

Towards Runtime Behavior Generation in Games

*Nicholas Jennings
Björn Hartmann, Ed.*

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-153

<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-153.html>

August 2, 2024



Copyright © 2024, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

For a more complete account of this project including results from interviews with game developers, please refer to the UIST 2024 Paper by the entire research team: What's the Game, then? Opportunities and Challenges for Runtime Behavior Generation. I would like to thank my research advisor Björn Hartmann for his guidance and support throughout the entire research process. James Smith, Han Wang and Isabel Li all worked alongside me for this project, and were instrumental to the development and analysis of GROMIT. I'd additionally like to acknowledge Cathy Wang, Roshan Nagaram, and Alvin Bao for their help in developing an early prototype of the GROMIT system.

Towards Runtime Behavior Generation in Games

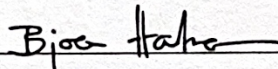
Nicholas Jennings

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

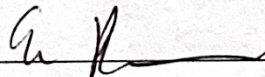
Committee



Björn Hartmann
Research Advisor

July 22, 2024

(Date)



Eric Paulos
Second Reader

1 Aug 2024

(Date)

Abstract

Procedural content generation (PCG), the process of algorithmically creating game components instead of manually, has been a common tool of game development for decades. Recent advances in large language models (LLMs) enable the generation of game behaviors based on player input at runtime. Such code generation brings with it the possibility of entirely new gameplay interactions that may be difficult to integrate with typical game development workflows. We explore these implications through GROMIT, a novel LLM-based runtime behavior generation system for Unity. When triggered by a player action, GROMIT generates a relevant behavior which is compiled without developer intervention and incorporated into the game. We create three demonstration scenarios with GROMIT to investigate how such a technology might be used in game development. In a system evaluation we find that our implementation is able to produce behaviors that result in significant downstream impacts to gameplay. We outline a future work agenda to address these concerns, including the need for additional guardrail systems for behavior generation.

Acknowledgement

For a more complete account of this project including results from interviews with game developers, please refer to the UIST 2024 Paper by the entire research team: **What's the Game, then? Opportunities and Challenges for Runtime Behavior Generation** [15]

I would like to thank my research advisor Björn Hartmann for his guidance and support throughout the entire research process. James Smith, Han Wang and Isabel Li all worked alongside me for this project, and were instrumental to the development and analysis of GROMIT. I'd additionally like to acknowledge Cathy Wang, Roshan Nagaram, and Alvin Bao for their help in developing an early prototype of the GROMIT system.

Contents

1	Introduction	5
2	Related Work	6
2.1	Procedural Content Generation	6
2.2	Runtime Generative AI in Video Games	8
2.3	Scene Manipulation	9
3	Runtime Behavior Generation	10
3.1	System Design	10
3.2	Demo: Sandbox	13
3.3	Demo: Escape Room	14
3.4	Demo: Adventure Game	15
4	System Evaluation	16
4.1	Explicit Scene Manipulation	16
4.2	Implicit Rule Generation	17
5	Discussion and Future Work	19
6	Conclusion	21
	Bibliography	22
1	Appendix	24

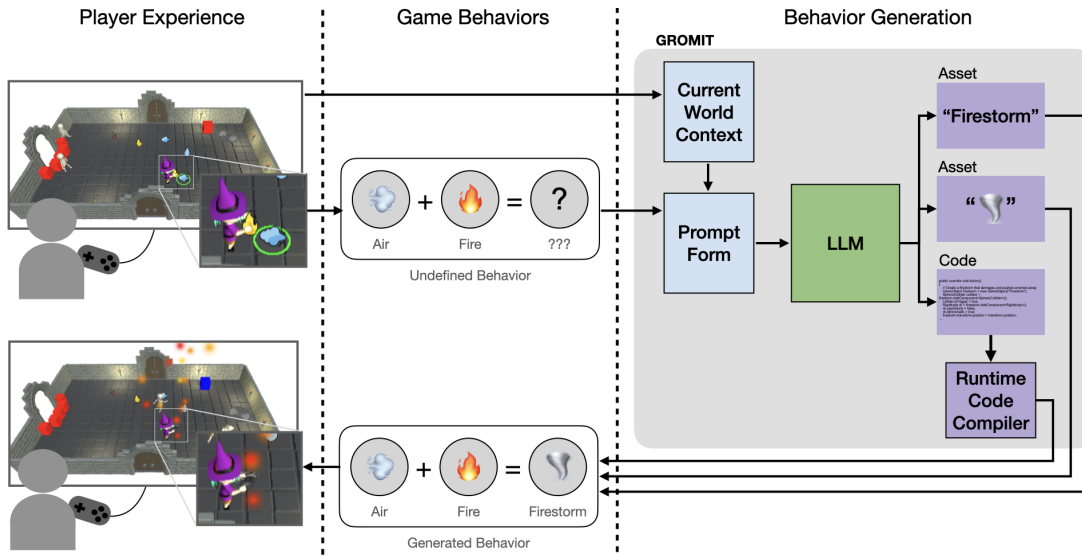


Figure 1: Example of runtime behavior generation in an adventure game. When the player initiates an interaction with no developer-defined output, the generation system is invoked, creating the name, description, and code defining the behavior of a new object to complete the interaction. The behavior code is compiled without developer intervention and the resulting object is incorporated into the game.

1 Introduction

In game development there are well established practices to generate game content algorithmically rather than manually, in a processes known as Procedural Content Generation (PCG). PCG use cases range from relatively straightforward tools for speeding up game development, to game-defining systems enabling novel player experiences. PCG tools have been developed to generate most types of game content, from textures to animations to entire virtual worlds. A major exception is game behaviors, the programmed rules, mechanics, and actions that define how the game itself is played. Due to their open-ended nature, game behaviors have so far largely stayed beyond the purview of generative systems. Traditional methods of procedural generation require the design space to be parameterized in some way, and parameterizing the design space of game behaviors requires severe restrictions that limit the scope of the generation. For instance, there is no clear way to parameterize "power-ups for a platformer game", but LLMs have no trouble creating designs for such a feature.

Recent advancements in the capabilities of Large Language Models (LLMs) promise to solve the technical side of this problem. LLMs don't require a properly defined design space, and so are promising candidates for navigating the semantic ambiguity inherent in game behavior requests.

