Pushing the Reasoning Abilities of Large Language Models with 2D Gaming

Étienne Mitchell-Bouchard
Mila
Université de Montréal

Abstract

Recent advances in large language models (LLMs) have raised questions about how close these systems are to human-level reasoning. While benchmarks largely focus on static or symbolic tasks, interactive environments such as video games provide a complementary and demanding testbed for sequential decision-making, perception, and planning. In this project, we evaluate the game-playing abilities of modern LLMs in the context of classic 2D Nintendo Entertainment System games, specifically Super Mario Bros. (1985) and The Legend of Zelda (1986). We propose a general LLM-to-game pipeline based on emulation, multimodal prompting, and controlled input timing inspired by tool-assisted speedruns. The framework parameterizes the visual context provided to the model and strictly constrains its action outputs to controller inputs. We test several models from Google's Gemini suite under varying prompt and screenshot configurations and measure performance using game-specific progress metrics. Our results show that while LLMs perform better than random play, they remain far from human-level competence, struggle with precise spatial control, and exhibit no learning across repeated situations. Additionally, stronger benchmark performance does not reliably translate to better in-game behaviour. These findings highlight current limitations of LLMs in embodied, interactive tasks and point toward future work involving richer temporal inputs, broader model comparisons, and improved evaluation at scale.

1 Introduction and Motivation

Throughout 2025, large language models (LLMs) have been consistently improving in various benchmarks. With Gemini 3 having the best scores among all of the available models on many benchmarks when it came out (Google, 2025), a question stands: how close are these models getting to human-level intelligence? Benchmarks like *Humanity's Last Exam* and *ARC-AGI-2* intend to answer just that question, and Gemini 3 fared much better than the other released models like Claude Sonnet 4.5 and GPT-5.1.

To truly determine how close LLMs are getting to human-level intelligence, they must be evaluated on all the tasks performed by humans. One such task is video game playing, which requires reasoning abilities beyond typical language token generation. As vision-language models (VLMs) are still limited, and to avoid the intricacies of depth perception in 3D gaming, the LLMs will be required to play 2D games. Specifically, older 2D games with fewer pixels per frame offer the perfect evaluation environment, as they allow for reduced inference cost while maintaining the same aspects we want to evaluate. In this context, we chose two games for the Nintendo Entertainment System (NES): Super Mario Bros. (1985) and The Legend of Zelda (1986). Super Mario Bros. (SMB) is a fast-paced platformer that will test the model's physics understanding and ability to reach a goal in a timely manner. The Legend of Zelda (TLOZ) is more about exploration and puzzles, which will test the model's abilities to properly find the next logical step to progress.

SMB has already been used to test LLMs, as it is part of the games in the LMGame-Bench (Hu et al., 2025), a benchmark to evaluate the game-playing abilities of LLMs. In the benchmark's paper, they enumerate the results they obtained with LLMs released in early 2025. Another one of our goals is to see if newer models

can beat the performance of the models in LMGame, as they are older and perform worse than Gemini 3 on many benchmarks (Google, 2025).

To summarize, here are the contributions made by this project:

- A parameterized way to determine the contents of the game images the model should use for inference
- A formalized description of the text content of the prompt the model should receive along with the images
- An implementation of an LLM-to-game pipeline that allows users to easily test models on a variety of games
- An automatized way to query Gemini through the browser instead of using the API

We publish our code to this GitHub repository. We also provide a video (Figure 4) of the project in action, which we highly recommend viewing for comprehension. Please read the whole project report to better understand the code and the video.

2 Problem Formulation

Playing video games can be seen as a sequential decision-making problem, where at a set interval of time, the player has to decide what the next inputs are to make to progress in the game. For humans, we perceive video games as continuous, as the frames per second (FPS) they display are high enough to make it seem like the game runs in continuous time. The two games we will test both run at 60 FPS, which means a frame is produced every $\frac{1}{60}$th of a second, which is equivalent to approximately 16.7 milliseconds. Since the game receives inputs every frame, this also means that humans can technically send new inputs at every frame, although this is often difficult and requires practice due to reaction time and muscular limits in the fingers.

LLMs have a variable inference time, with models using Chain-of-Thought (CoT) often taking several seconds to produce an answer. This makes these models unviable for real-time non-turn-based play, which means we will have to give them an advantage over humans to allow them to play. Taking inspiration from Tool Assisted Speedruns (TAS), the game will be paused every time the LLM has to produce an input and will wait for as long as necessary to receive the input. This means that compared to humans, the LLM will have as much time as it wants to think about the next inputs, making reflexes obsolete. This approach also comes with downsides, which will be described in 3.2.

