushik Subramanian, Thomas J. Walsh, Roberto Capobianco, Alisa Devlic, Franziska Eckert, Florian Fuchs, et al. Outracing champion Gran Turismo drivers with deep reinforcement learning. *Nature*, 602(7896): 223–228, 2022.

Xinrun Xu, Yuxin Wang, Chaoyi Xu, Ziluo Ding, Jiechuan Jiang, Zhiming Ding, and Börje F. Karlsson. A Survey on Game Playing Agents and Large Models: Methods, Applications, and Challenges. *arXiv preprint arXiv:2403.10249*, 2024.

An Yan, Zhengyuan Yang, Wanrong Zhu, Kevin Lin, Linjie Li, Jianfeng Wang, Jianwei Yang, Yiwu Zhong, Julian McAuley, Jianfeng Gao, Zicheng Liu, and Lijuan Wang. GPT-4V in wonderland: Large multimodal models for zero-shot smartphone GUI navigation. *arXiv preprint arXiv:2311.07562*, 2023.

Zhao Yang, Jiaxuan Liu, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. AppAgent: Multimodal agents as smartphone users. *arXiv preprint arXiv:2312.13771*, 2023.

Chaoyun Zhang, Liqun Li, Shilin He, Xu Zhang, Bo Qiao, Si Qin, Minghua Ma, Yu Kang, Qingwei Lin, Saravan Rajmohan, et al. UFO: A UI-focused agent for Windows OS interaction. *arXiv preprint arXiv:2402.07939*, 2024.

Boyuan Zheng, Boyu Gou, Jihyung Kil, Huan Sun, and Yu Su. GPT-4V(ision) is a generalist web agent, if grounded. *arXiv preprint arXiv:2401.01614*, 2024a.

Longtao Zheng, Rundong Wang, Xinrun Wang, and Bo An. Synapse: Trajectory-as-exemplar prompting with memory for computer control. In *ICLR*, 2024b.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Yonatan Bisk, Daniel Fried, Uri Alon, et al. WebArena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023.

A Extended Related Work

A.1 LLM-based Agents for Computer Tasks

The prosperity of LLMs has broadened the potential of deploying powerful foundation models as autonomous agents for completing complex tasks in various computer applications, such as web navigation (Zhou et al., 2023; Deng et al., 2023; Mialon et al., 2023), software manipulation (Rawles et al., 2023; Yang et al., 2023; Kapoor et al., 2024) and game playing (Wang et al., 2023c;a; Ma et al., 2023a; Xu et al., 2024). While previous LLM-based web agents (Deng et al., 2023; Zhou et al., 2023; Gur et al., 2023; Zheng et al., 2024b) show some promising results in effectively navigating, understanding, and interacting with content on webpages, they usually use raw HTML code and DOM tree as input and interact with the available element IDs, missing the rich visual patterns with key information, like icons, images, and spatial relations.

Empowered by the advanced LMMs, multimodal web agents (Hong et al., 2023; Furuta et al., 2023; Yan et al., 2023; He et al., 2024; Zheng et al., 2024a) and mobile app agents (Yang et al., 2023; Wang et al., 2024) have been explored. Instead of HTML source code, they usually take screenshots as input, however, they still need to use the built-in APIs to get the available interactive element IDs to execute corresponding actions.

Similar to web agents, recent works attempt to deploy LLM agents to various complex video games, such as Minecraft (Wang et al., 2023c;a), Starcraft II (Ma et al., 2023a) and Civilization-like game (Qi et al., 2024) with textual observations obtained from internal APIs and pre-defined semantic actions. These domain-specific settings make them struggle with generalizing to other games, let alone websites and other software. Although JARVIS-1(Wang et al., 2023b) claims to interact with the environment in a human-like manner with the screenshots as input and mouse and keyboard for control, its action space is predefined as a hybrid space composed of keyboard, mouse, and API.

Though achieving promising results in specific tasks, these methods fail to generalize across various tasks, due to the inconsistent observation and action spaces. This indicates the necessity of the GCC setting, which provides a unified representation of the observation and action spaces for enormous challenging computer tasks.

Concurrent with our work, there are several works (Gao et al., 2023; Cheng et al., 2024; Niu et al., 2024; Zhang et al., 2024; Wu et al., 2024; Kapoor et al., 2024) aiming to scale the web agent to more applications with screenshots as input and keyboard and mouse operations as output. However, none of them takes video games into consideration since they mainly focus on static websites and software, which greatly reduces the need for timeliness and simplifies the setting by ignoring the dynamics between adjacent screenshots, *i.e.,* animations, and incomplete action space without considering the duration of the key pressed and different mouse mode.

A.2 Decision-Making in Video Games

Video games are ideal environments for validating agent's various abilities due to their diversity, controllability, safety, and reproducibility, which are also believed to be the most complicated tasks in computer control. Atari games (Bellemare et al., 2013), Super Mario Bros (Kauten, 2018), Google Research Football (Kurach et al., 2020), StarCraft II (Vinyals et al., 2019; Samvelyan et al., 2019), Minecraft (Johnson et al., 2016; Guss et al., 2019; Fan et al., 2022) etc, have been the popular environments and benchmarks for reinforcement learning (RL) agents. Besides, RL agents also exhibit impressive performance in Dota II (Berner et al., 2019), Quake III (Jaderberg et al., 2019), Gran Turismo (Wurman et al., 2022) and Diplomacy (Bakhtin et al., 2022). However, to abstract complex computer control, these environments usually simplify the whole action space (*i.e.,* keyboard and mouse movement) to pre-defined domain-specific actions, which vary from game to game, exacerbating the poor generalization of RL agents across environments.

Discarding the semantics in the observation and action also leads to low efficiency. LLMs enable decision-making agents to leverage semantic information in the environments, which dramatically improves the reasoning ability of agents, Without any training process, Voyager (Wang et al., 2023a) can efficiently learn to finish long-horizontal complex tasks through code generation. However, it heavily relies on the built-in API tool, Mineflayer, to obtain internal information and execute

high-level actions, which are not available in other games. TextStarCraft II (Ma et al., 2023a) and CivRealm (Qi et al., 2024) also suffer the same issue. Therefore, closed-source AAA games with rich textual and visual information are rarely to be explored. Pre-trained with videos with action labels, VPT (Baker et al., 2022) manages to output mouse and keyboard control with raw screenshots as input without any additional information. However, collecting videos with action labels is time-consuming and costly, which is difficult to generalize to multiple environments.

In a nutshell, to our best knowledge, there are currently no agents under the GCC setting, reported to show superior performance and generalization in complex video games or across computer tasks. In this work, we make a preliminary attempt to apply our framework to the epic AAA game RDR2, under the GCC setting.

B Red Dead Redemption II

B.1 Introduction to RDR2

Red Dead Redemption II (RDR2) is an epic AAA Western-themed action-adventure game by Rockstar Games. As one of the most famous and highest-selling games in the world, it is widely acknowledged for its movie-like realistic scenes, rich storylines, and immersive open-ended world. The game applies a typical role-playing game (RPG) control system, played from a first- or third-person perspective, which uses WASD for movement, mouse control for view changing, first- or third-person shooting for combat, and inventory and manipulation.