An important distinction in generative systems is whether the output content is first viewed by developers during the game's initial development, or by players afterwards. In this paper we use the terms *devtime* and *runtime* to describe each scenario respectively¹. There is nothing inherently limiting this sort of behavior generation to tools used by developers in a devtime context. Implementing **Runtime Behavior Generation (RBG)** would allow for more personalized interaction with individual players. Compared to a devtime system, runtime systems can incorporate player input, and can cover a larger potential design space than could be manually checked by a developer. Optimistically, this method of behavior generation can enable entirely new forms of gameplay and player agency, and offer a new dimension of exploration. What would a runtime gameplay behavior generation system look like, concretely? How might game developers go about creating and incorporating such a system?

In this paper we begin exploring these questions through a design probe. We created a system capable of runtime gameplay behavior generation, and used it to create three demos of possible use-cases. We then ran these demos through quantitative tests to outline the efficacy of such a system, and highlight certain development pitfalls.

2 Related Work

2.1 Procedural Content Generation

Procedural Generation, the act of creating data algorithmically rather than manually, is a common approach to efficiently creating a large amount of content. PCG systems have been used in many stages of game development and for most game systems [13]. Tools like Material Maker allow developers to quickly generate textures and materials [28]. A large library of systems exists for generating foliage [25, 31]. While these tools can be used while a game is being developed, many of the most notable PCG systems are run without direct developer oversight. Brewer analyzes the 30-year legacy of *Rogue*, this influential game's procedurally generated dungeons and items

¹These terms heavily overlap with the definitions of 'online' and 'offline' used in prior PCG work, which describe whether the content generation occurs after or prior to the game being shipped to players, although they differ in some key edge cases. For example, in *No Man's Sky* the game universe was generated prior to the game's release, but is far too large for any significant portion to have been manually checked by developers prior to the shipping. In this case the world generation is technically offline, but is still considered runtime by our definition.

	Assets	Behaviors
Devtime	Speedtree [31] Material Maker [28]	Ludi [3]
Runtime	Rogue [2] Minecraft [21]	GROMIT

Table 1: Generation types of a sample of prior work in game development contexts, along with the GROMIT RBG system introduced in this paper.

enhanced its exploration and replay potential, and inspired the massively popular "roguelike" video game genre [2]. Games such as Minecraft [21] and No Man's Sky [12] use procedural generation to create entire virtual worlds for players to discover. World generation systems that additionally account for player challenge have also been developed [5]. Nitsche et. al. demonstrate a world generation system capable of combining player input with procedural methods [22]. Beyond these straightforward examples, Compton et. al. note that generative methods have been used for many systems that may not typically be considered "Content" [6].

Procedural Content Generation has also been applied to the rule sets and behaviors of a game. The design spaces of full game genres or mechanics are too large to be reasonably parameterized, so prior work has explicitly chosen sub-spaces to work with [27]. Togelius and Schmidhuber used a discrete 15x15 grid populated with a player controlled agent and various colored objects, then used an evolutionary system to create games based on the grid layout and object behavior [30]. Browne et al. developed the Ludi system, which generates board games in the style of tic-tac-toe or Go [3]. Chu et al.'s BPAIt system allows a developer to parameterize a design space in the Unreal engine, and then explore that space in a structured manner [4]. In all cases, explicit restrictions on game rules are given which allow for a structured search of the design space. This requires some level of human designer involvement, so these tools are devtime. A grid organizing some of these prior works along with the prototype RBG system used in this paper can be seen in Table 1. By exchanging explicit restrictions with semantic requests, we can develop systems that interpret user input for design restrictions in a runtime setting.

Khaled et. al. provide a set of metaphors describing the uses of PCG systems. Systems can be treated as a *Tool*, used by a developer as part of the game design process. Systems can also be seen as *Designers* which undertake design tasks alongside human developers, and as *Materials* which are dynamically generated [16]. Which metaphors are used affects how a PCG system is thought of by designers and developers, and in this paper we note how these metaphors can be applied to a RBG system.

2.2 Runtime Generative AI in Video Games

In game development, use of devtime Generative AI systems is already somewhat common. The GDC 2024 State Of The Game Industry Report shows 31% of developers use some form of Generative AI tools such as ChatGPT, DALL-E, and GitHub Copilot [10]. Tools are also being built specifically for game development. Muse is a suite of AI tools for Unity, which allows developers to prototype code, 2D art, animations, and conversation text-trees [29].

Systems also exist for the runtime scenario. Rieder demonstrated that a machine learning based system could be used as a game mechanic to power a runtime generative material [26]. In the industry space, Infinite Craft is a sandbox game where the player combines object icons to form an endless number of new objects [1]. In a similar vein, Suck Up! is a comedy adventure game where the player takes the role of a vampire convincing AI-powered townsfolk to let the vampire into their house [24].

Volum et al. use a large language model to write API code for piloting a character in the popular video game Minecraft [32]. The agent is able to chain API function calls in response to user prompting, and can coordinate with human-controlled characters to perform complicated in-game tasks such as mining for specific items and solving escape-room puzzles. Inworld.ai is a commercial software for creating AI non-playable characters (NPCs) for video games [14]. While Inworld gives developers a high degree of control over how the NPCs will behave narratively by prompting the agent's personality and knowledge of the virtual environment, the NPC's ability to interact with the in-game world is largely left up to the developer to implement. These are all examples of runtime content generation that follow behaviors manually created by the developers, RBG differs in that the possible actions themselves can also be generated.

While not specific to games, Park et. al.'s Generative Agents system also tackles the problem of piloting characters through a virtual environment called Smallville, and takes the approach of using LLMs to directly manage Smallville as well as the agents [23]. The state of virtual objects is determined by natural language prompts, which allows the generative agents to interact with their environment in the same format they run on internally. This has the benefit of making the Smallville robust to changes made by the agents, so most actions made by agents can have a true effect on the game state. This makes Smallville an example of a true RBG system. However, since each interaction equates to at least one LLM query, this approach would not scale to full sized real-time games. To be playable on a personal computer RBG systems still require most moment-to-moment gameplay to be handled by traditional code.