2.1 Game Environment

The game's environment has two main components: the action space and the state space.
The action space is fairly small, composed of the available inputs on whatever controller the game's console uses. The NES controller has exactly 8 inputs. 4 are on the directional pad (D-Pad): *Up*, *Right*, *Down*, *Left*. 2 are for regular gameplay, *A* and *B*, and 2 are for navigating menus and pausing the game, *Start* and *Select*. For our context, *Start* and *Select* can be ignored, as they are not used in regular gameplay.
The state space is far bigger and far more complicated. It can be represented as the possible values of the game's memory or as the set of all possible frames the game can produce. This state space is, however, not as important as the action space and will only be used to obtain information about the game, such as progress and player control. Furthermore, this state space cannot be fully observed and will be represented as screenshots of the game to the LLM.

2.2 Emulation Framework

To emulate the games, we used the Mesen emulator, which supports multiple game consoles: the NES, the SNES, the Game Boy Color, the Game Boy Advance, and many more. As well as being open-source, Mesen supports Lua scripting, which allows access to the game's memory and allows the emulator to communicate with other programs through sockets. This emulator also possesses a functionality called *Movies*, which allows recorded inputs to be replayed in-game, just as if they were being played for the first time. This allows us to review the actions performed by the LLMs to better understand how to improve their performance. Since Mesen is an accurate emulator, it also guarantees that a given sequence of inputs always produces the same sequence of game states (except for RNG-dependent game values), which is crucial for reproducibility. Thankfully for us, random game states produced by random number generators (RNG) won't be an issue, as game screenshots properly represent these states. As for

input replay, the *Movies* functionality stores the RNG values to ensure identical game states to when the inputs were performed for the first time.

2.3 LLM-Agent Interface

To control the main execution of the playing sequence, a Python program will make the bridge between the LLM and the game. Since Mesen is a standalone program, we use `localhost` to create a socket through which the Python program and Mesen will send and receive information. More precisely, the Python program will host a socket through a certain port, which Mesen will connect to in order to exchange data. Since Mesen is already running the game, the objective was to put as much of the logic as possible in the Python program so that Mesen only has to execute inputs and send information about the game. The execution loop is described in more detail in 3.4.

2.4 Performance Metric

To measure how the LLM performs in the game, we need a way to measure progress in each game. Since objectives vary between games, this progress measurement needs to be game-specific.

For SMB, progress will be measured in terms of how far Mario has travelled to the right compared to the leftmost point in the level. This gives us a number in in-game units, which varies from 0 to around 3000, depending on the level. This is the metric used by LMGame-Bench (Hu et al., 2025), which is not very descriptive, as this value is never shown to players in the game and is only available by doing some calculations on values in the game's memory. To make this progress more understandable to the average player, we first indicate in which world and level the player is in. SMB offers 8 worlds with 4 levels each, represented at the top of the screen in the form "World-Level". To add more precision to this metric, we then divide the current position of Mario in in-game units by the position of the level's objective (flag or axe), which varies by level. This gives us a ratio, which we transform into a percentage rounded to one decimal place for better readability. By combining these two values, we obtain the much clearer performance metric "World-Level (Percentage of Level Completed)". For example, when Mario starts his adventure in the first level, his progress is "1-1 (1.3 %)".

For TLOZ, progress is not as linear. In the game, the objective is to obtain items and defeat bosses, which are located in dungeons. Location does not matter as much as in SMB, and so it will not be considered in this game's progress metric. Instead, the progress metric will be composed of what items Link has obtained and what dungeons he has cleared.

2.5 Models Evaluated

The models tested are from Google's Gemini suite, more precisely:

- Gemini 3 Pro
- Gemini 3 Flash with Thinking
- Gemini 3 Flash
- Gemini 2.5 Flash
- Gemini 2.5 Flash Lite
- Gemini 2.0 Flash

The three variants of Gemini 3 are available for free through Gemini's browser UI with a Google AI Pro subscription, which was free for students for a full year up until December. This subscription promises unlimited image uploads but does not deliver, as we were limited to about 100 image uploads per day. This severely affected our ability to collect data and test the models, as a lot of inference calls are needed to obtain inputs. Furthermore, to automate the process of sending prompts and interpreting model answers, which is usually something done by a human with the browser UI, we used Playwright to automate browser actions such as mouse clicks, file uploads, keyboard inputs, and web scraping. This allowed us to use the browser UI just like if it was a regular API, automating the game-playing process.

