BYTESIZED32: A Corpus and Challenge Task for Generating Task-Specific World Models Expressed as Text Games

Ruoyao Wang[†], **Graham Todd**[‡], **Eric Yuan**[◇], **Ziang Xiao**[◇♠], **Marc-Alexandre Côté**[◇], **Peter Jansen**[†]
[†]University of Arizona [◇]Microsoft Research Montréal
[‡]New York University [♠]Johns Hopkins University
{ruoyaowang,pajansen}@arizona.edu
gdrtodd@nyu.edu {eric.yuan,ziangxiao,macote}@microsoft.com

Abstract

In this work we examine the ability of language models to generate explicit world models of scientific and common-sense reasoning tasks by framing this as a problem of generating text-based games. To support this, we introduce BYTESIZED32, a corpus of 32 highly-templated text games written in PYTHON totaling 24K lines of code, each centered around a particular task, and paired with a set of 16 unseen text game specifications for evaluation. We propose a suite of automatic and manual metrics for assessing simulation validity, compliance with task specifications, playability, winnability, and alignment with the physical world. In a single-shot evaluation of GPT-4 on this simulation-as-code-generation task, we find it capable of producing runnable games in 27% of cases, highlighting the difficulty of this challenge task. We discuss areas of future improvement, including GPT-4's apparent capacity to perform well at simulating near canonical task solutions, with performance dropping off as simulations include distractors or deviate from canonical solutions in the action space. [1]

1 Introduction

In a relatively short span of time, the capabilities of large language models (LLMs) have exploded. While trained on natural language to predict subsequent tokens in context, LLMs have nonetheless been deployed in a variety of additional contexts – as conversational partners (Bang et al., 2023; Bahrini et al., 2023), code assistants (Bird et al., 2023; Xia and Zhang, 2023), and even artificial agents (Huang et al., 2022; Ahn et al., 2022). In this work, we investigate the ability for large language models to operate on a novel task: producing small but explicit world models, in the form of simulations of performing various scientific and common-sense reasoning tasks.

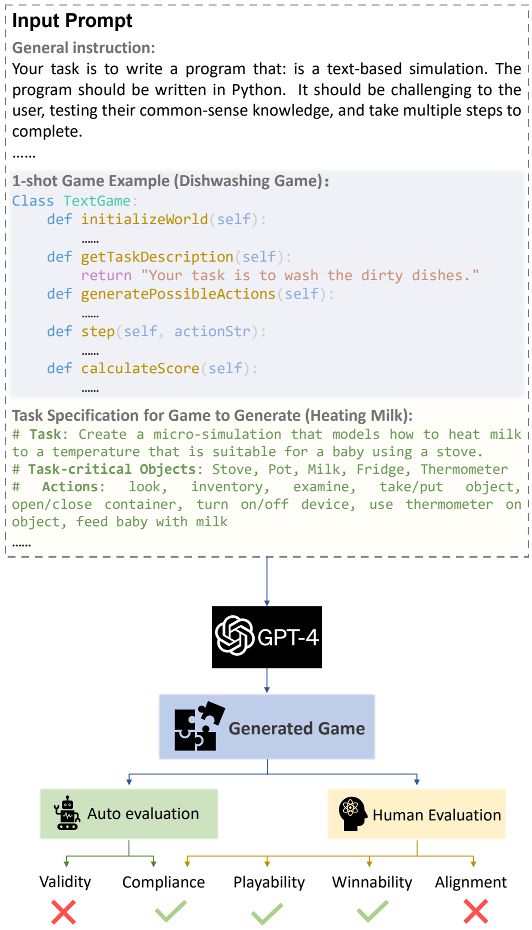

Figure 1: An overview of our text game generation and evaluation process. The model, here GPT-4, generates a game by using a prompt consisting of (1) a 1-shot game example, and (2) the task specification for the game to generate. The generated game is then evaluated by an automated evaluation harness, as well as manually by human evaluators, to measure its validity, compliance, playability, winnability, and physical world alignment.