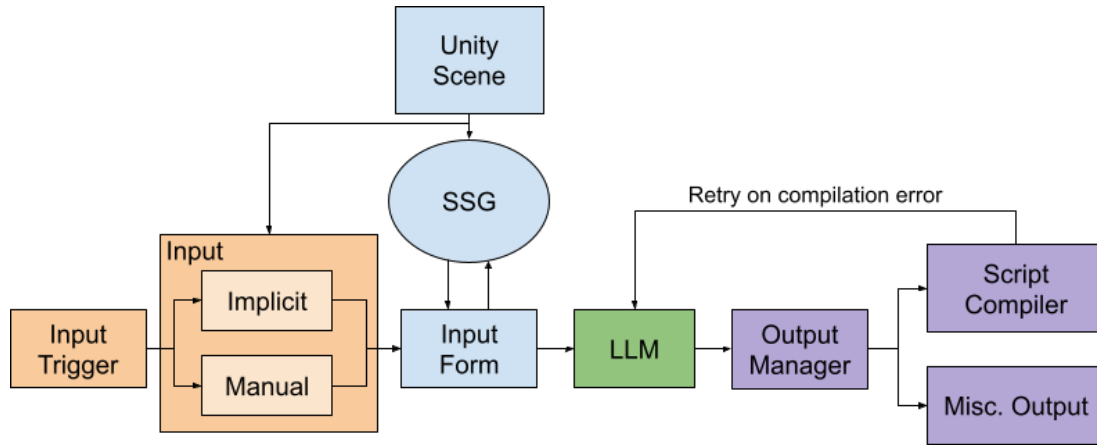


Figure 2: System Diagram for GROMIT. The input system, scene understanding, LLM, and output system are highlighted in orange, blue, green, and purple respectively. Once triggered, the input system combines implicit and manual input. This is then combined with the semantic scene graph to create an input form, which is sent to the LLM. Code from the LLM output is compiled and added to the virtual environment. If compilation fails, the error is sent to the LLM and the request is retried. Depending on the application, manual input may not be used and miscellaneous output may change.

2.3 Scene Manipulation

Behavior generation depends upon the context of the scene inside which objects exist. Attempts to achieve scene understanding and manipulation date back to 1968 with the SHRDLU system [34], where its users could move colored blocks using natural language. A shared approach to the representation of virtual environments is through the use of a Semantic Scene Graph (SSG), which structures the environment in terms of nodes and links, representing spatial relations. SSGs provide adaptivity to 3D interaction tasks [8], allow semantic control over generated content [9], and function in a way to generate and manipulate 3D scenes [33]. We use SSGs to interface an LLM with a 3D environment.

LLMR is a complete GenAI scene manipulation tool, including object and animation generation and behavior generation [7]. They focus on the explicit prompts to the generative system; we are interested in how our RBG system affects game developers' workflows.

3 Runtime Behavior Generation

Our analysis of related work indicates that Runtime Behavior Generation systems suitable for realtime games have been under-explored. As such, we elected to take a research-through-design approach to investigate systems of this nature. We built an example of a runtime behavior generation system, and used it to construct three demo experiences that sample the space of possible use cases.

The ways in which runtime behavior generation systems can be used in a game can be partitioned into three main types:

1. **Fully Generative Games:** Generating a significant portion of all behaviors at runtime, resulting in a game that largely adapts itself to the wants of any particular user. In this case, the generative system has a large degree of control over the whole game.
2. **Partially Generative Games:** Manually creating key game behaviors, but generating edge-case behaviors as they are encountered by the player. In this case, the generative system has a small degree of control over the whole game.
3. **Games With Generative Mechanics:** Some combination of the first two cases, choosing a particular section of the game to utilize behavior generation, and keeping the rest hand-crafted. In this case the generative system has a large degree of control over a small portion of the whole game.

Each of our demos embodies one of these use cases, and we use the demos to evaluate our RBG system’s capabilities, as well as a demonstrative tool to help with communicating these use cases in interviews with game developers.

3.1 System Design

To gain insights into the specific properties of a runtime behaviour generation system we built an example of such a system, which we call GROMIT². GROMIT generates behaviors by making requests to a LLM for program code which it then compiles and runs.

Where prior behavior generation systems restrict their design space through explicit parameterization, GROMIT’s restrictions come from a combination of prior context of the game’s code/setting and the plain-text request prompt. This is an important distinction from filling out a

²Named after the scene in the Wallace and Gromit animated series where Gromit lays tracks for a train as the train is running.

pre-formatted API. The generality of this format is what allows the system to truly generate novel behaviors and is also the root cause of many of the issues that will be discussed in section 4.

GROMIT is built for the Unity3D game engine and uses GPT-4 as its LLM. These choices were made primarily due to our team’s prior experience with these tools. In Unity, game scenes are built up of a collection of entities, called GameObjects, each with an attached set of components. Most components are typed as Monobehaviors, which typically express a single behavior/attribute associated with the GameObject. At a high level, GROMIT works by recording a prompt for a desired behaviour, combing this prompt with contextual scene information, and then sending the request to a GPT-4. Depending on the usecase, the initial prompt may be created directly by the player, or indirectly based on their in-game actions. The request is always framed as requiring a JSON string containing C# code as part of the response, but may also require secondary information. GROMIT then takes the response from GPT-4 and compiles the C# code. The method of prompt recording, and the way in which the compiled code is linked to the rest of the project, differs depending on the use-case scenario.

The C# compilation system was adapted from a project by Sebastian Lague [17], and contains several compilation tricks to compensate for programming errors GPT-4 consistently makes. For instance, GPT-4 regularly does not include import statements and class names in the code snippets it returns. Rather than attempting to prompt engineer the LLM to add these features, which produces inconsistent results, we simply detect if the features are missing before script compilation, and add default imports and dummy class wrappers if necessary. Additionally, if a compilation error occurs GROMIT will re-prompt the LLM and include the compilation error. The general system diagram for GROMIT is shown in Figure 2, details of the input and output systems change depending on how the behavior generation is being used.

Scene information is formatted using a Semantic Scene Graph (SSG), as shown in Figure 3. The SSG represents each object in the Unity3D scene as a node, encompassing information such as the object’s name, description, and spatial data. These nodes are arranged in a hierarchy to reflect spatial and relational aspects of the scene. In the GROMIT implementation, nodes are manually defined by the developer, and are organized in their hierarchy based on a combination of their size/shape and their position in the existing Unity transform hierarchy. As objects move around during gameplay, events are triggered which recompute local portions of the SSG. Automatic methods for generating SSGs exist [19], however for our purposes of exploring behavior generation we found a simple manual implementation was sufficient.