For most of the game, players need to control the main character, Arthur Morgan, upon choosing to complete mission scenarios following the main storyline. Otherwise, they can freely explore the interactive world, such as go hunting, fishing, chatting with non-player characters (NPCs), training horses, witnessing or partaking in random events, and participating in side quests. As the main storyline progresses, different skills are gradually unlocked. As a close-source commercial game, no APIs are available for obtaining additional game-internal information nor pre-defined automation actions. Following its characteristics, this game serves as a fitting and challenging environment for the GCC setting.

B.2 RDR2 Tasks

Table 1: Tasks in the first two missions of RDR2. *Difficulty* refers to how hard it is for our agent to accomplish the corresponding tasks. Figures 8 and 9 showcase snapshots of each task (specific sub-figures marked in parenthesis in the table).

Mission 1: Explore shelter	Description	Difficulty
Follow Dutch (Fig. 8a)	Follow Dutch to the small town, by riding the horse.	Easy
Hitch horse (Fig. 8b)	Dismount at the hitching post, after reaching the town.	Easy
Go to shed (Fig. 8c)	Move to the nearby shed to take cover.	Easy
Choose weapon (Fig. 8d)	Choose the correct weapon to prepare for combat.	Hard
Protect Dutch (Fig. 8e)	Protect Dutch from the enemies through shooting.	Hard
Search house (Fig. 8f)	Follow Dutch to enter the house and search for supplies.	Hard
Eat something (Fig. 8g)	Open Satchel and eat a provision to restore some Health Core.	Medium
Investigate barn (Fig. 8h)	Leave the horse and go to the barn to investigate.	Easy
Defeat O'Driscoll (Fig. 8i)	Fight with the enemy hidden in the barn.	Medium
Pick up equipment (Fig. 8j)	Find and pick up gun and hat lost during barn fight.	Hard
Lead horse (Fig. 8k)	Calm and Lead the horse out of the barn to hitching post.	Medium
Mission 2: Rescue John		
Follow Javier (Fig. 9a)	Follow Javier through the snow mountain to look for John.	Medium
Equip shotgun (Fig. 9b)	Equip correct shotgun to prepare for combat.	Medium
Look for John (Fig. 9c)	Crouch down, enter narrow tunnel, and climb up cliffs.	Hard
Shoot wolves (Fig. 9d)	Shoot wolves that will attack you and companions.	Hard
Protect Javier from wolves (Fig. 9e)	Ride horse and protect your companions from wolves.	Hard

Tables 1 and 2 provide a brief introduction of each task in the first two missions of the main game storyline and an open-ended mission, along with approximate estimates of their difficulty. Due to GPT-4V's poor performance in spatial understanding and fine-manipulation skills, it can be

(a) Follow Dutch

(b) Hitch horse

(c) Go to shed

(d) Choose weapon

(e) Protect Dutch

(f) Search house

(g) Eat something

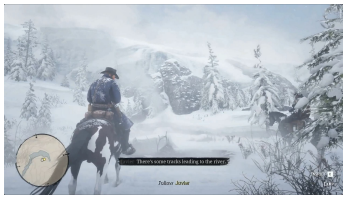
(h) Investigate barn

(i) Defeat O'Driscoll

(j) Pick up equipment

(k) Lead horse

Figure 8: Image examples of tasks in the first mission of *Explore shelter*. (The picture has been brightened for easier reading.)



(a) Follow Javier

(b) Equip shotgun

(c) Look for John

(d) Shoot wolves

(e) Protect Javier from wolves

Figure 9: Image examples of tasks in the second mission of *Rescue John*.

Table 2: Tasks in the open-ended mission, *Buy supply* in RDR2. *Difficulty* refers to how hard it is for our agent to accomplish the corresponding tasks. Figure 10 showcases snapshots of each task (specific sub-figures marked in parenthesis in the table).

Open-ended mission: Buy supply	Description	Difficulty
Find horse (Fig. 10a)	Find and mount the horse in the camp.	Medium
Prepare to navigate to saloon (Fig. 10b)	Open map, find the saloon and create waypoint.	Medium
Go to saloon (Fig. 10c)	Ride horse to the saloon.	Easy
Prepare to navigate to shop (Fig. 10d)	Open map, find the general store and create waypoint.	Medium
Go to shop (Fig. 10e)	Ride horse to the shop.	Hard
Enter shop (Fig. 10f)	Dismount the horse and enter the shop.	Hard
Talk to shopkeeper (Fig. 10g)	Approach the shopkeeper and talk.	Easy
Buy target product (Fig. 10h)	Open the menu, find and buy the target product.	Medium

(a) Find horse (b) Prepare to navigate to saloon (c) Go to saloon

(d) Prepare to navigate to shop (e) Go to shop (f) Enter shop

(g) Talk to shopkeeper (h) Buy target product

Figure 10: Image examples of tasks in the open-ended task of RDR2.

challenging for our agent to perform certain actions, like entering or leaving a building, or getting to precise indoor locations to retrieve specific items. Additionally, the high latency of GPT-4V's responses also makes it harder for an agent to deal with time-sensitive events, *e.g.,* during combat.

C CRADLE IN GCC

C.1 IMPLEMENTATION DETAILS

As shown in Figure 4, strictly following the GCC setting, our agent takes the video of the screen as input and outputs keyboard and mouse operations to interact with the computer and the game. An observation thread is responsible for the collection of video frames from the screen and each video clip records the whole in-game process since executing the last action.

We employ GPT-4V(ision) (Achiam et al., 2023), *gpt-4-vision-preview*, currently one of the most capable LMM models, as the framework's backbone model. To lower the frequency of interaction with backbone models, the video recorder captures a game screenshot every 0.5 seconds, which proves to be sufficient for information gathering without missing any important information.

Information Gathering. To extract keyframes from the video observation, we utilize the VideoSubFinder tool [4], a professional subtitle discovery and extraction tool. These keyframes usually contain rich meaningful textual information in the game, which are highly relevant to the completion of tasks and missions (such as character status, location, dialogues, in-game prompts and tips, etc.) We use GPT-4V to extract and categorize all the meaningful contexts in these keyframes and perform OCR, and call this processing "gathering text information". Then, to save interactions with GPT-4V, we only let GPT-4V provide a detailed description of the last frame of the video.

While GPT-4V exhibits impressive visual understanding abilities across various CV tasks, we find that it struggles with spatial reasoning and recognizing some game-specific icons. To address these limitations, we add a visual augmentation sub-module within our *Information Gathering* module. This augmentation step serves two main purposes: i) utilize Grounding DINO (Liu et al., 2023), an open-set object detector, to output precise bounding boxes of possible targets in an image and serve as spatial clues for GPT-4V; and ii) perform template matching (Brunelli, 2009) to provide icon recognition grounding truth for GPT-4V when interpreting instructions or menus shown on screen. As LMM capabilities mature, it should be possible to disable such augmentation.