We frame the problem of simulation as generating text-based games, in which agents receive environmental observations and perform actions entirely through natural language. Text games can encode aspects of reality without the need for graphical rendering pipelines, and are a challenging domain for artificial agents (Hausknecht et al.,

[1]BYTESIZED32 available at: https://github.com/cognitiveailab/BYTESIZED32

2020), as well as a fruitful method for interrogating agents' understanding of scientific reasoning concepts (Wang et al., 2022). Text-based games thus offer the ability to convert a model's latent general knowledge absorbed during the training process into actionable and falsifiable hypotheses about the world. By formulating this as a code generation task, where models must generate hundreds of lines of PYTHON code to create a simulation, we provide a vehicle for examining a language model's internal world model.

We introduce BYTESIZED32, a dataset of text-based games written in PYTHON, each testing a different common-sense or scientific reasoning concept, in order to facilitate the development of automatic simulation-generation systems through few-shot learning. The dataset uses a consistent code architecture and provides generative models with a robust template that can be adapted to a wide range of simulations. In addition, we propose a corresponding challenge task in using the BYTESIZED32 corpus to generate novel text game simulations using a corpus of unseen text game specifications used for evaluation. We perform a preliminary analysis of the performance of GPT-4 on this task, shown in Figure 1, using a range of proposed automatic and manual metrics that measure the model's ability to generate text-based games that: (1) are syntactically correct, (2) contain core aspects of the simulation enumerated in a specification prompt, (3) are playable and winnable, and (4) align with known facts about physical reality. Our results indicate that GPT-4 is broadly capable of generating novel text-based games that match the BYTESIZED32 template and contain specified objects and actions. However, it often fails to produce simulations that are robust to all possible user actions and that fully align with reality.

The contributions of this work are:

1. We introduce BYTESIZED32, a corpus and challenge task for generating world model simulations framed as long-form code generation of text games.

2. We provide a corpus of 32 highly-templated text games centered around specific scientific and common sense reasoning tasks. The corpus totals 24K lines of PYTHON code, with individual games designed to fit within the 8k context window of popular and capable LLMs.

3. We empirically demonstrate that, in an *single-shot* setting, a baseline GPT-4 model is capable of producing running simulations 27% of the time, but simulations are physically correct in only 2% of cases, highlighting the difficulty of this task.

4. We further propose automatic evaluation metrics that break down performance into finer-grained components, such as the ability to initialize the world, produce valid actions, or generate scores that accurately model task performance. We pair these with human evaluation metrics that measure whether generated games are playable, winnable, and accurately models the physical world.

2 Related Work

Text Games and Virtual Environments: Interactive text environments are an attractive choice for studying embodied agents, owing to their relative simplicity compared to full 3D simulations and ability to model complex and abstract tasks (Jansen, 2021). While early text game research focused on testing agents on a small set of extant "interactive fiction" games like *Zork*, recent approaches have leaned towards procedurally generating a wider set of simple text-based games in order to evaluate agents' ability to generalize (Côté et al., 2018; Urbanek et al., 2019; Shridhar et al., 2020; Wang et al., 2022). These frameworks typically rely on hand-crafted rules and templates programmatically arranged in novel configurations, though some efforts leverage external data sources (Barros et al., 2016) and generative language models (Fan et al., 2019) as well. In contrast, the BYTESIZED32 challenge task requires models to produce a novel text game as a complete program, expressed as PYTHON code, using only a single existing game for reference.

Code Generation: As large language models have become more capable, interest in their ability to generate working snippets of program code has only grown. Several recent datasets have been proposed to facilitate this research, covering a wide range of programming languages and problem types (Yu et al., 2018; Lin et al., 2018; Austin et al., 2021; Chen et al., 2021). Contemporaneously, improvements in model architecture and training have led to impressive gains in code generation (Chen et al., 2021; Nijkamp et al., 2022; Li et al., 2022; Fried et al., 2023). The GPT-4 language model

(OpenAI, 2023), in particular, has sparked an interest in the use of prompting for code generation tasks, a technique which has led to advancements in self-debugging (Chen et al., 2023) and problem decomposition (Pourreza and Rafiei, 2023). Despite these gains, however, existing code generation benchmarks tend to require short and relatively simple programs. In contrast, here models must generate many hundreds of lines of PYTHON code to generate complete and accurate task simulations, with training games ranging from 500 to 1000 lines of code.