This structure facilitates the conversion of the 3D scene into a JSON format, making it a text-based representation that is compatible with LLMs. A significant feature of the SSG is its ability

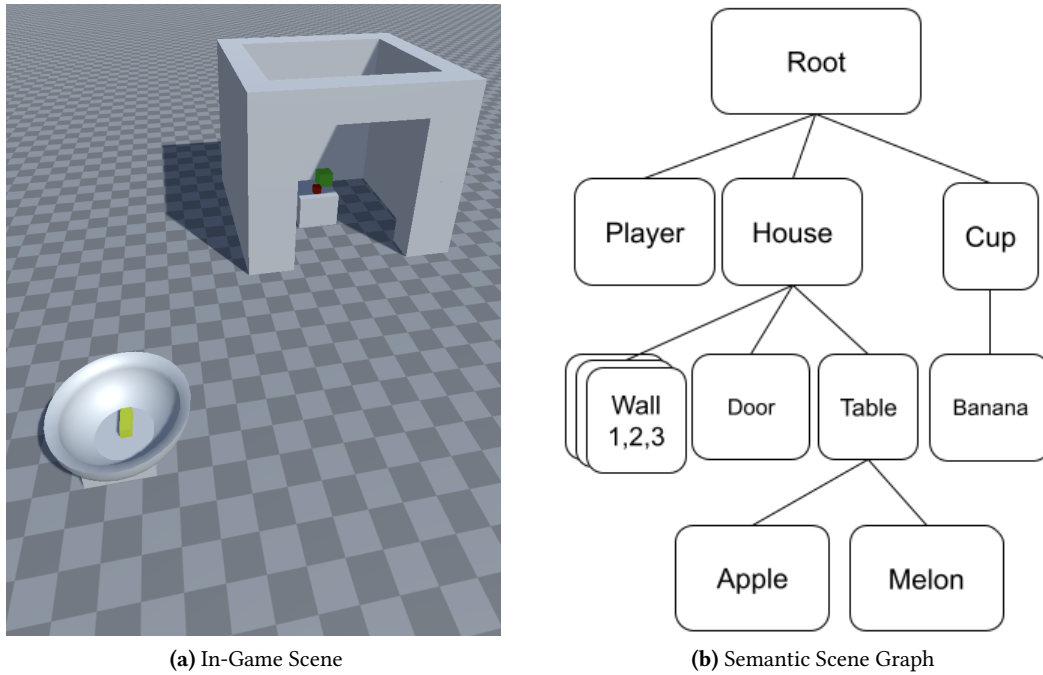
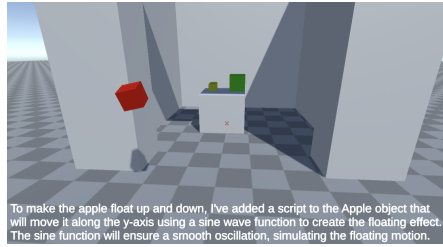


Figure 3: A sample game scene, and its resulting semantic scene graph. Each vertex in the graph contains additional data regarding its coordinates, behaviors, and text description. Vertices are labeled manually by the designer, and their position within the scene graph is calculated at runtime.

to filter nodes of its graph based on keywords derived from user input, streamlining the data processed by the LLM. For example, in the scene shown in Figure 3 if the initial prompt only mentions the table, then objects outside the house may be trimmed from the SSG before it is added to the prompt. This is done in the spirit of Retrieval-Augmented Generation, adding only the relevant information necessary for the request [18]. The scene graph is dynamic and capable of updating in real-time to reflect changes within the 3D scene, whether due to user interaction or LLM-driven modifications.

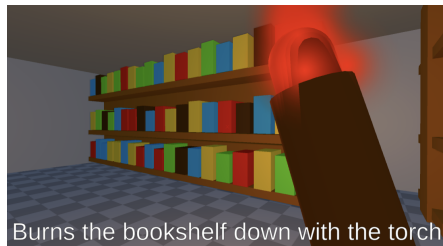
Depending on its usecase, GROMIT can be seen through either of Khaled et. al.'s **Tool**, **Designer**, and **Material** metaphors. Additionally, GROMIT can be used in an **Explicit** setting where a player deliberately invokes the generation, or in an **Implicit** setting where the generation is triggered by regular gameplay. Crucially, in implicit scenarios it may be possible to obfuscate that any generation is occurring at all.



(a) Blockland Scene



(b) Traffic Scene



(c) Escape Room Scene



(d) Adventure Game Scene

Figure 4: Screenshots of each demo scene, including written responses from GROMIT explaining newly generated behaviors. 4a Shows the Sandbox blockland scene, and the generation system’s response to the verbal request "Make the apple float around and tell me how you did it". 4b shows the Traffic scene for the Sandbox demo and GROMIT’s response to the request "Shrink the buildings and tell me how you did it". 4c shows the result of interacting the torch with one of the bookshelves in the library, revealing the bookshelf behind it. 4d shows the player using a ‘firestorm’ spell that was created by GROMIT.

3.2 Demo: Sandbox

In the Sandbox demo, GROMIT is used as a general tool for manipulating the environment. A screenshot of one of the scenes used for the Sandbox demo is shown in Figure 4a. Players click a button to begin recording input, and then provide a command to the system. For example, if a player says "make the apple spin", GROMIT will attach a script to the apple that makes it rotate over time. The primary medium for these instructions is spoken audio processed with a Whisper audio-to-text model. We utilized the Whisper-for-Unity asset [11], although direct calls to any speech-to-text API would suffice. The audio input can be supplemented with pointing gestures. Pointing is an incredibly common type of gesture, and has been well explored in prior work [20]. By casting a ray from the user’s reticle we can determine which object the user was pointing to and include this in the action. We also highlight the object for visual feedback to the user.

The LLM is prompted to complete the request by either writing code for a static method to be run once, or by writing a MonoBehaviour component to be added to a Unity GameObject. If the LLM writes a MonoBehaviour, it also provides the name of the GameObject it must be added to. From the player's perspective, they request a behavior change, wait a few seconds, and then that behavior change is manifested in the scene.

Two scenes were made for the Sandbox demo, a simple box scene called "Blockland" and a larger city scene called "Traffic". These scenes were used for investigating GROMIT's ability to work in differently constructed games, and are discussed in more detail in section 4.1.

This demo shows how GROMIT can be used as an **Explicit Tool** for players. There is a specific action, in this case a button press, that triggers the system, and the system is used to directly carry out a request by the player.