Self-Reflection. The reflection module mainly serves to evaluate whether the previously executed action was successfully carried out and whether the current executing task is finished. To achieve this, we uniformly sample at most 8 sequential frames from the video observation since the execution of the last action and use GPT-4V to estimate the success of its execution. Additionally, we expect GPT-4V can also provide analysis for any failure of the last action (*e.g.,* the move-forward action failed and the cause could be the agent was blocked by an obstacle). With such valuable information as input for *Action Planning*, including the failure/success of the last action and the corresponding analysis, the agent is capable of attempting to remedy an inappropriate decision or action execution.

Moreover, some actions require prolonged durations, such as holding down specific keys, which can coexist or interfere with other actions decided by subsequent decisions. Consequently, the reflection module must also decide whether an ongoing action should continue to be executed. Furthermore, self-reflection can be leveraged to dissect why the last action failed to bring the agent close to the target task completion, better understand the factors that led to the successful completion of the preceding task, and so on.

Besides, we observe that instead of providing GPT-4V with sequential high-resolution images for self-reflection, low-resolution images make it easier for GPT-4V to understand the relation among the sequential screenshots and capture dynamic changes, resulting in a significantly higher success rate of detecting whether the action is executed successfully and take any effect. We hypothesize that since a high-resolution image can cost as many as 2000 tokens, too many high-resolution images make GPT-4V fail to capture the overall changes across screenshots and be caught up in the local details.

Task Inference. During gameplay, we let GPT-4V propose the current task to perform whenever it believes it is time to start a new task. GPT-4V also outputs whether the task is a long- or short-horizon task when proposing a new task. Long-horizon tasks, such as traveling to a location, typically require multiple iterations, whereas short-horizon tasks, like picking up an item or conversing with someone, involve fewer iterations. The agent will follow the newly generated task for the next h interactions. After h interactions, the agent returns to the last long-horizon task in the stack. Deciding on a binary task horizon is much easier and more robust for GPT-4V, than re-planning at every iteration. Since a long-horizon task frequently includes multiple short-horizon sub-tasks, this implementation also helps avoid forgetting the long-horizon tasks under execution.

To save interactions with GPT-4V, we implement this as part of the *Information Gathering* module. When GPT-4V detects instructional text in the recent observation, it will directly generate a task description and determine whether it is a long- or a shot-horizon task.

Skill Curation. As shown in Figure 6, during gameplay, instructions often appear on the screen, such as “press [Q] to take over" and “hold [TAB] to view your stored weapons," which serve as essential directives for completing current and future tasks proficiently. To translate these textual and iconic instructions into executable mouse and keyboard actions, we provide GPT-4V with skill code samples using interfaces for mouse and keyboard manipulation, *e.g.*, io_env.key_press (to press

[4] VideoSubFinder standalone tool - https://sourceforge.net/projects/videosubfinder/

a key), io_env.key_hold (to keep a key held), io_env.mouse_click (to click the mouse button), and io_env.mouse_hold (to keep the mouse button held). GPT-4V is required to strictly follow the provided interfaces and examples to generate the corresponding code for new skills. Moreover, GPT-4V is required to include documentation/comments within the generated code, delineating the functionality of each skill. The *Procedural Memory* sub-module where skills are stored will then check whether the code is valid, whether the format of documentation is right, and whether any skill with the same name already exists. If all conditions are passed, the newly generated skill is persisted for future utilization.

Similarly to *Task Inference*, we also implement a simple version of this module inside *Information Gathering* to reduce interactions with GPT-4V. When GPT-4V detects and classifies some instructional text in the recent observation, which usually contains key/button hints, it will directly generate the corresponding code and description.

Action Planning. Upon execution of this module, we first retrieve the top k relevant skills for the task from procedural memory, alongside the newly generated skills. We then provide GPT-4V with the current task, the set of retrieved skills, and other information collected in *Information Gathering* that may be helpful for decision-making (*e.g.*, recent screenshots with corresponding descriptions, previous decisions, and examples) and let it suggest which skills should be executed. We also request that GPT-4V provide the reasons to choose these skills, which increases the accuracy, stability, and explainability of skill selection and thus greatly improves framework performance. While GPT-4V sometimes may generate a sequence of actions, we currently only execute the first one, and perform *Self-Reflection*, since we observe a tendency for the second action to usually suffer from severe hallucinations.

Action Execution. For the action space, we categorize the keyboard and mouse actions into 4 key categories: `press(key, duration)`, `hold(key, duration)`, `release(key)`, and `pointer_move(x, y)`; which can be combined in different ways to form combos, use keys in fast sequence, or coordinate timings. Skill code needs to be generated by the agent in order to utilize such functions and affordances so executed actions take effect. Table 3 illustrates **CRADLE**'s action space.

It is important to note that, while some works (*e.g.,* AssistantGUI (Gao et al., 2023) and OmniACT (Kapoor et al., 2024)) use *PyAutoGUI* [6] for keyboard and mouse control, this approach does not work in all applications, particularly in modern video games using DirectX [7]. Moreover, such work chooses to expose a subset of the library functionality in its action space, ignoring dimensions like press duration and movement speed, which are critical in many scenarios.

To ensure wide game compatibility and avoid interference between different code libraries, in our current implementation we use the similar *PyDirectInput* library [8] only for keyboard control, and write our own abstraction for mouse control (using the *ctypes* library [9] to send low-level mouse commands to the operating system. For increased portability and ease of maintenance, all keyboard and mouse control is encapsulated in a class, called *IO_env*.

Unlike the conventional mouse operation in standard software, where the cursor is restricted to a 2D grid and remains visible on the screen to navigate and interact with elements, the utilization of the mouse in 3D games like RDR2 introduces a varied control scheme. In menu screens, the mouse behaves traditionally, offering familiar point-and-click functionality. However, during gameplay, the mouse cursor disappears, requiring players to move the mouse according to specific action semantics. For example, to alter the character's viewpoint, the player needs to map the actual mouse movement to in-game direction angle changes, which differ in magnitude in the X and Y axes. Another special transition applies to shooting mode, where the front sight is fixed at the center of the screen, and players must maneuver the mouse to align the sight with target enemies. This nuanced

[6] Python library that provides a cross-platform GUI automation module - https://github.com/asweigart/pyautogui

[7] Microsoft DirectX graphics provides a set of APIs for high-performance multimedia apps - https://learn.microsoft.com/en-us/windows/win32/directx

[8] Python library encapsulating Microsoft's *DirectInput* calls for convenience manipulating keyboard keys - https://github.com/learncodebygaming/pydirectinput

[9] Python library that provides C compatible data types, and allows calling functions in DLL/.so binaries - https://docs.python.org/3/library/ctypes.html

Table 3: Action space in the **CRADLE** framework, including action attributes. Coordinate system is either *absolute* or *relative*. Actions with durations can be either *synchronous* or *asynchronous*.