In addition to the browser UI, we also used the few free API calls Google offers to nonpaying users. As the API's bills are processed through Google Cloud and the browser version uses the Google One subscription,

our access to Gemini 3 could not be transferred to the API. We used the free API calls to test 2.5 Flash and 2.5 Flash Lite, but with limits of 10 to 20 requests per day, we couldn't collect much data.

3 Methods

One of the goals of the project being to determine what prompts result in the best LLM performance, we have to concretely define the contents of our prompts. Our prompts will be multimodal, as they will contain an image and text. NES games also produce sounds, but these often do not offer additional information about the game and will not be considered for this experiment.

3.1 Model answers

The LLM's answers will be heavily restricted. Since we want the model to act like a video game player with a controller in hand, we need the model to only output a combination of actions. These actions correspond to the relevant buttons available on the NES controller: *Up*, *Right*, *Down*, *Left*, *A* and *B*. The model will be given the instruction to only answer with a combination of these inputs, separated by commas. If it fails to do so, the game will unpause and apply no inputs, moving on to the next frame where inputs are queried. To avoid this behaviour, an instruction will be added to the next prompt, reminding the model that it must only answer with comma-separated inputs.

3.2 Game Screenshot Parameters

To make the model understand what is happening in the game, we want to send a single image describing what recently happened in the game. To formalize what this image must contain, we define three integer parameters, which together will determine the contents of said image. These parameters are defined as follows:

1. **input_length**: Duration in frames for which a combination of inputs is held. This is comparable to the model's reaction time, as no matter what happens during these holding frames, the model must wait until the next input querying frame to react. This parameter can take values between 1, which is frame-perfect playing (the model has immediate reaction time), and the game's FPS, which is equivalent to holding each combination of inputs for exactly one second. Values could be bigger than the game's FPS, but this severely hinders the model's ability to play, as its actions are simply not precise nor fast enough to play correctly. Lower values also require a lot more compute, as more inference calls are needed. For a value of 1 and a game that runs at 60 FPS, 60 inference calls are required for 1 second of real-time gameplay.
2. **n_screenshots**: The number of past frames included in the image. If this parameter's value is bigger than 1, frames will be ordered this way: the leftmost frame is the oldest, and the rightmost frame is the most recent. When multiple frames are present, they are separated by a 1-pixel-thick vertical line (see Figure 1). This parameter can take values from 0 (no screenshot) to a maximum of 10, after which it will be difficult for the model's VLM to correctly place the frames in time.
3. **freq_screenshots**: The amount of frames between each screenshot, equivalent to the frequency at which screenshots are taken. This parameter allows the LLM to see further back in time with fewer screenshots, as each screenshot is more distant in time. The values this parameter can take depend on the values of the other two parameters, with a minimum of 1. For instance, a value of **freq_screenshots** that makes the LLM see frames for which it has already acted makes little sense and is inefficient. If **n_screenshots**$= 1$, a value of more than 1 for **freq_screenshots** is not logical since it means considering an outdated game state.

Together, these parameters aim to recreate a near-human playing experience for the LLM, although we as humans perceive the game in continuous time. The most important parameter is definitely **input_length**, as a higher value truly limits the precision and accuracy at which the LLM can play. It also severely affects the amount of testing that can be done given a certain amount of compute, so finding a well-balanced value between model performance and inference cost is crucial.

3.3 Text Prompting Strategy

To give context to the model on what its task is, the first prompt we send is a text-only description of what the LLM must do. This initial prompt will not vary between game-playing sessions. An example is provided in subsection A.2 of the Appendix. Here are the main points this prompt must get across to the LLM:

- The task of the LLM, which is playing a game
- Information about the game being played, such as its name, console, release date, and goal.
- The restrictions of the LLM's answer, which are the inputs, accompanied by a description of what each input does
- A description of the impact of the answered inputs on the game (**input_length**)
- A description of the information it will receive to choose the inputs

Along with the game screenshots, the model might benefit from additional textual information about the game each time inputs are queried. This textual part of the prompt can contain:

- The model's current game progress
- Encouragement for the model to try different inputs if it is stuck
- A reminder of the correct answer format if the previous answer was not valid inputs

Along with this, a textual description of the contents of the current screenshots could help the model better understand the game's current state, but we did not experiment with this.