3.3 Demo: Escape Room

To explore the ability for GROMIT to generate new interactions from implicit input, we built the Escape Room demo. In the Escape Room demo, players are placed in a library and told to find a way to escape. The intended solution for the game is to search the library for a unique book. Behind the book is a key that unlocks the door to the library. Besides the door, key, bookshelves, and unique book, there are an additional 11 objects in the library room. While the only human-programmed interaction in the demo is the key opening the door, the UI action to trigger the interaction (holding an object and pressing the 'e' key) can be performed between nearly any pair of objects in the scene.

GROMIT is triggered when the player attempts to interact with a pair of objects that don't already have a defined interaction. The LLM is prompted to write a method to be run when the objects interact, as shown in Figure 5. This prompt is generated based on the names and descriptions of the interacting objects with no direct input from the player. Once the method is compiled and run it is linked to the rest of the demo such that subsequent interactions between the objects will call the method without triggering GROMIT. Each interaction consists of the method itself, and a text description. For example, interacting a torch with a bookshelf may generate a method that destroys the bookshelf object along with the description "Burns the bookshelf down with the torch", as seen in Figure 4c.

In this case, GROMIT is used as an **Implicit Designer**. The player never experiences the system being explicitly invoked as in the Sandbox demo, instead the system is triggered as necessary

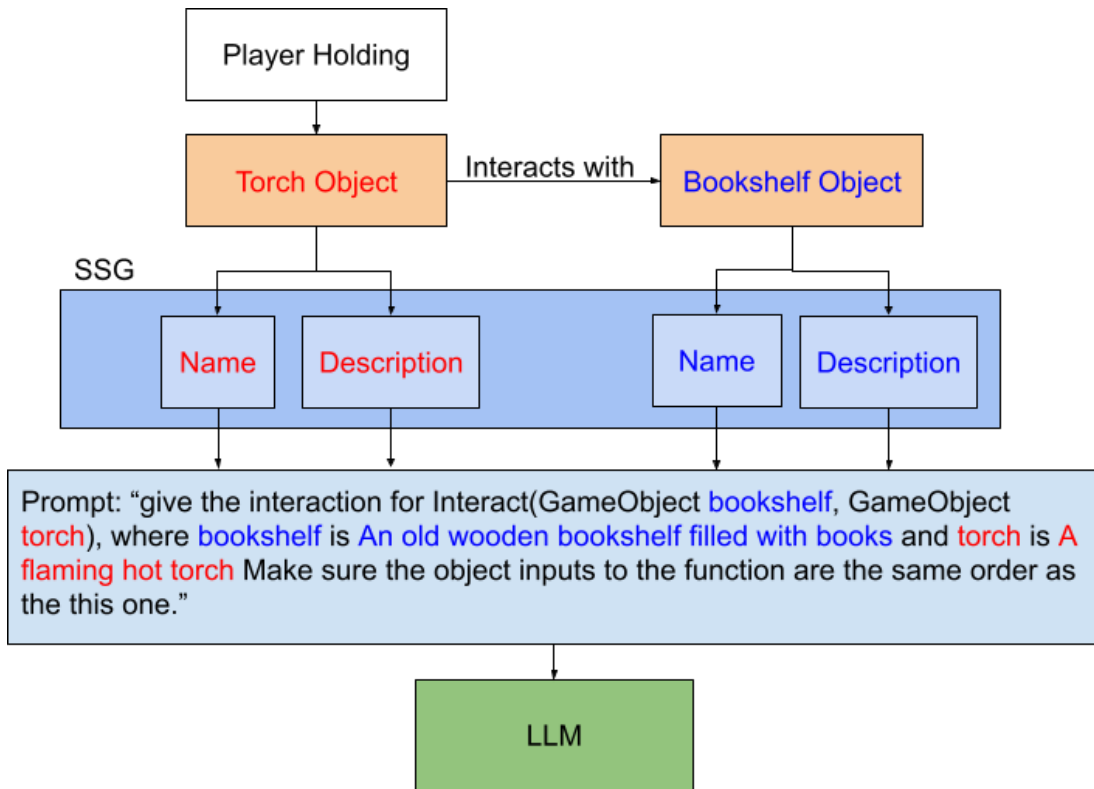


Figure 5: Implicit prompting process in the Escape Room demo. When a player performs an interaction with no existing behavior, the scene context is used to automatically prompt the LLM.

during the course of normal gameplay. GROMIT is also given a clear design task in not only implementing an interaction with a method, but also deciding what that interaction should be.

3.4 Demo: Adventure Game

The Adventure Game demo takes deliberate inspiration from the classic adventure game series The Legend of Zelda, which often sees the player navigating through a dungeon (shown in Figure 4d) by fighting enemies and solving puzzles. In the demo, the combat system has the player cast "spells", and any two spells can be combined to form a new spell in the style of Infinite Craft [1]. The initial set of spells were manually created by the authors, and spell combinations are created at runtime by GROMIT. The generation process can be seen in Figure 1. The puzzle system consists

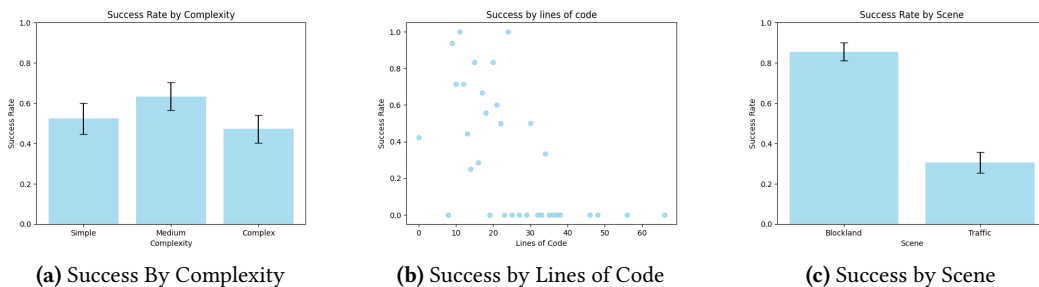


Figure 6: Success rate of requests in the Sandbox Demos. 6a shows the success rate by the user-supplied complexity on a 3 point scale. 6b shows the success rate by the line length of the script GROMIT attempted to run. 6c shows the success rate by the scene the requests were run in. Error bars in Figures 6a and 6c are for a 95% confidence level.

of switches and keys that change the world state and allow the player to reach different parts of the dungeon. Unlike the combat system, GROMIT has no direct control over the puzzle system.