3 Simulation as Code Generation

In this work, we investigate generating simulations – in the form of PYTHON programs – that provide minimal world models to sufficient detail that they allow accomplishing specific multi-step tasks drawn from the scientific and common-sense reasoning domains, such as *boiling water* or *washing dishes*. We operationalize this as a task of generating task-specific text games, which are virtual game environments that are rendered (and interacted upon) exclusively through text. Specifically, we pose as a challenge task the notion of generating text games when provided with two components as input: (1) a detailed *task specification* for the game to generate, and (2) one or more highly-templated text games as examples.

In Section 4, we describe a manually-constructed corpus of highly-templated text games crafted to facilitate *n-shot* generation of new text-game-based simulations. In Section 5, we describe the challenges in evaluating long (several hundred line) programs for functionlity, task faithfulness, and simulation fidelity. In Section 6, we evaluate the performance of a baseline GPT-4 model at generating novel text games. Finally, we discuss future challenges and opportunities in Section 8.

Python Text Game Template

```
# Generic parent class for all game objects
# Provides getters/setters for object properties
class GameObject(): ...

# Parent class for game objects that are containers
# Provides methods for adding/removing objects from container
class Container(GameObject): ...

# Parent class for game objects that are devices
# Provides methods for activating/deactivating a device
class Device(GameObject): ...

# Example object: Soap for washing dishes
class DishSoap(GameObject): ...

# Example object: A dish (that can contain food)
class Dish(Container): ...

# Example object: A dishwasher (that can contain dishes,
# dish soap, and be activated to wash the dishes)
class Dishwasher(Device, Container):

# Main Simulation Class
class TextGame():
    # Creates the game world and populates with game objects
    # (including the kitchen, dishes, dishwasher, foods, etc.)
    def initializeWorld(): ...

    # Returns a string describing the game and task
    def getTaskDescription(): ...

    # Returns an array with all possible valid actions given
    # the current game state
    def generateValidActions(): ...

    # Performs an action (e.g. turn on dishwasher) in the environment,
    # changing the environment state.
    def step(action:str): ...

    # Calculate the current game score given progress, as well as
    # whether the game has been won.
    def calculateScore(): ...

# Main Entry Point (example of a user playing)
if __name__ == "__main__":
    game = TextGame()
    print("Task: " + game.getTaskDescription())
    while not game.gameOver:
        actionStr = input("> ")
        observation, score, reward = game.step(actionStr)
        print("Observation: " + observation)
        print("Score: " + score)
        print("Reward: " + reward)
    print("Game Completed.)
    print("Game Won: " + str(game.gameWon))
```

Table 1: An illustration of the core classes and member functions present in the highly-templated games of the BYTESIZED32 corpus. Note that as each game consists of between 500 and 1000 lines of code, the example here provides an overview of only a subset of the most important functions examined during scoring.

4 Text Games Corpus

To support the task of generating simulations in the form of text games, we construct a corpus of highly-templated text games written in PYTHON that can serve as examples in a *few-shot* generation setting. Each game is between 500 and 1000 lines code, and covers specific tasks within the domain of scientific and common-sense inference, such as *boiling water*, *loading a dishwasher*, or *making a campfire*. Each game has two components: the game code itself, as well as a detailed *task specification* in the form of structured comments at the top of each game that provide a detailed summary of the game task, critical objects, and solution paths. These components are described below, with an example game playthrough shown in Table 2.

4.1 Task Specification

The *task specification* is a set of structured comments at the start of each game in the corpus, that serve as a high-level outline for the critical components of each game. These are intended to provide a useful high-level scaffold that language models can use to better structure games they generate. The components of the task specification include:

- **Task Description:** The task an agent playing the game has to solve – for example, *washing dirty dishes using a dishwasher*.
- **Task-Critical Objects:** Names of any task-critical objects, such as *dishes*, *dish soap*, and a *dishwasher*.
- **Actions:** Actions that an agent playing the game can take, such as *opening* or *closing* containers, *activating* or *deactivating* devices, *picking up* or *putting down* objects, and so forth.
- **Distractors:** Objects (or actions) that limit or hinder task performance – for example, adding *food* that an agent can eat, that creates more dirty dishes.
- **Solution:** A high-level solution to the game. For example: opening the dishwasher, moving each dirty dish from the kitchen into the dishwasher, moving dish soap into the dishwasher, closing the dishwasher, and activating the dishwasher.