3.4 Execution Loop

The Python program and Mesen work turn by turn to create an execution loop, which goes as follows:

1. Mesen lets the game run for **input_length** frames, during which it collects the game screenshots and the current progress.
2. Mesen then sends the screenshots and progress to the Python program through the connected socket.
3. After receiving the data, the Python program builds a prompt to send to the LLM and saves its answer.
4. The inputs, if valid, are then sent to Mesen, which will hold them during the first step, thus looping back to step 1.

Since the Python program controls the main flow of execution, it determines when the execution ends. The simplest ending condition is to play until the game reaches a `Game Over` state, which is equivalent to one playthrough of the game. It could also be possible to go through multiple playthroughs to test different values of our integer/textual parameters in one execution.

4 Experiments

We test each of the available models with various values of our integer parameters, while keeping the textual parameters mostly constant. All our results and experiments are in the `data` folder of the repository.

4.1 Data Collection

We store our collected data in the form of playthroughs, which are a sequence of inputs and progress logged each time new inputs are queried. The data files are in `CSV` format, where each input and progress pair are comma separated. Inputs are separated by semicolons (;), and a pipe (|) separates the inputs and the progress. The name of each `CSV` file is important, as it indicates the values of the integer parameters and the model used. Each new value combination of parameter value results in a new file. Each line of the file represents a playthrough, and a playthrough ends when the progress reaches the `GAME OVER` state. For SMB, the `GAME OVER` state is reached when Mario loses all three of his lives. For TLOZ, the `GAME OVER` state is reached when Link has no hearts left.

4.2 Parameter Values and Models

With the few inference calls we had access to, we attempted to test what we thought would be the best parameter values for optimal model performance. The collected playthroughs are in the `playthroughs` folders of each game in the `data` folder.

Model	Progress (No Harness)	Progress (Harness)
claude-3-5-sonnet-20241022	1-1 (48.7 %)	1-1 (40.1 %)
claude-3-7-sonnet-20250219 (thinking)	1-1 (45.2 %)	1-1 (44.9 %)
gemini-2.5-flash-preview-04-17 (thinking)	1-1 (48.7 %)	1-1 (44.1 %)
gemini-2.5-pro-preview-05-06 (thinking)	1-1 (32.4 %)	1-1 (47.4 %)
llama-4-maverick-17b-128e-instruct-fp8	1-1 (24.9 %)	1-1 (46.5 %)
gpt-4.1-2025-04-14	1-1 (63.0 %)	1-1 (67.3 %)
gpt-4o-2024-11-20	1-1 (32.5 %)	1-1 (64.8 %)
o1-2024-12-17	1-1 (45.4 %)	1-1 (27.0 %)
o3-2025-04-16	1-1 (61.8 %)	1-2 (9.0 %)
o4-mini-2025-04-16	1-1 (42.7 %)	1-1 (45.8 %)

Table 1: Super Mario Bros. average progress results over 3 playthroughs of various models with or without their harness in the LM-Game benchmark (Hu et al., 2025)

Model	Best Progress	input_length	n_screenshots	freq_screenshots
gemini-3-pro-preview	1-1 (35.6 %)	45	3	1
gemini-3-flash-preview (thinking)	1-1 (56.3 %)	30	1	1
gemini-3-flash-preview	1-1 (20.8 %)	30	3	3
gemini-2.5-flash	1-1 (44.8 %)	30	3	1
gemini-2.5-flash-lite	1-1 (38.0 %)	45	3	1
gemini-2.0-flash	1-1 (78.1 %)	45	3	1

Table 2: Best progress of all playthroughs of all tested Gemini models, along with their integer parameters

4.3 Textual Parameters

The contents of the textual parameters were updated and improved when the models expressed obvious difficulties that could easily be rectified with a more descriptive textual prompt. The values of these parameters were not logged in the data, but their contents are described in more detail in 3.3.