Type	Action	Attributes
Keyboard	Key Press	Key name (string), Key press duration (seconds:float)
	Key Hold	Key name (string), Key press duration (seconds:float), Wait behaviour (sync/async)
	Key Release	Key name (string)
	Key Combo	Key names (strings), Key combo duration (seconds:float), Wait behaviour (sync/async)
	Hotkey	Key names (strings), Hotkey sequence duration (seconds:float), Wait behaviour (sync/async)
	Text Type	String to type (string), Typing duration (seconds:float)
Mouse	Button Click	Mouse button (left/middle/right), Button click duration (seconds:float)
	Button Hold	Mouse button (left/middle/right), Button hold duration (seconds:float), Wait behaviour (sync/async)
	Button Release	Mouse button (left/middle/right)
	Button Double Click	Mouse button (left/middle/right), Button click duration (seconds:float)
	Move/Hover/Point	Mouse position (width:int, height:int), Mouse speed (seconds:float), Coordinate system (enum), Tween mode (enum) [5]
	Drag	Mouse final position (width:int, height:int), Mouse speed (seconds:float), Coordinate system (enum), Tween mode (enum)
	Scroll	Orientation (vertical/horizontal), Distance (clicks:int), Duration (seconds:float)
Wait Action		Wait time (float)

approach to mouse control in different contexts adds an extra layer of challenge to general computer handling, showcasing the adaptability required in game environments, compared to regular software applications.

Procedural Memory. Procedural memory stores pre-defined basic skills and the generated skills captured from the *skill curation*. However, as we continuously obtain new skills during game playing, the number of skills in procedural memory keeps increasing, and it is hard for GPT-4V to precisely select the most suitable skill from the large memory. Thus, we propose first to retrieve a subset of skills, that are relevant to the given task, and then let GPT-4V select the most suitable one from the subset. In the skill retrieval, we pre-compute the embeddings of the documentations (code, comments and descriptions) of skill functions, which describe the skill functionality, and compute the embedding of the given task. Then we compute the cosine similarities between the skill documentation embeddings and the task embedding. The higher similarity means that the skill's functionality is more relevant to the given task. We select the most similar 10 skills as the subset. Using similarity matching to select a small candidate set simplifies the process of choosing skills. We use OpenAI's *text-embedding-ada-002 model* to compute embeddings for the skill information.

In our target setting, We intend to let the agent learn all skills from scratch, to the extent possible for the main storyline missions. The procedural memory is initialized with only preliminary skills for basic movement, which are not clearly provided by the in-game tutorial and guidance.

- *turn(degree)*, *move_forward(duration)*: Since the game does not precisely introduce how to move in the world through in-game instructions, we provide these two basic actions in advance, so GPT-4V can perform basic mobility, while greatly reducing the number of calls to the model.
- *shoot(x, y)*: RDR2 also does not provide detailed instructions on how to aim and shoot. Moreover, due to limitations with GPT-4V spatial reasoning and the need to sometimes augment images with object bounding boxes, we provide such basic skill for the agent to complete relevant tasks.
- *select_item_at(x, y)*: Similarly to *shoot()*, due to the lack of instructions, we provide such skill for the agent to move the mouse to a certain place to select a given item.

Beyond these basic atomic low-level actions, we introduce a few composite skills to facilitate the game playing progress. The agent should be able to complete tasks using only the basic skills above and the skills it learns, but these composite skills streamline the process by greatly reducing calls to the backend model.

- *turn_and_move_forward(degree, duration)*: This skill is just a simple composition of *turn()* and *move_forward()* to save frequent calls to GPT-4V in a common sequence.
- *follow(duration)* and *navigate_path(duration)*: In RDR2, tasks often guide players to follow NPCs or generated paths (red lines) in the minimap to certain locations. This can be reliably accomplished via the basic movement skills, but requires numerous interactions with GPT-4V. To control both cost and time budgets involving GPT-4V's responses, we leverage the information shown in the minimap to implement a composite skill to follow target NPCs or red lines for a short set of game iterations.
- *fight()*: As output of an interaction with GPT-4V, the agent will only take one action per step. However, though the action is generated correctly, specifically in fight scenarios, the action frequency may not be high enough to defeat an opponent. In order to allow sub-second punches, we provide a pre-defined action that wraps this multi-action punching, which can be selected by GPT-4V to effectively win fights.

For the open-ended mission, since the agent skips all the tutorials in Chapter I, we provide all the necessary skills in the procedural memory at the beginning of the mission.

Episodic Memory stores all the useful information provided by the environment and LMM, which consists of short-term memory and long-term summary.

The short-term memory stores the screenshots within the recent k interactions in game playing and the corresponding information from other modules, *e.g.,* screenshot descriptions, task guidance, actions, and reasoning. We set k to five, and it can be regarded as the memory length. Information

stored over k interactions ago will be forgotten from direct short-term memory. Empirically, we found that recent information is crucial for decision-making, while a too-long memory length would cause hallucinations. In addition, other modules continuously retrieve recent information from short-term memory and update the short-term memory by storing the newest information.

For some long-horizon tasks, short-term memory is not enough. This is because the completion of a long-horizon task might require historical information from a long steps ago. For example, the agent might do a series of short-horizon tasks during a long-horizon task, which makes the original long-horizon task forgotten in short-term memory. To maintain the long-term valuable information while avoiding the long-token burden of GPT-4V, we propose a recurrent information summary as long-term memory, which is the text summarization of experiences in game playing, including the ongoing task, the past entities that the player met, and the past behaviors of the player and NPCs.

In more detail, we provide GPT-4V with the summarization before the current screenshot and the recent screenshots with corresponding descriptions, and GPT-4V will make a new summarization by organizing the tasks, entities, and behaviors in the time order with sentence number restriction. Then we update the summarization to be the newly generated one, which includes the information in the current screenshot. The recurrent summarization update, inspired by RNN, achieves linear-time inference by preserving a hidden state that encapsulates historical input. This method ensures the compactness of summarization token lengths and recent input data. Furthermore, the incorporation of long-term memory enables the agent to effectively retain crucial information over extended periods, thereby enhancing decision-making capabilities.

C.2 Application Target and Setting Challenges

Choosing a complex game like RDR2 introduces its own set of challenges beyond just the GCC setting and a complex application target.

C.2.1 Model latency and Game pause

RDR2 is a dynamic game where events happen in a real-time-like manner. As such, it is unfeasible to simply wait for the latency in GPT-4V responses. During this time any event could happen in the game and invalidate the state passed to the backend model. To sidestep this issue we utilize the *pause* feature in the game, similar to works like Voyager (Wang et al., 2023a), to freeze the game until a response from the backend model is received. However, differently from games like Minecraft, RDR2 doesn't provide instantaneous pause/resume. This leads to additional latency before and after actions. Moreover, pausing the game interferes with the ongoing keyboard/mouse state, which must be reset on the resume.

C.2.2 Mouse modes

Differently from traditional applications which use a flat 2D grid-based GUI (from web apps to productivity suites), games, and especially 3D games, make heavy use of different styles or modes of mouse interaction. Movement of the mouse can be mapped to arcs in a 3D view cone, distances moved on the X and Y axis may be non-uniform and differ across scenes in the game, mouse coordinates may move beyond the game screen region, etc. In order to successfully manipulate the game, both the action space must provide for functionality to address these issues and the agent itself must be able to effectively leverage such affordances.

In RDR2, for example, the mouse movement behavior is different in map-mode, regular gameplay, and when aiming the weapon in hunting/combat situations.