In this demo, GROMIT can be seen as an **Implicit Material**. Similar to the Escape Room demo, the system is never invoked directly by the user. Where in the Escape Room demo GROMIT can define interactions between any two objects and these interactions can have any effects, in the Adventure Game demo GROMIT can only write the behaviors of new spell objects. Spells created by GROMIT can be fed back into GROMIT, so the entirety of the behavior generation in the demo can be abstracted as a property of the spell objects. In this sense, spells in the demo are a generative material powered by GROMIT.

4 System Evaluation

4.1 Explicit Scene Manipulation

To evaluate GROMIT’s ability to manipulate virtual scenes, we conducted a quantitative evaluation consisting of making various behavior requests in the Sandbox demo. Two scenes were used in the study. The first scene, Blockland, was a simple scene designed for testing GROMIT’s functionality and was built with GROMIT in mind. Blockland is shown in Figures 3a and 4a. The second scene, Traffic, was a larger scale traffic simulation imported from another project and is shown in Figure 4b. Traffic was not built with GROMIT in mind.

Behavior requests were collected through a Mechanical Turk survey. Survey respondents were instructed to provide 6 requests of **varying complexity** for GROMIT to perform for each of the

two scenes: 2 simple, 2 medium, and 2 complex. 25 surveys were given, which resulted in 145 unique requests after nonsensical or partial entries were removed.

Each request was run through GROMIT in its relevant scene and was marked by the authors as "Successful" or "Unsuccessful" based on whether the script written by GROMIT eventually compiled without errors and whether the effect of GROMIT's output could reasonably be said to complete the request. The length of GROMIT's output for each request was also recorded.

Overall, across the 145 unique requests, GROMIT achieved a success rate of 54%, successfully executing 78 requests while failing in 67. Comparing success rates by the complexity assigned in the survey submissions shows no correlation, as seen in Fig. 6a. A Chi-Squared test between the complexity groups did not show a significant difference (for all pairs $p > .25$). Linear regression shows a negative correlation between success rate and the line length of code outputted by GROMIT (Spearman's $r = -.6369$, $p < .005$, see Figure 6b). These results suggest that GROMIT has a harder time completing requests that are complicated to implement, but that humans use different internal metrics to judge complexity.

Comparing the success rates of requests by the test scene shows a clear difference ($p < .0005$, see Fig. 6c). Requests made in Blockland, which was designed with GROMIT in mind, tend to succeed with a 85% success rate, whereas requests made in the Traffic scene tend to fail with a 30% success rate. These results suggest that designing a program to be easily manipulable is necessary for GROMIT to be effective, and that programs designed without GROMIT in mind are unlikely to work well with it. For example, a number of requests in both scenes were some variation of "Change the color of X to C". In the Blockland scene all objects used standard Unity materials, which can have color filters applied in code. In the Traffic scene all objects used a mobile diffuse material, where the only way to change the color of the object is to edit the image file of the texture. This is comparatively very complicated to do in code. All of the color change requests succeeded in the Blockland scene, and failed in the Traffic scene. On average, GROMIT completed all requests in 10.6 seconds. The average time drops to 5.2 seconds when using the gpt-4o model.

4.2 Implicit Rule Generation

To determine if GROMIT has the ability to support implicit use cases, we ran a system evaluation to determine the success and failure rates of interactions in the Escape Room and Adventure Game demos.

To generate data for the Escape Room demo, we populated the room with additional items. In the MTurk survey we also asked participants for 10 additional items to include in an escape room.

The results were a sword, a wizard's hat, a potion, a relic, a stationary kit, a stray frog, a sheet of paper, a newspaper, an old walking stick, and a pen. We then used GROMIT's auto prompting to generate interactions between each pair of items. Besides the 10 items from the survey, there were 5 items (a torch, bookshelves, the special book, the key, the door) which we implemented for the intended puzzle solution. Since each item was not allowed to interact with itself, there were 105 interaction item pairs. We implemented the 1 interaction necessary for the intended solution (the key opens the door). Additionally, 2 objects (the door and the bookshelf) were stationary so could not interact with each other. This left 103 potential interactions for GROMIT to generate. We used GROMIT to generate interactions between each viable object pair.

To generate data for the adventure game, we started with 11 spells created by the paper authors. These were combined in breadth-first order until 205 spells had been generated by GROMIT. We categorized each of the behaviors generated by GROMIT based on whether the generation was successful, meaning the generated behavior ran without errors and aligned with its text description. If the generation was unsuccessful, the category of error was also recorded. The results are summarized in Figure 7.

In the Escape Room Demo, none of the combinations resulted in compiler errors and 6 resulted in runtime errors. Another 10 had had text descriptions that suggested an event should happen, but the written code did not produce this event. GROMIT responded that 14 of the pairs shouldn't have any interaction. The remaining 73 pairs gained some form of successful interaction. 68 of these interactions appear to be mainly visual changes, such as a torch heating up a sword by changing its color to red, but 5 interactions resulted in alternate solutions to the escape room. These alternate solutions either destroyed the bookshelves in some way, allowing a faster way to find the key, or created new "magical" objects such as wands or staffs which could open the door.

In the Adventure Game Demo, 30 interactions resulted in continued compilation errors after re-prompting, 1 resulted in runtime errors, 6 did not produce an effect related to their description (in most of these cases the spell simply deleted itself on use), and 12 did not produce any effect. The remaining 156 attempts produced new functional spells. Of these, 9 could be used in some meaningful way outside of combat. These spells either allowed the player to trigger the puzzle switches from new locations, or enhanced player movement through various kinds of teleportation.