4.4 Results

Let's first compare our results for SMB. To put them into context, let's compare them to the results obtained by the models evaluated in LMGame-Bench (Hu et al., 2025), which are displayed in Table 1. We can see that in their study, most models struggle to go beyond 50% completion of the first level. The clear winner is OpenAI's o3, which is the only model that reached level 2 with the help of the harness. The authors also tested random inputs, which gave them a progress of 1-1 (31.2 %). Let's now compare this to our models' best scores, along with the parameters used to obtain this score, which are in Table 2. We can see that none of our models beat the results of *o3-2025-04-16*, LM-Game's best-performing model. We can also observe that there doesn't seem to be a correlation between smarter models (defined by their scores on intelligence benchmarks) and best progress, since the results seem pretty random and *gemini-2.0-flash*, the oldest model in the models we tested, performed the best. There also doesn't seem to be a clear correlation between integer values and performance, as the best progress varies wildly between values.

For TLOZ, none of the models that we tested managed to succeed in reaching any form of progress, which corresponded to obtaining an item or beating a dungeon. They were tested with integer parameters **input_length** $= 30$, **n_screenshots** $= 1$ and **freq_screenshots** $= 1$. Indeed, the first objective of the game is to obtain the sword, located in a small cave with the entrance being in the first screen of the game (see Figure 2). The models managed to reach the cave but never managed to grab the sword, as they couldn't pinpoint the exact movements they needed to make for Link to reach the sword (see Figure 3). The models tested were the three

variants of Gemini 3 present in Table 2. Furthermore, no models manage to reach a `GAME OVER`, and so no playthroughs were logged in the `data` folder.

4.5 Failure Cases

Model decisions for certain images were almost consistent, with models never learning to change their inputs if the previous inputs they provided in response to an image they had already seen led to a bad outcome. This meant that once the model had failed, it was highly likely that it would fail in the same way if it encountered the same situation again. For example, in SMB, Mario is reset to his starting position after losing a life. To lose a life, models produced a series of inputs that led to Mario dying is some way, either to an enemy or by falling in the void. After resetting, the LLMs almost always repeated the same exact input sequence that led to their first death, indicating that there was no learning whatsoever happening.

The LLMs also did not seem to correctly understand where objects were placed in the games. They often recognized that an object was on screen but struggled to reach or avoid it. This was especially obvious in TLOZ, where models knew the general position of the cave and sword, but they struggled to reach them, even failing entirely for the sword.

5 Conclusion

In conclusion, too few experiments were made to discover meaningful relationships between the parameters we defined and model performance. Better model performance in other benchmarks also does not seem to correlate with better progress in the video games we tested, but further testing is required to validate this hypothesis. We were also unable to beat the results obtained in the LM-Game benchmark, probably because their harness mechanism allowed for better performance, and they had more resources at their disposal for testing.

The main goal of this project was to determine if LLMs are capable of playing games. In the case of the older 2D games we tested, we can conclude that they have better than random game-playing abilities, but they are still far from human-level. More research would be needed to get a more definitive and precise conclusion, but it is clear that humans are far superior to LLMs in the task of playing games.

5.1 Future Work

With this project providing everything needed to test the LLMs on their game-playing abilities, the next logical step would be to test the LLMs further by having access to more compute and inference calls. With Mesen providing access to a huge library of games, testing the LLMs on other games would solidify the findings and would provide a more concrete answer as to if they can play games or not. Since we only tested Gemini models from Google, evaluating LLMs from other providers such as OpenAI, Anthropic, Meta and some open-source models like Qwen or Deepseek seems necessary. One suite of models is not enough to generalize to all LLMs, so testing models from other providers would be required to verify the validity of our claims.

For the LLMs to get closer to human-level playing abilities, it seems natural that they would need to perceive the games like humans do. To do so, they would need to receive video prompts instead of images and text, as videos contain the sounds and images in temporal order, just like how humans perceive the game when playing in real time. Only with this mode of prompting could we really compare humans and LLMs, as they would then be receiving the same information about the environment. LLM inference time would also need to be reduced greatly to allow the models to display the reflexes we see in human playing.

References

Google. Gemini 3: Introducing the latest Gemini AI model, Nov 2025. URL https://blog.google/products/gemini/gemini-3/.

Lanxiang Hu, Mingjia Huo, Yuxuan Zhang, Haoyang Yu, Eric P. Xing, Ion Stoica, Tajana Rosing, Haojian Jin, and Hao Zhang. LMGame-Bench: How good are Large Language Models at playing games?, 2025. URL https://arxiv.org/abs/2505.15146.

A Prompt examples

A.1 Image Prompts

Figure 1: Screenshot image example of Super Mario Bros., with integer parameters **n_screenshots** $= 3$ and **freq_screenshots** $= 10$

Figure 2: Screenshot image example of The Legend of Zelda, with integer parameters **n_screenshots** $= 1$ and **freq_screenshots** $= 1$. The cave entrance is represented by the black square in the green hill, to the left of Link.