C.2.3 Multi-episodes

Agent development in digital environments can also usually greatly benefit from multi-episode simulation and learning. But this requires being able to precisely control the environment where the agent is executed and reset it to exact conditions between runs.

RDR2 again is a more challenging environment, where: i) saving the state of the game is not freely allowed (only at specific milestones), and ii) saving/re-loading the game does not precisely preserve the state of the game. For example, if an agent moves to a location in the woods, opens the map, and

creates a path on it to a third location, even if allowed to save the game, when the game resumes, the map is closed and no path is defined, as well as the location of the agents reverts to the last game checkpoint location (not where the save happened).

C.2.4 Action Execution and Feedback

Proper reasoning about environment feedback is critical due to the generality of the GCC setting and the level of abstraction to interact with the complex game world. The semantic gaps between the execution of an action, its effects in the game world, and observing the relevant outcomes for further reasoning lead to several potential issues that **CRADLE** needs to deal with. Such issues can be categorized into four major cases:

Lack of grounding feedback. In many situations, due to the lack of precise information from the environment, it can be difficult for the system to deduce the applicability or outcome of a given action. For example, when picking an item from the floor, the action may fail due to the distance to the object not yet being close enough. Or, if within pick up range, the chosen action may not exactly apply due to other factors (*e.g.,* character's package is full).

Even if the right action is selected and executed successfully, the agent still needs to figure out its results from the partial visual observation of the game world. If the agent needs to pick or manipulate an object that is occluded from view, the action may execute correctly, but no outcome can be seen.

A representative example in RDR2 happens when the agent tries to pick up its gun from the floor after a fight. Getting to the right distance, without completely occluding the object, can lead to multiple re-trials. Figure 11a showcases a situation where, though the character is already standing near the gun (as seen in the minimap), it's still not possible to pick it up.

Previous efforts (Wang et al., 2023c;a) that utilize in-game state APIs unreasonably bypass such issues by leveraging internal structured information from the game and the full semantics of responses (data) or failures (error messages).

Imprecise timing in IO-level calls. This issue is caused by the ambiguity in the game instructions or differences in specific in-game action behaviors, where even the execution of a correct action may fail due to minor timing mismatches. For example, when executing an action like 'open cabinet', which requires pressing the 'r' key on the keyboard, if the press is too fast, no effect happens in the game world. However, as there is no visual change in the game nor other forms of feedback, it can be difficult for GPT-4V to figure out if an inappropriate action was chosen at this game state or if the minor timing factor was the problem. Pressing the key for longer triggers an animation around the button (only if the helper menu is on screen), but this is easily missed and any key release before the circle completes also results in no effect. Figure 11b illustrates the situation.

The same problem also manifests in other situations in the game, where pressing the same key for longer triggers a completely different action (*e.g.,* lightly pressing the 'left alt' key vs. holding it for longer).

Change in the semantics of key and button. A somewhat similar situation occurs when the same keyboard key or mouse button gets attributed different semantics in different situations (or even in a multi-step action). GPT-4V may decide to execute a given skill, but the original semantics no longer hold. The lack of in-game effect parallels the previous situations. Worse yet, an undesired effect will confuse the system regarding the correct action being selected or not.

For example, when approaching a farm in the beginning of the game, the agent needs to hitch the horse to a pole to continue. The operation to perform the action consists of pressing the 'e' key near a hitching post (as shown in Figure 11c). However, the same 'e' key press is the only constituting step in other actions with different semantics, like *dismount the horse* or *open the door*. Wrongly triggering a horse dismount at the situation shown in the figure can lead to undesired side effects, *i.e.,* it may mislead the system about the actual effects of the action or affect the planning of which next actions to perform.

Interference issues. Lastly, completion of some actions requires the correct execution of multiple steps sequentially, which could be interrupted in many ways not related to the agent's own actions. Without the use of APIs that expose internal states or other forms of feedback, it is much harder for the agent to decide when to repeat sub-actions or try different strategies. For example, if the agents

(a) ‘Pick gun’ unavailable. (b) ‘Open cabinet’ press timing. (c) ‘Hitch horse’ re-use of ‘e’ key

Figure 11: Examples of action execution uncertainty. Lack of environment feedback to actions and semantic gaps between action intent and game command can lead to challenging situations for agent reasoning.

gets shot and loses aim while in combat, or an unrelated in-game animation is triggered mid-action, cancelling it.

Since there is no direct environment feedback, the agent needs to carefully analyze the situation and try to infer if any action step needs re-execution.