In the implicit rule generation study, all failure cases resulted in no change to game behavior, usually due to the generated scripts not interacting with the scene. In the explicit scene manipulation study, 62 of the 67 failure cases similarly resulted in no visible changes to game behavior. The 5 remaining cases resulted in visible errors, but these fell short of crashing the game or preventing continued gameplay. One of the more dramatic occurred when the request "Implement a day-night

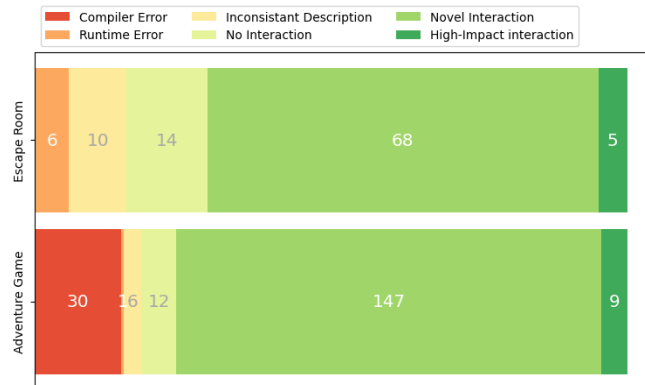


Figure 7: Results of automatically generating interactions with GROMIT. A result was considered a compiler or runtime error if the behavior threw an error that prevented it from compiling or running properly. A result was categorized as an "Inconsistent Description" if it ran without errors but had a significant mismatch between how the behavior was described and what it actually did. Results where the LLM responded that the behavior should produce no effect were marked as "no interaction". If the behavior did produce some effect it was labeled a "Novel Interaction". If a Novel Interaction was found to cause significant gameplay changes it was recategorized as a "High-Impact Interaction".

cycle" resulted in a script which disabled the light source in the blockland scene. We have created some requests that can cause larger-scale errors than were seen in the study. For example, in the traffic scene the request "Make each building spin individually" can instead cause all buildings to spin around a central point. In theory, generated code could crash the entire game though we have not observed this in our testing. No behavior generated for either demo resulted in the game crashing in any way. This is largely due to GROMIT handling compilation errors internally, and runtime errors being limited in scope. On average, GROMIT completed all requests in 5.6 and 14.1 seconds for the escape room and adventure game scenes respectively. The average times drop to 2.6 and 6.1 seconds when using the gpt-4o model.

5 Discussion and Future Work

The complete version of this project also includes an interview study with n=13 game developers using GROMIT as a probe to elicit their current opinion on runtime behavior generation tools, in order to enumerate the specific themes curtailing the wider use of such tools.

This paper has demonstrated RBG in several small-scale scenarios. We expect that incorporating common LLM scaling strategies, such as adding a planning stage to the behavior generation pipeline, could allow GROMIT-like systems to scale to larger tasks. Properly assessing scalability would ideally involve creating a large-scale game system incorporating RBG, which we leave to future work.

Future work should also investigate player perspectives on RBG. Exploring, among other things, if and how players can detect RBG, how their gameplay changes with knowledge of RBG, and player opinion on the general use of Generative AI. Comparing those findings with developer opinions would help both to characterize the relationship between developers and players with this technology, and to inform developer decisions on making games with RBG.

Finding a process to better control RBG systems is a primary concern, which is unsurprising when comparing RBG to commonly used PCG systems. Devtime PCG systems are often experienced as tools used to develop a part of the resulting game, which gives developers two avenues of control. They can develop the PCG system itself, and/or they can verify the output of the system before it is included in the main game. In this sense content created by an devtime system can still express developer intent even if the developer was not responsible for the PCG software itself, such as when using a GenAI system. In terms of ownership, the output of an devtime system acts similarly to content purchased from an asset store. The asset itself might not have been created by the developer, but its inclusion is still a vector of developer intent.

Runtime systems, in contrast, only have the first avenue of control. Without the ability for developers to verify output before it is shown to users, the developer impact on the PCG output can only come from their effect on the PCG system itself. This makes PCG systems that are both runtime and GenAI-based problematic, since they have no direct method of developer control. The high-level choice to use the runtime tool at all is certainly still made by the developer, but their level of control is significantly reduced.

LLMs already provide several methods of developer input. Prompts can be partially engineered by the developers. Few-shot examples can be provided to demonstrate intended output. Indeed, we used both these methods in GROMIT to achieve basic functionality. However, these methods don't necessarily fit with the requirements expressed by developers. Depending on the desired guardrails, there may be better interfaces for expressing the requirements.

Additionally, based on the results of section 4, as well as the general quality concerns expressed by developers, restricting only the input to the model may be insufficient. We identify that GROMIT and behavior generation systems with a similar implementation have 4 main avenues for restriction implementations. These are:

1. Modified input to the LLM
2. Static analysis of code generated by the LLM
3. Dynamic analysis of LLM-generated code in a sandboxed environment
4. Rollback/Undo functionality if restriction violation occurs

A "Guardrail System" that maps the constraint descriptions from a vocabulary developers already use to a set of implementations from the above list could improve on both the usability and effectiveness of an approach using only existing LLM control methods. Ideally, this could restore the avenue of direct control present in traditional PCG systems. We conclude that there's a need for a set of such Guardrail Systems, although the degree to which such tools can/should be generalized between games is unclear. Future work should explore the efficacy of specific guardrail systems, both in their ability to control a RBG system and in their alignment with developer needs.

6 Conclusion

In this paper, we solidified the emerging concept of Runtime Behavior Generation as it applies to the games industry. Through three concrete examples, we explored possible ways RBG can be used for games. We conducted a system evaluation and found that, using our current system, generated behaviors can achieve a high success rate if other game systems are designed to be easily manipulable through code. We also found that some generated behaviors can have significant effects on gameplay. We highlight potential future work that could address these challenges.