4.2 Game Code

To maximize utility as *n-shot* training data for code generation tasks, each game in the corpus uses a highly-templated structure consisting of core objects and member functions, shown in Table 1 and described below. The core architecture and API of these functions mirrors other frameworks for text game research (Hausknecht et al., 2020), which are derived from the OPENAI GYM specification for reinforcement learning models (Brockman et al., 2016).

- **World Initialization:** Initialize the game world, including any objects. For example, for the dishwasher game, create a *kitchen* room that includes *dirty dishes*, *dish soap*, a *dishwasher*, and any other relevant objects.
- **Valid Actions:** Return a list of all possible valid actions that an agent could take, given the current state of the environment. For example, *take dirty dish*, or *open dishwasher*.
- **Take Action Step:** Take a step in the environment by choosing a particular action to take. This function returns the result of that action – for example, the *take dirty dish* action might return the observation *"the dirty dish is now in your inventory"*.
- **Scoring:** Return an agent's current progress in solving the game task, abstracted to an arbitrary numerical score, and a set of boolean flags that represent whether the game has been won or lost.

While the above methods are provided through a main game class (`TextGame`), each game also includes a large number of classes representing specific **game objects**. Each game object derives from a common class, `GameObject`, from which generic subclasses that share common methods inherit (e.g. `Containers`, that can store objects, or `Devices`, that can be activated), before finally instantiating specific game objects (e.g. `Dish`, `Dishwasher`).

5 Evaluating Generated Simulations

Evaluating model-generated simulations (in the form of hundreds of lines of PYTHON code) presents a number of challenges, in part due to the largely open-ended possibilities when generating large simulations constrained only by short task prompts. Here, we devise a set of automatic and manual metrics that measure both technical aspects (such as whether the simulation runs error-free), as well as how well generated simulations adhere to the task specification.

5.1 Automatic Metrics

We propose automated metrics designed to measure two aspects of a generated simulation: *technical validity*, and *specification compliance*.

Technical Validity: Generated games are required to adhere to the API provided Table 1, and the *technical validity* metric evaluates this by testing whether each member function of the generated game's API can run without error. First, each API function is tested in turn, with errors recorded. Finally, the *environment step* function is tested by conducting a breadth-first search over the game's action space, as returned by its own GENERATEPOSSIBLEACTIONS() function, to exhaustively test all game trajectories for errors up to

Boil Water Game
Task Description: Your task is to boil water. **Observation**: You find yourself in a kitchen. In the kitchen, you see: yourself a stove that is currently off and has nothing on it. a sink that is empty a pot that is empty a peanut butter a orange Type 'help' for a list of possible actions. > *take pot* **Observation**: The pot is removed from the kitchen. You put the pot in your inventory. > *put pot in sink* **Observation**: The pot is removed from the agent. The pot is placed in the sink. > *examine sink* **Observation**: a sink that contains the following items: a pot that is empty > *turn on sink* **Observation**: The sink is now turned on. > *examine sink* **Observation**: a sink that contains the following items: a pot that looks to have some water in it > *turn off sink* **Observation**: The sink is now turned off. > *take pot* **Observation**: The pot is removed from the sink. You put the pot in your inventory. > *put pot on stove* **Observation**: The pot is removed from the agent. The pot is placed in the stove. > *turn on stove* **Observation**: The stove is now turned on. > *examine stove* **Observation**: a stove that is currently on and has the following items on it: a pot that looks to have some water in it > *examine stove* **Observation**: a stove that is currently on and has the following items on it: a pot that looks to have some water in it > *examine stove* **Observation**: a stove that is currently on and has the following items on it: a pot that looks to have some water in it > *examine stove* *Game completed.*

Table 2: An example playthrough of the *water boiling* game from the BYTESIZED32 training set.

some maximum number of steps n. An error on any step is considered a failure of the STEP() function. Because a game can have as many as 2000 possible valid actions at each step, we pragmatically limit n to only 3 steps in order to ensure tractability.