C.3 CRADLE PROMPTS

Prompt 1: Gather Text Information prompt.

```
Assume you are a helpful AI assistant integrated with 'Red Dead
    Redemption 2' on the PC, equipped to handle a wide range of tasks in
    the game. Your advanced capabilities enable you to process and
    interpret gameplay screenshots and other relevant information.

<$image_introduction$>

Information: List all text prompts on the screenshot from the top to the
    bottom, even the text prompt is one word.

All information should be categorized into one or more kinds of <
    $information_type$>. If you think a piece of information is both "A"
    and "B" categories, you should write information in both "A" and "B"
    categories. For example, "use E to drink water" could both be "Action
     Guidance" and "Task Guidance" categories.

Item_status: The helpful information to the current context in the game,
    such as the cash, amount of ammo, current using item, if the player
    is wanted, etc. This content should be pairs of status names and
    their values. For example, "cash: 100$". If there is no on-screen
    text and no item status, only output "null".

Environment_information: The information about the location, time,
    weather, etc. This content should be pairs of status names and their
    values. For example, "location: VALENTINE". If there is no on-screen
    text and environment information, only output "null".

Notification: The game will give notifications showing the events in the
    world, such as obtaining items or rewards, completing objectives, and
     becoming wanted. Besides, it also contains valuable notifications of
     the game's mechanisms, such as "Health is displayed in the lower
    left corner". The content must be the on-screen text. If there is no
    on-screen text or notification, only output "null".

Task_guidance: The content should obey the following rules:
1. The content of task guidance must be an on-screen text prompt,
    including the menu and the general game interface.
2. The game will give guidance on what should be done to proceed with the
     game, for example, "follow Tom". This is task guidance.
3. The game will give guidance on how to perform a task using keyboard
    keys or mouse buttons, for example, "use E to drink water". This is
    task guidance.
4. If no on-screen text prompt or task guidance exists, only output "null
    ". Never derive the task guidance from the dialogue or notifications.

Action_guidance: The game will give guidance on how to perform a task
    using keyboard keys or mouse buttons; you must generate the code
    based on the on-screen text. The content of the code should obey the
    following code rules:
1. You should first identify the exact keyboard or mouse key represented
    by the icon on the screenshot. 'Ent' refers to 'enter'. 'RM' refers
    to 'right mouse button'. 'LM' refers to 'left mouse button'. You
    should output the full name of the key in the code.
2. You should refer to different examples strictly based on the word used
     to control the key, such as 'use', 'hold', 'release', 'press', and '
    click'.
3. If 'use' or 'press' is in the prompt to control the keyboard key or
    mouse button, io_env.key_press('key', 2) or io_env.mouse_press('
    button', 2) must be used to act on it. Refer to Examples 1, 2, and 3.
4. If there are multiple keys, io_env.key_press('key1,key2', 2) must be
    used to act on it. Refer to Example 4.
```

````
5. If 'hold' is in the prompt to control the keyboard key or mouse button
    , it means keeping the key held with io_env.key_hold or the button
    held with io_env.mouse_hold (usually indefinitely, with no duration).
     If you need to hold it briefly, specify a duration argument. Refer
    to Examples 5 and 6.
6. All durations are set to a minimum of 2 seconds by default. You can
    choose a longer or shorter duration. If it should be indefinite, do
    not specify a duration argument.
7. The name of the created function should only use phrasal verbs, verbs,
     nouns, or adverbs shown in the prompt and should be in the verb+noun
     or verb+adverb format, such as drink_water, slow_down_car, and
    ride_faster. Note that words that do not show in the prompt are
    prohibited.

This is Example 1. If "press" is in the prompt and the text prompt on the
     screenshot is "press X to play the card", your output should be:
```python
def play_card():
    """
    press "x" to play the card
    """
    io_env.key_press('x', 2)
```
This is Example 2. If the instructions involve the mouse and the text
    prompt on the screenshot is "use the left mouse button to confirm",
    your output should be:
```python
def confirm():
    """
    use "left mouse button" to confirm
    """
    io_env.mouse_press("left mouse button")
```
This is Example 3. If "use" is in the prompt and the text prompt on the
    screenshot is "use ENTER to drink water", your output should be:
```python
def drink_water():
    """
    use "enter" to drink water
    """
    io_env.key_press('enter', 2)
```
This is Example 4. If "use" is in the prompt and the text prompt on the
    screenshot is "use W and J to jump the barrier", your output should
    be:
```python
def jump_barrier():
    """
    use "w" and "j" to jump the barrier
    """
    io_env.key_press('w,j', 3)
```
This is Example 5. If "hold" is in the prompt and the text prompt on the
    screenshot is "hold H to run", your output should be:
```python
def run():
    """
    hold "h" to run
    """
    io_env.key_hold('h')
```
This is Example 6. If the instructions involve the mouse and the text
    prompt on the screenshot is "hold the right mouse button to focus on
    the target", your output should be:
```python
````
````

````
def focus_on_target():
    """
    hold "right mouse button" to focus
    """
    io_env.mouse_hold("right mouse button")
```
This is Example 7. If "release" is in the prompt and the text prompt on
    the screenshot is "release Q to drop the items", your output should
    be:
```python
def drop_items():
    """
    release "q" to drop the items
    """
    io_env.key_release('q')
```

Dialogue: Conversations between characters in the game. This content
    should be in the format of "character name: dialogue". For example, "
    Arthur: I'm fine". If there is no on-screen text or dialogue, only
    output "null".

Other: Other information that does not belong to the above categories. If
     there is no on-screen text, only output "null".

Reasoning: The reasons for classification for each piece of information.
If the on-screen text prompt is an instruction on how to perform a task
    using keyboard keys or mouse buttons, it should also classified as
    action guidance and task guidance.
For action guidance, which code rules should you follow based on the word
     used to control the key or button, such as press, hold, release, and
     click?

The information should be in the following categories, and you should
    output the following content without adding any other explanation:
Information:
1. ...
2. ...
...
Reasoning:
1. ...
2. ...
...
Item_status:
Item_status is ...
Environment_information:
Environment information is ...
Notification:
Notification is ...
Task_guidance:
Task is ...
Action_guidance:
```python
Python code to execute
```
```python
Python code to execute
```
...
Dialogue:
Dialogue is ...
Other:
Other information is ...
````
````

Prompt 2: Gather Situation Information prompt.

```
Assume you are a helpful AI assistant integrated with 'Red Dead
    Redemption 2' on the PC, equipped to handle a wide range of tasks in
    the game. Your advanced capabilities enable you to process and
    interpret gameplay screenshots and other relevant information.

<$few_shots$>

<$image_introduction$>

Current task:
<$task_description$>

Target_object_name: Assume you can use an object detection model to
    detect the most relevant object for completing the current task if
    needed. What object should be detected to complete the task based on
    the current screenshot and the current task? You should obey the
    following rules:
1. The object should be relevant to the current target or the
    intermediate target of the current task. Just give one name without
    any modifiers.
2. If no explicit weapon is specified in the weapon interface, prioritize
     choosing 'gun' as the weapon.
3. If no explicit shoot target is specified, prioritize choosing 'person'
     as the target.
4. If no explicit item is specified, only output "null".
5. If the object name belongs to the person type, replace it with 'person
    '.
6. If there is no need to detect an object, only output "null".
7. If you are on the trade or map interfaces, only output "null".

Reasoning_of_object: Why was this object chosen, or why is there no need
    to detect an object?

Description: Please describe the screenshot image in detail. Pay
    attention to any maps in the image, if any, especially critical icons
    , red paths to follow, or created waypoints. If there are multiple
    images, please focus on the last one.

Screen_classification: Please select the class that best describes the
    screenshot among "Inventory", "Radial menu", "Satchel", "Map", "Trade
    ", "Pause", and "General game interface without any menu". Output the
     class of the screenshot in the output of Screen_classification.

Reasoning_of_screen: Why was this class chosen for the current screenshot
    ?

Movement: Does the current task require the character to go somewhere?

Noun_and_Verb: The number of nouns and verbs in the current task.

Task_horizon: Please judge the horizon of the current task, i.e., whether
     this task needs multiple or only one interaction.
There are two horizon types: long-horizon and short-horizon. For long-
    horizon tasks, the output should be 1. For short-horizon tasks, the
    output should be 0. You should obey the following rules:
1. If the task contains only nouns without verbs, it is short-horizon.
2. If the task contains more than one verb, it is long-horizon.
3. If the task requires the character to go somewhere, it is long-horizon
    .
Short-horizon tasks are sub-goals during a long-horizon task, which only
    need one interaction. There are some examples of short-horizon tasks:
1. Pick up something: To complete this task, the character needs to
    execute the action "pick up" only once, so it is short-horizon.
```

```
2. Use or press [B] key: The character needs to press the key [B] only
    once to talk, so it is short-horizon.
3. Talk to somebody: The character needs to press a certain button once
    to complete this task, so it is short-horizon.
Long-horizon tasks are long-term goals, which usually need many
    interactions. There are some examples of long-horizon tasks.
1. Go outside: The character should go outside step by step, so it is
    long-horizon.
2. Approach something: The character should move closer to the target
    step by step, so it is long-horizon.
3. Keep away from something, shoot, take down, or battle with something:
    The character must engage in a series of interactions, so it is long-
    horizon.

Reasoning_of_task: Why do you make such a judgment of task_horizon?

You should only respond in the format described below and not output
    comments or other information.
Target_object_name:
Name
Reasoning_of_object:
1. ...
2. ...
...
Description:
The image shows...
Screen_classification:
Class of the screenshot
Reasoning_of_screen:
1. ...
2. ...
...
Movement:
Yes or No
Noun_and_Verb:
1 noun 1 verb
Task_horizon:
1
Reasoning_of_task:
1. ...
2. ...
...
```