Figure 3: Screenshot image example of The Legend of Zelda, where we can see the sword in the cave that represents the first objective of the game. Note that Link is not correctly aligned horizontally with the sword to reach it by just pressing *Up*.

A.2 Text Prompts

Example of an initial context prompt for Super Mario Bros. :

> You are a video game player. You are currently playing the game Super Mario Bros. (1985) for the NES. The goal of the game is: You must reach the flag pole at the end of each regular level and the axe at the end of each castle level, which are both always to the right. Jumping over pipes, enemies, obstacles and gaps is crucial to progress.
> Your answers will be limited. The only thing you can answer is a combination of these inputs, separated by commas (,):
> left : Moves Mario left.
> right : Moves Mario right.
> down : Makes Mario enter vertical pipes if they are enterable. Also allows big Mario to crouch.
> a : Makes Mario jump. Holding it longer makes Mario jump higher. The button needs to be released to jump again.
> b : Holding it makes Mario run faster when 'left' or 'right' is also held. Tapping it allows fiery Mario to shoot fireballs.
> The inputs you answer will be applied for 45 frames, which is equivalent to 0.75 seconds, after which you will receive a new image to repeat the process. The inputs you answer must respect the format, and they must contribute to reaching the game's goal. Only one of each input must be in the answer. If you see that the inputs have no effects on the game, try different ones; don't try the same inputs more than 3 times if you don't see any changes.
> To decide which inputs to choose, you will be given images of the last 3 frames that the game has rendered, where the leftmost frame is the oldest and the rightmost frame is the most recent. With these images, you will also receive your current game progress.
> Answer "Understood." if you understood these instructions.

B Video example

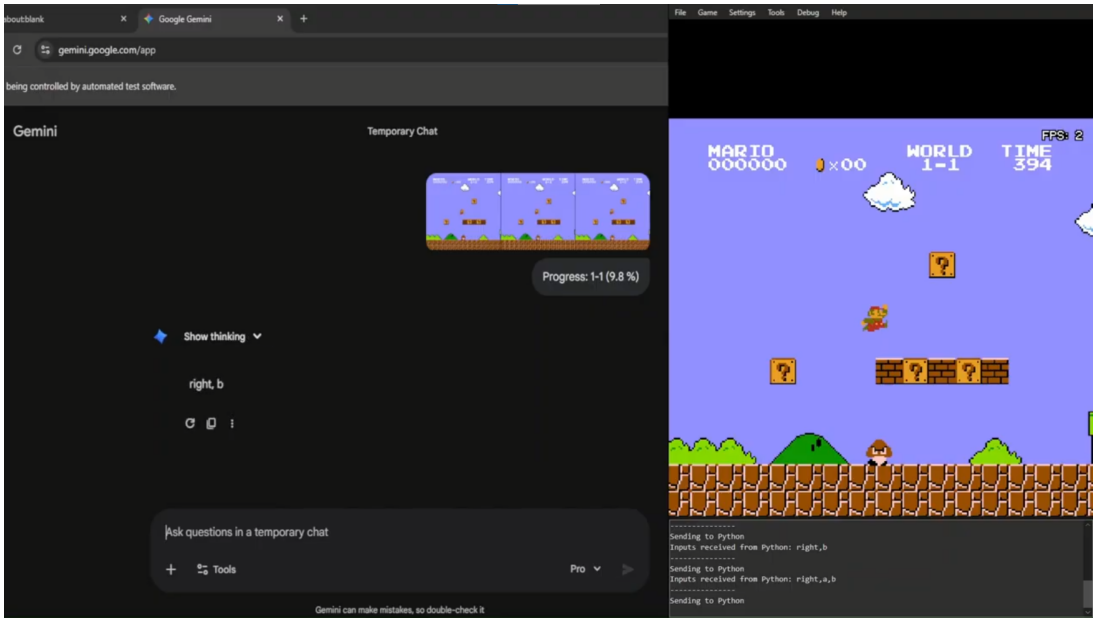

Figure 4: Super Mario Bros. gameplay with the model Gemini 3 Pro in the browser. The integer parameters are **input_length** $= 30$, **n_screenshots** $= 3$ and **freq_screenshots** $= 1$. For brevity, the gameplay episode ends when Mario loses a life, which is not a full playthrough.
Click the image to view the full video.