Specification Compliance (automatic): This metric reflects whether the generated game meets the requirements outlined in the task specification. In particular, we measure whether a generated game includes the required action, object, and distractor requested in the task specification. In order to facilitate this evaluation at scale, we leverage the question-answering capacity of GPT-4. The generated game and its accompanying prompt are fed into GPT-4, which is then asked a series of true-or-false questions about the presence of the required components – for example, *"Does the simulation contain the object 'Sink'?"*

To validate this automatic GPT-4-based specification compliance metric we also perform a corresponding manual evaluation. Two human-raters independently evaluated the compliance of the generated game on each of the three metrics (i.e. action, object, and distractor compliance). Human raters (co-authors of this work) had a high initial inter-annotator agreement (Cohen's $\kappa = 0.92$), and any disagreements were subsequently resolved. These consensus-based manual ratings were then compared against GPT-4's automatic evaluation. The average agreement is substantial (Avg. $\kappa = 0.79$; Object: $\kappa = 1.00$; Action: $\kappa = 0.62$; Distractor: $\kappa = 0.64$), which suggests the automatic GPT-4 metric has a high correlation with human judgements.

5.2 Manual Metrics

We propose three additional binary metrics to further evaluate if generated games are *functionally* and *conceptually* correct, which are currently difficult to measure automatically. In light of this, we measure these quantities by a manual human evaluation, both in examining generated game code, and playing generated game.

Playability: We consider a game to be "playable" if there exists at least one executable action for the first 20 steps or until the winning state and at least one executable action that contributes positively to the game's winning objective. By executable, we mean actions can be performed without any error and change the game state. The human evaluator determines the set of possible "contributing actions" and then manually checks to see if any of them can be performed. For instance, in a game about *boiling water*, actions that contribute to the winning state might include *removing a pot from a cabinet* or *turning on the stove*.

Winnability: A game is considered "winnable" if there exists a sequence of actions that, when performed in order, will lead to a winning state of the game. To determine if a game is winnable, a human evaluator plays the game and attempts to complete the task by submitting natural language actions to the game's `step` function. We note that this process does not produce a perfect evaluation of a game's winnability, as it may be the case that a human evaluator fails to find a winning trajectory. While this is a risk, we find that pragmatically in the vast majority of cases a game is either obviously winnable or obviously impossible to win.

Physical Reality Alignment: This subjective metric measures if a generated game accurately models the physical world, by having a human manually play the game while actively attempting to break the simulation – for example, by actively trying to move unmovable objects, and observing whether task-critical causal relations are implemented (e.g. does water on a stove increase temperature).

6 Experiments

Here we examine how well a baseline GPT-4 model performs at the simulation generation task.

6.1 Model and Prompt

We use GPT-4 (OpenAI, 2023) with a context window of 32K tokens for each of our experiments. We perform single-shot experiments, where the model receives the following as part of its input prompt:

1. *Purpose:* A general statement about the purpose and requirements of the text game generation task
2. *1-shot example:* A single PYTHON reference game from the BYTESIZED32 corpus, and its accompanying high-level task specification.
3. *Task request:* The *task specification* for the game to generate, drawn from the evaluation set.

For each experiment we use greedy decoding (setting the generation temperature to zero), and leave all other hyperparameters at their default values.

6.2 Evaluation Set

In addition to the 32 games in the BYTESIZED32 dataset, we also provide a test set of 16 additional games in the form of *task specifications*. These specifications are intended to evaluate models' generative capabilities for unseen games, and each task in the test set explicitly requires at least one distractor to be included. Each game in the evaluation set is explicitly crafted to have highly similar or highly dissimilar characteristics to games found in the training set, to evaluate the effect of game template similarity on generation quality. This alignment between training and evaluation games is described in detail below.

6.3 Reference Game Selection

We hypothesize that the quality of a generated game will depend on the alignment between the task specification and the game provided as reference – for example, we might expect apriori that reference games that are highly similar to target games might have better generation performance than reference and target games that are highly dissimilar. To test this hypothesis, we pair each target game specification in the evaluation set with distinct reference games that either resemble or differ from the evaluation specification in the following ways:

1. **Objects**: Two games align with respect to objects if they both contain at least one object of the same category – for example, if the reference game contains a *device* (such as a *dishwasher*), and the target game also requests generating a *device* (such as a *sink*).
2. **Actions**: Two games align in terms of actions if they share at least one action that is required to successfully complete the game, such as *opening* a container or *activating* a device.
3. **Distractors**: Two games align in terms of distractors if they either both require a distractor, or both do not require a distractor. We note that all evaluation games in the test set require generating a distractor.