Prompt 3: Information Summary prompt.

```
Assume you are a helpful AI assistant integrated with 'Red Dead
    Redemption 2' on the the PC, equipped to handle a wide range of tasks
     in the game. You will be sequentially given <$event_count$>
    screenshots and corresponding descriptions of recent events. You will
     also be given a summary of the history that happened before the last
     screenshot. You should assist in summarizing the events for future
    decision-making.

The following are <$event_count$> successive screenshots and
    corresponding descriptions:

<$image_introduction$>

The following is the summary of history that happened before the last
    screenshot:
<$previous_summarization$>

Current task:
<$task_description$>
```

```
Info_summary: Based on the above input, please make a summary from the
    screenshots with descriptions and the history in no less than 10
    sentences, following the rules below.
1. Summarize the tasks from the history and the current task, with a
    special note on the method of crucial press operations.
2. Summarize the entities and behaviors mentioned in the successive
    descriptions.
3. If entities and behaviors in the history and screenshots are missed in
     the descriptions, please add them to the summarization.
4. Organize the summarization as a story in order of time, including the
    past entities and behaviors.
5. Only give descriptions; do not provide suggestions.

Entities_and_behaviors: Entities and behaviors which are summarized, e.g
    ., The entities include the player's character, the target character,
     and horses for both the player and the target. The behaviors consist
     of the player character riding horseback, following the target on
    horseback, and moving forward to maintain a distance behind the
    target.

The output should be in the following format:
Info_summary:
The summary is...
Entities_and_behaviors:
The summary is...
```

Prompt 4: Self-Reflection prompt.

```
Assume you are a helpful AI assistant integrated with 'Red Dead
    Redemption 2' on the PC, equipped to handle a wide range of tasks in
    the game. Your advanced capabilities enable you to process and
    interpret gameplay screenshots and other relevant information. Your
    task is to examine these inputs, interpret the in-game context, and
    determine whether the executed action takes effect.

Current task:
<$task_description$>

Last executed action:
<$previous_action$>

Implementation of the last executed action:
<$action_code$>

Error report for the last executed action:
<$executing_action_error$>

Reasoning for the last action:
<$previous_reasoning$>

Valid action set in Python format to select the next action:
<$skill_library$>

<$image_introduction$>

Reasoning: You need to answer the following questions step by step to get
     some reasoning based on the last action and sequential frames of the
     character during the execution of the last action.
1. What is the last executed action not based on the sequential frames?
2. Was the last executed action successful? Give reasons. You should
    refer to the following rules:
- If the action involves moving forward, it is considered unsuccessful
    only when the character's position remains unchanged across
    sequential frames, regardless of background elements and other people
    .
```

```
3. If the last action is not executed successfully, what is the most
    probable cause? You should give only one cause and refer to the
    following rules:
- The reasoning for the last action could be wrong.
- Not holding enough time should not be considered in this part.
- If it is an interaction action, the most probable cause was that the
    action was unavailable or not activated at the current place.
- If it is a movement action, the most probable cause was that you were
    blocked by seen or unseen obstacles.
- If there is an error report, analyze the cause based on the report.

You should only respond in the format as described below:
Reasoning:
1. ...
2. ...
3. ...
...
```

Prompt 5: Action Planning prompt.

```
You are a helpful AI assistant integrated with 'Red Dead Redemption 2' on
     the PC, equipped to handle various tasks in the game. Your advanced
    capabilities enable you to process and interpret gameplay screenshots
     and other relevant information. By analyzing these inputs, you gain
    a comprehensive understanding of the current context and situation
    within the game. Utilizing this insight, you are tasked with
    identifying the most suitable in-game action to take next, given the
    current task. You control the game character and can execute actions
    from the available action set. Upon evaluating the provided
    information, your role is to articulate the precise action you would
    deploy, considering the game's present circumstances, and specify any
     necessary parameters for implementing that action.

Here is some helpful information to help you make the decision.

Current task:
<$task_description$>

Memory examples:
<$memory_introduction$>

<$few_shots$>

<$image_introduction$>

Last executed action:
<$previous_action$>

Reasoning for the last action:
<$previous_reasoning$>

Self-reflection for the last executed action:
<$previous_self_reflection_reasoning$>

Summarization of recent history:
<$info_summary$>

Valid action set in Python format to select the next action:
<$skill_library$>

Minimap information:
<$minimap_information$>

Based on the above information, you should first analyze the current
    situation and provide the reasoning for what you should do for the
```

```
    next step to complete the task. Then, you should output the exact
    action you want to execute in the game. You should respond to me with
    :

Reasoning: You should think step by step and provide detailed reasoning
    to determine the next action executed on the current state of the
    task. You need to answer the following questions step by step. You
    cannot miss the question number 13:
    1. Only answer this question when the catalogue, menu, map, or
    inventory are open. You should first describe each item in the screen
     line by line, from the top left and moving right. Is the target item
     in the current screen?
    2. Only answer this question when the catalogue, menu, map, or
    inventory are open. Which item is selected currently?
    3. Only answer this question when the character is visible in the
    screenshot of the current step. Where is the character in the
    screenshot of the current step?
    4. Where is the target in the screenshot of the current step based on
     the task description, on the left side or on the right side? Does it
     appear in the previous screenshots?
    5. Are there any bounding boxes with coordinates values and object
    labels, such as "door x = 0.5, y = 0.5", shown in the screenshot? The
     answer must be based only on the screenshot of the current step, not
     from any previous steps. If the answer is no, ignore the questions 6
     to 8.
    6. You should first describe each bounding box, from left to right.
    Which bounding box is more relevant to the target?
    7. What is the value x of the most relevant bounding box only in the
    current screenshot? The value is the central coordination (x,y) of
    the central point of the box.
    8. Based on the few shots and the value x, where is the relevant
    bounding box in the current screenshot? Clearly on the left side,
    slightly on the left side, in the center, slightly on the right side,
     or clearly on the right side?
    9. Only answer this question when the catalogue, menu, map, or
    inventory are not open. Summarize the contents of recent history,
    mainly focusing on the historical tasks and behaviors.
    10. Only answer this question when the catalogue, menu, map, or
    inventory are not open. Summarize the content of self-reflection for
    the last executed action, and do not be distracted by other
    information.
    11. What was the previous action? If the previous action was a turn,
    was it a left or a right turn?
    12. Based on Actions Rule 12, do you need to consider or ignore the
    angle information from the minimap? If considering it, summarize the
    content of the minimap information.
    13. This is the most critical question. Based on the action rules and
     self-reflection, what should be the most suitable action in the
    valid action set for the next step? You should analyze the effects of
     the action step by step.

Actions: The best action, or short sequence of actions without gaps, to
    execute next to progress in achieving the goal. Pay attention to the
    names of the available skills and to the previous skills already
    executed, if any. You should also pay more attention to the following
     rules:
    1. You should output actions in Python code format and specify any
    necessary parameters to execute that action. If the function has
    parameters, you should also include their names and decide their
    values, like "move(duration=1)". If it does not have a parameter,
    just output the action, like "mount_horse()".
    2. Given the current situation and task, you should only choose the
    most suitable action from the valid action set. You cannot use
    actions that are not in the valid action set to control the character
    .
```

```
    3. If the target is not on the catalogue, menu, or inventory, you
    MUST choose the skill 'view_next_page'. For the map, ignore the skill
     'view_next_page'.
    4. If the minimap information exists, it may include angle
    information for red points, yellow points, or yellow regions. Angle
    information specifies the direction of the corresponding point or
    area. A negative angle indicates the left side, while a positive
    value signifies the right side. If the angle is 30, the corresponding
     point or area is 30 degrees to the character's right. If the angle
    is -50, the corresponding point or area is 50 degrees to the
    character's left. Do not doubt the correctness of these angles; you
    can refer to them when you approach these points or regions.
    5. When you decide to control the character to move, if the relevant
    bounding box is clearly on the left side in the current screenshot,
    you MUST turn left with a big degree. If the relevant bounding box is
     slightly on the left side in the current screenshot, you MUST turn
    left with a small degree. If the relevant bounding box is clearly on
    the right side in the current screenshot, you MUST turn right with a
    big degree. If the relevant bounding box is slightly on the right
    side in the current screenshot, you MUST turn right with a small
    degree. If the relevant bounding box is on the central side of the
    current screenshot, you can choose to move forward.
    6. When you decide to control the character to move, if yellow
    regions or yellow points exist in minimap information, they are
    related to the current task or instruction. This implies that you
    should approach within the yellow region or approach the yellow
    points. You can refer to the corresponding angle information when
    deciding to approach these regions or points. If red points exist in
    the minimap information, they are also related to the current task or
     instruction. This implies that you should turn towards them, and you
     can also refer to the corresponding angle information.
    7. When you decide to control the character to move, if minimap
    information does not exist, the 'theta' you use to turn MUST be more
    than 10 degrees and less than 60 degrees.
    8. When you decide to control the character to move, if you are in a
    normal road condition, the 'duration' you use to move forward should
    be 1 second. If you have bad road conditions, such as snow, and grass
    , that can slow you down, the 'duration' you use to move forward
    should be 2 second.
    9. When you are exploring or searching a place, if you are leaving
    the place, you MUST make a sharp turn to face the inside of the place
    . Any values for degrees are allowed.
    10. If upon self-reflection you think the last action was unavailable
     at the current place, you MUST move to another place.
    11. If upon self-reflection you think you were blocked, you MUST make
     a moderate turn in the same direction as the previous turn action
    and move forward, so that you can pass obstacles.
    12. You MUST ignore the angle information provided by the minimap in
    the following situations: when you think you were blocked based on
    self-reflection or when you were inside the highlighted area in the
    minimap.
    13. When you are indoors, or the current task does not imply
    following, you MUST not use the follow action.
    14. When you are outdoors, and the current task implies following,
    you MUST use the follow action.
    15. If the game fails, you MUST retry from the latest checkpoint, not
     restart from the beginning of the mission.

You should only respond in the format described below, and you should not
     output comments or other information:
Reasoning:
1. ...
2. ...
3. ...
Actions:
```

```python
    action(args1=x,args2=y)
```

D LIMITATIONS OF GPT-4V

Deploying **CRADLE** in this complex game, RDR2, requires the backbone LMM model to deal with multimodal input, which revealed several limitations of GPT-4V that needs external tools to provide additional groundings to improve the overall framework performance, as mentioned in Appendix C.1.

Spatial Perception. As shown in Figure 12a, GPT-4V's spatial-visual recognition capability is insufficient for precise fine-grained control, particularly in detecting whether the character is being or going to be blocked and estimating the accurate relative positions of target objects. These led to the need to augment certain images to provide auxiliary visual clues for decision-making, *i.e.,* bounding boxes of possible target objects.

Icon Understanding. GPT-4V also struggles with domain-specific concepts, such as unique icons within the game, which may represent specific targets or refer to certain mouse and key actions. As shown in Figure 12b, GPT-4V fails to recognize the left shift, right mouse button, and space icons. Attempts to incorporate few-shot learning to improve image understanding cannot be generalized. Therefore, we match prepared pattern templates, *e.g.,* icon images, against each screenshot to continuously detect and highlight any appearing icons.

Figure 12c also demonstrates that GPT-4V cannot understand the mini-map correctly, which usually contains key information about the position and direction of the character. Failure of localization leads the agent to sometimes get lost in the town and miss the task target. Although the above issues can be slightly alleviated by providing additional few-shot examples, it can only have an obvious effect if we crop the image and provide the GPT-4V with the region exactly containing the icon to be recognized, which makes the issue intractable.

History Processing. Moreover, GPT-4V can easily get distracted by irrelevant information in longer contexts, resulting in hallucinations. For example, when action planning utilizes too many historical screenshots, GPT-4V may confuse past and present frames. Additionally, GPT-4V frequently generates output not adhering to the rules in the provided prompts. To mitigate the issue of hallucinations, we more strictly control input information by further summarizing long-term memory.

World Understanding. Lastly, the absence of an RDR2 world model limits GPT-4V's understanding of the consequences of its actions in the game. This often results in inappropriate action selection, such as overestimating the necessary adjustments for aligning targets or misjudging the duration required for certain actions. To alleviate this problem, we introduced extra prompt rules regarding action parameters and more flexibility into the self-reflection module.

ChatGPT

There are **no immediate obstacles** directly in front of the character in the image. The character has a **clear path ahead** on a muddy street in what appears to be a Western-style town setting.

ChatGPT

The shopkeeper in the image is on the **right side**, behind the counter.

(a) GPT-4V fails to recognize obstacles in the environment and the position of the shopkeeper.

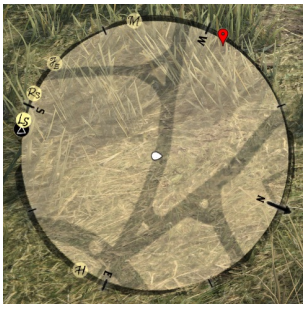
ChatGPT

Based on the provided game screenshot, I've identified the following control prompt:
- "**Hold L** ⬆ to ride faster"
This control suggests that holding a certain key (which seems to be an **upward arrow key**, often associated with the **'W' key** for forward movement in many games) will make the character ride faster.

ChatGPT

From the provided image screenshot from the game, I can identify the following keyboard and mouse control prompts:
1. "**Toggle View**" - The key for this action is "**V**".
2. "**Zoom**" - The key for this action is "**O**".
3. "**Flip**" - The key for this action is "**L**".
4. "**Back**" - The key for this action is "**ESC**".

(b) GPT-4V struggles to recognize the icons for keys on keyboard and mouse buttons.



ChatGPT

The central white arrow is facing towards the **left**.

ChatGPT

The central white arrow with the white arrowhead is facing towards the **left**.

(c) GPT-4V cannot understand the correct direction of arrow points, *i.e.,* character, towards in the mini-map.

Figure 12: Example situations of GPT-4V's limitations in understanding visual information from the game.