At test time, for each game specification in the test set, we randomly select six reference games from the training corpus: three training games that align with objects, actions, and distractors, respectively, for the *similar reference game* conditions, and three training games that do not align on each of these criteria for the *dissimilar reference game* condition. With 16 game specifications in the test set, this results in creating a total of 96 model-generated games.

API Method	GPT-4
Game Initialization	90.6
Task Description Generation	83.6
Score Calculation	83.6
Possible Action Generation	77.3
Step	27.3
All Checks Passed	27.3

Table 3: Fine-grained *technical validity* evaluation results on the 96 games generated by GPT-4. Values represent the proportion of games that pass a given metric.

Measurement (Auto)	**In template**	**Not in template**
Task-critical objects	100	100
Task-critical actions	81.8	100
Distractors	31.8	12.5
Measurement (Human)	**In template**	**Not in template**
Task-critical objects	100	100
Task-critical actions	68.8	93.8
Distractors	43.8	31.3

Table 4: Automatic (GPT-4) and manual (human) evaluated specification compliance results. Values represent the percentage of games that include a certain feature. *In template* refers to given feature being present within the reference game provided in the model prompt.

7 Results

7.1 Technical Validity

Results of the *technical validity* evaluation across each of the 96 model-generated games are shown in Table 3. Among the 96 generated games, the vast majority implement the `initializeWorld()` and `generateTaskDescription()` functions without error (90.6% and 83.6%, respectively), with a similar proportion generating `calculateScore()` without error. More than three-quarters of games successfully implement `generatePossibleActions()` (77.3%). Unsurprisingly, the `step()` function, which requires parsing input actions then updating game states and observations accordingly, proves the most difficult to implement – with only 27.3% of games (26 of 96) passing the exhaustive 3-step trajectory search without error.

7.2 Specification Compliance

Automated specification compliance results are shown in Table 4, broken down by whether the reference game has similar (*“in template”*) or dissimilar (*“not in template”*) characteristics. We observe that the GPT-4 model demonstrates the ability to consistently generate task-critical objects regardless of whether a similar object is present in the reference game. In terms of task-critical actions, we find an unexpected result: 100% of generated games contain the required task-critical actions when the action is *not* present in the reference game, while only 81.3% of generated games contain the action when it *is* present in the reference – though we find that in the cases where the model fails to generate a required action, it tends to generate a semantically-equivalent action with a different name. For instance, instead of generating the required action `putOn`, the model generates the action `wear`. Regardless, the overall success rates are high in both conditions, indicating that GPT-4 is able to follow instructions to generate new objects and actions, even when they are not present in its reference prompt.

Generating distractors proves to be more challenging for GPT-4. We observe that, even when the reference game contains a distractor, only 31.3% of generated games meet the requirement of incorporating a distractor. This number falls to 12.5% in cases where the reference game does not include a distractor. The implications of this result are two-fold: that distractor objects and actions represent a more difficult concept for the language model to encode than task-critical objects and actions, and that the presence of distractors in the reference game goes some of the way towards mitigating this difficulty.

We present the manual evaluation results for specification compliance in Table 4. We note that the manual evaluation differs from the automatic evaluation in only 8 of the 96 games, indicating that automatic evaluation is a viable substitute for costly human evaluation of these metrics. We also note that the automatic evaluation most frequently differed from the manual evaluation in the distractors section, which is also the section that proved the most difficult in terms of generation.

Finally, it is also worth noting that these compliance metrics indicate only whether the specified the object, action, or distractor is present in the game, without assessing the correctness of their implementation.

7.3 Playability, Winnability, and Physical Plausibility

Table 5 shows the results of the manual evaluation of each game’s playability, winnability, and

Measurement	GPT-4
Playability	72.9
Winnability	40.6
Physical Reality Alignment	2.1

Table 5: Results of the 96 evaluation games on fully-human evaluation metrics. Values represent the percentage of games that are playable, winnable, or that correctly model the physical world.

physical reality alignment. We note that while only 27.3% of generated games pass the exhaustive trajectory search, a full 72.9% of games are playable (i.e. successfully implement at least one action that contributes towards the task objective) and 40.6% are winnable. This suggests that a substantial number of generated games are valid within the narrow scope of a gold solution trajectory, but begin to exhibit errors at the edges of the action space as they drift from the canonical solution trajectory. In addition, only 2.1% (2 out of 96) of games manage to produce a simulation that is fully consistent with the limited subset of physical reality they model. This fundamental component of simulation through code generation thus appears to be beyond the capacity of current models, providing the basis for a strong challenge task in physical simulation through code generation.

8 Discussion

To what extent can GPT-4 generate long structured text games under in a single-shot setting? At a high level, our results indicate that GPT-4 is capable of generating syntactically valid and templated programs that are hundreds of lines in length. Of the generated games, 72.9% implement at least one task-critical action without error, and a full 40.6% allow a user or agent to reach a winning state. A more nuanced interpretation of these results suggests that the model has best learned to successfully replicate the *high-level structure* of the highly-templated BYTESIZED32 game API – as model performance begins to degrade once we examine the minute details: only 27.3% of games include a simulation that is robust to a 3-step exhaustive trajectory search, and only 31.5% of games include a required distractor despite their presence in the reference games.

Can GPT-4 create items and actions that were not observed in the reference game? GPT-4 demonstrates a capability to generate games that incorporate unseen objects, as required by the task specifications in all of our test cases. Similarly, in an impressive 93.8% of cases, GPT-4 generates task-critical actions that are not observed in the reference game. However, it is important to note that in numerous instances, these generated objects and actions do not accurately reflect the physical world. Only a mere 2.1% of generated games successfully complete the text game generation task without any errors in terms of modeling the physical world.

Can we observe the internal world models of LLMs through the simulation they generated? The “simulation as code generation” paradigm potentially offers a valuable means to explicitly assess how large language models understand the world. For example, for an evaluation specification where the objective is to generate a game involving burying a treasure box in a hole, the generated game mandated the placement of soil in the hole before burying the treasure box – highlighting a deficiency in GPT-4’s comprehension of the process of *burying*. Similarly, in another case, the agent was able to directly access water without utilizing any containers, indicating a lack of grasp of the concept of liquids by the model – at least in certain contexts. Despite the extensive knowledge encompassed within its pretraining data, only 2.1% of the GPT-4 generated games accurately model the physical world. Consequently, constructing correct explicit world models in code remains a formidable challenge for LLMs.

9 Conclusion

In this paper, we examine the task of generating simulations of scientific and common-sense reasoning tasks, formulated as a code generation problem of generating text games that explicitly test these tasks in PYTHON. We present a text game corpus for code generation, BYTESIZED32, containing 32 training games paried with detailed structured task specifications, as well as a separate corpus of 16 unseen task specifications for evaluation. Notably, all the games in the training set are implemented following a unified code template, while the test game specifications are carefully designed to evaluate the benefit of the presence or absense of similar features in training games, such as shared objects, actions, or distractors. This facilitates investigating the influence of reference games present in *n-shot* prompts on text generation, that we hope will enable a comprehensive analysis of this topic.

We introduce a suite of evaluation metrics for measuring the quality of generated games. These include fully-automatic metrics that assess generation *validity* and *specification compliance*, while fully-manual human evaluations are used to measure game *playability*, *winnability*, and *physical reality alignment* to assess the functional and conceptual correctness of the generated games.

Our experimental results show that while only 27.3% of the generated games pass an exhaustive *validity* test without encountering runtime errors, 72.9% of the games are playable and 40.6% of the games are winnable – highlighting that GPT-4 makes impressive progress in generating games along a solution path, but tends to make errors when deviating from canonical solutions or that are unrelated to the task. These findings highlight the considerable progress that still needs to be made for LLMs in order to effectively demonstrate their understanding of the world by constructing comprehensive virtual simulation environments.

Acknowledgements

This work supported in part by National Science Foundation (NSF) award #1815948 to PJ, and the Allen Institute of Artificial Intelligence (AI2).

References

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, Sally Jesmonth, Nikhil J Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Kuang-Huei Lee, Sergey Levine, Yao Lu, Linda Luu, Carolina Parada, Peter Pastor, Jornell Quiambao, Kanishka Rao, Jarek Rettinghouse, Diego Reyes, Pierre Sermanet, Nicolas Sievers, Clayton Tan, Alexander Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, Mengyuan Yan, and Andy Zeng. 2022. Do as i can, not as i say: Grounding language in robotic affordances.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Aram Bahrini, Mohammadsadra Khamoshifar, Hossein Abbasimehr, Robert J. Riggs, Maryam Esmaeili, Rastin Mastali Majdabadkohne, and Morteza Pasehvar. 2023. Chatgpt: Applications, opportunities, and threats.

Yejin Bang, Samuel Cahyawijaya, Nayeon Lee, Wenliang Dai, Dan Su, Bryan Wilie, Holy Lovenia, Ziwei Ji, Tiezheng Yu, Willy Chung, Quyet V. Do, Yan Xu, and Pascale Fung. 2023. A multitask, multilingual, multimodal evaluation of chatgpt on reasoning, hallucination, and interactivity.

Gabriella A.B. Barros, Antonios Liapis, and Julian Togelius. 2016. Murder mystery generation from open data. In *Proceedings of the 7th International Conference on Computational Creativity, ICCC 2016*, Proceedings of the 7th International Conference on Computational Creativity, ICCC 2016, pages 197–204. Sony CSL.

Christian Bird, Denae Ford, Thomas Zimmermann, Nicole Forsgren, Eirini Kalliamvakou, Travis Lowdermilk, and Idan Gazit. 2023. Taking flight with copilot: Early insights and opportunities of ai-powered pair-programming tools. *Queue*, 20(6):35–57.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. Openai gym. *arXiv preprint arXiv:1606.01540*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

Marc-Alexandre Côté, Ákos Kádár, Xingdi Yuan, Ben Kybartas, Tavian Barnes, Emery Fine, James Moore, Ruo Yu Tao, Matthew Hausknecht, Layla El Asri, Mahmoud Adada, Wendy Tay, and Adam Trischler. 2018. Textworld: A learning environment for text-based games. *CoRR*, abs/1806.11532.

Angela Fan, Jack Urbanek, Pratik Ringshia, Emily Dinan, Emma Qian, Siddharth Karamcheti, Shrimai Prabhumoye, Douwe Kiela, Tim Rocktaschel, Arthur Szlam, and Jason Weston. 2019. Generating interactive worlds with text.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. Incoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations*.

Matthew Hausknecht, Prithviraj Ammanabrolu, Marc-Alexandre Côté, and Xingdi Yuan. 2020. Interactive fiction games: A colossal adventure. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 7903–7910.

Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, Pierre Sermanet, Noah Brown, Tomas Jackson, Linda Luu, Sergey Levine, Karol Hausman, and Brian Ichter. 2022. Inner monologue: Embodied reasoning through planning with language models.

Peter A Jansen. 2021. A systematic survey of text worlds as embodied natural language environments. *arXiv preprint arXiv:2107.04132*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. 2018. NL2Bash: A corpus and semantic parser for natural language interface to the linux operating system. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan. European Language Resources Association (ELRA).

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A conversational paradigm for program synthesis. *arXiv e-prints*, pages arXiv–2203.

OpenAI. 2023. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*.

Mohammadreza Pourreza and Davood Rafiei. 2023. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *arXiv preprint arXiv:2304.11015*.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. 2020. Alfworld: Aligning text and embodied environments for interactive learning. *arXiv preprint arXiv:2010.03768*.

Jack Urbanek, Angela Fan, Siddharth Karamcheti, Saachi Jain, Samuel Humeau, Emily Dinan, Tim Rocktäschel, Douwe Kiela, Arthur Szlam, and Jason Weston. 2019. Learning to speak and act in a fantasy text adventure game.

Ruoyao Wang, Peter Alexander Jansen, Marc-Alexandre Côté, and Prithviraj Ammanabrolu. 2022. Scienceworld: Is your agent smarter than a 5th grader? *ArXiv*, abs/2203.07540.

Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational automated program repair.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium. Association for Computational Linguistics.

A Evaluation

A.1 Additional notes on technical validity measurement

Validity measurements are reported in order, such that failure of a function called earlier in the API implies failure for all subsequent tests. We note, however, that the `initializeWorld()` and `generateTaskDescription()` functions are evaluated only once, at the beginning of the game, while the `generatePossibleActions()`, `calculateScore()`, and `step()` functions are necessarily evaluated at each step.