Bibliography

1. N. Agarwal. *Infinite Craft*. 2024. URL: <https://neal.fun/infinite-craft/>.
2. N. Brewer. “Computerized dungeons and randomly generated worlds: From rogue to minecraft [scanning our past]”. In: *Proceedings of the IEEE* 105:5, 2017, pp. 970–977.
3. C. Browne and F. Maire. “Evolutionary Game Design”. In: *IEEE Transactions on Computational Intelligence and AI in Games* 2:1, 2010, pp. 1–16. DOI: 10.1109/TCIAIG.2010.2041928.
4. E. Chu and L. Zaman. “Exploring alternatives with unreal engine’s blueprints visual scripting system”. In: *Entertainment Computing* 36, 2021, p. 100388.
5. K. Compton and M. Mateas. “Procedural level design for platform games”. In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. Vol. 2. 1. 2006, pp. 109–111.
6. K. Compton, J. C. Osborn, and M. Mateas. “Generative methods”. In: *The fourth procedural content generation in games workshop, pcg*. Vol. 1. 2013.
7. F. De La Torre, C. M. Fang, H. Huang, A. Banburski-Fahey, J. Amores Fernandez, and J. Lanier. “LLMR: Real-time Prompting of Interactive Worlds using Large Language Models”. In: *Proceedings of the CHI Conference on Human Factors in Computing Systems*. CHI ’24. Association for Computing Machinery, Honolulu, HI, USA, 2024. ISBN: 9798400703300. DOI: 10.1145/3613904.3642579. URL: <https://doi-org.libproxy.berkeley.edu/10.1145/3613904.3642579>.
8. Y. Dennemont, G. Bouyer, S. Otmame, and M. Mallem. “3D Interaction assistance through context-awareness”. In: *2012 IEEE Virtual Reality Workshops (VRW)*. 2012, pp. 103–104. DOI: 10.1109/VR.2012.6180903.
9. H. Dharmo, F. Manhardt, N. Navab, and F. Tombari. “Graph-to-3D: End-to-End Generation and Manipulation of 3D Scenes Using Scene Graphs”. In: *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*. 2021, pp. 16332–16341. DOI: 10.1109/ICCV48922.2021.01604.
10. B. Elderkin. *GDC 2024 state of the game industry: DEVS discuss layoffs, Generative AI, and more: News: GDC: Game Developers Conference*. 2024. URL: https://gdconf.com/news/gdc-2024-state-game-industry-devs-discuss-layoffs-generative-ai-and-more%3F_mc%3Dedit_gdcsf_gdcsf_le_x_17_x_2024%26kcode%3DBLG_GDC.
11. A. Evgrashin. *Macoron/whisper.unity: Running speech to text model (whisper.cpp) in unity3d on your local machine*. 2023. URL: <https://github.com/Macoron/whisper.unity>.

12. H. Games. *No man's sky*. 2024. URL: <https://www.nomanssky.com/>.
13. M. Hendrikx, S. Meijer, J. Van Der Velden, and A. Iosup. "Procedural content generation for games: A survey". In: *ACM Trans. Multimedia Comput. Commun. Appl.* 9:1, 2013. ISSN: 1551-6857. DOI: 10.1145/2422956.2422957. URL: <https://dl.acm.org/doi/10.1145/2422956.2422957>.
14. Inworld.ai. 2023. URL: <https://inworld.ai/>.
15. N. Jennings, H. Wang, I. Li, J. Smith, and B. Hartmann. "What's the Game, then? Opportunities and Challenges for Runtime Behavior Generation". In: *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*. UIST '24. Association for Computing Machinery, New York, NY, USA, 2024.
16. R. Khaled, M.J. Nelson, and P. Barr. "Design metaphors for procedural content generation in games". In: *Proceedings of the SIGCHI conference on human factors in computing systems*. 2013, pp. 1509–1518.
17. S. Lague. *Seblague/runtime-csharp-test*. 2019. URL: <https://github.com/Seblague/Runtime-CSharp-Test>.
18. H. Li, Y. Su, D. Cai, Y. Wang, and L. Liu. "A survey on retrieval-augmented text generation". In: *arXiv preprint arXiv:2202.01110*, 2022.
19. T. Liu, S. Chaudhuri, V.G. Kim, Q. Huang, N.J. Mitra, and T. Funkhouser. "Creating Consistent Scene Graphs Using a Probabilistic Grammar". In: *ACM Trans. Graph.* 33:6, 2014. ISSN: 0730-0301. DOI: 10.1145/2661229.2661243. URL: <https://dl.acm.org/doi/10.1145/2661229.2661243>.
20. D. McNeill. *Hand and mind: What gestures reveal about thought*. University of Chicago press, 1992.
21. Mojang. *Minecraft*. 2024. URL: <https://www.minecraft.net/en-us>.
22. M. Nitsche, C. Ashmore, W. Hankinson, R. Fitzpatrick, J. Kelly, and K. Margenau. "Designing procedural game spaces: A case study". In: *Proceedings of FuturePlay 2006*, 2006.
23. J. S. Park, J. O'Brien, C. J. Cai, M. R. Morris, P. Liang, and M. S. Bernstein. "Generative agents: Interactive simulacra of human behavior". In: *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. 2023, pp. 1–22.
24. Proxima. *Hilarious vampire adventure game*. 2023. URL: <https://www.playsuckup.com/>.
25. A. de la Re, F. Abad, E. Camahort, and M. C. Juan. "Tools for procedural generation of plants in virtual scenes". In: *Computational Science—ICCS 2009: 9th International Conference Baton Rouge, LA, USA, May 25-27, 2009 Proceedings, Part II 9*. Springer. 2009, pp. 801–810.
26. B. Rieder. "Using procedural content generation via machine learning as a game mechanic". In: *Austrian Marshall Plan Foundation*, 2018.
27. N. Shaker, J. Togelius, and M. J. Nelson. "Procedural content generation in games". In: 2016.
28. R. Suescun. *Material maker*. 2023. URL: <https://www.materialmaker.org/>.

29. U. Technologies. *Unity muse*. 2024. URL: <https://unity.com/products/muse>.
30. J. Togelius and J. Schmidhuber. "An experiment in automatic game design". In: *2008 IEEE Symposium On Computational Intelligence and Games*. 2008, pp. 111–118. DOI: 10.1109/CIG.2008.5035629.
31. I.D. Visualization. *SpeedTree*. 2024. URL: <https://store.speedtree.com/>.
32. R. Volum, S. Rao, M. Xu, G. DesGarenes, C. Brockett, B. Van Durme, O. Deng, A. Malhotra, and B. Dolan. "Craft an Iron Sword: Dynamically Generating Interactive Game Characters by Prompting Large Language Models Tuned on Code". In: *Proceedings of the 3rd Wordplay: When Language Meets Games Workshop (Wordplay 2022)*. Ed. by M.-A. Côté, X. Yuan, and P. Ammanabrolu. Association for Computational Linguistics, Seattle, United States, 2022, pp. 25–43. DOI: 10.18653/v1/2022.wordplay-1.3. URL: <https://aclanthology.org/2022.wordplay-1.3>.
33. J. Wald, H. Dhano, N. Navab, and F. Tombari. "Learning 3D Semantic Scene Graphs From 3D Indoor Reconstructions". In: *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020, pp. 3960–3969. DOI: 10.1109/cvpr42600.2020.00402.
34. T. Winograd. "Understanding natural language". In: *Cognitive psychology* 3:1, 1972, pp. 1–191.

1 Appendix

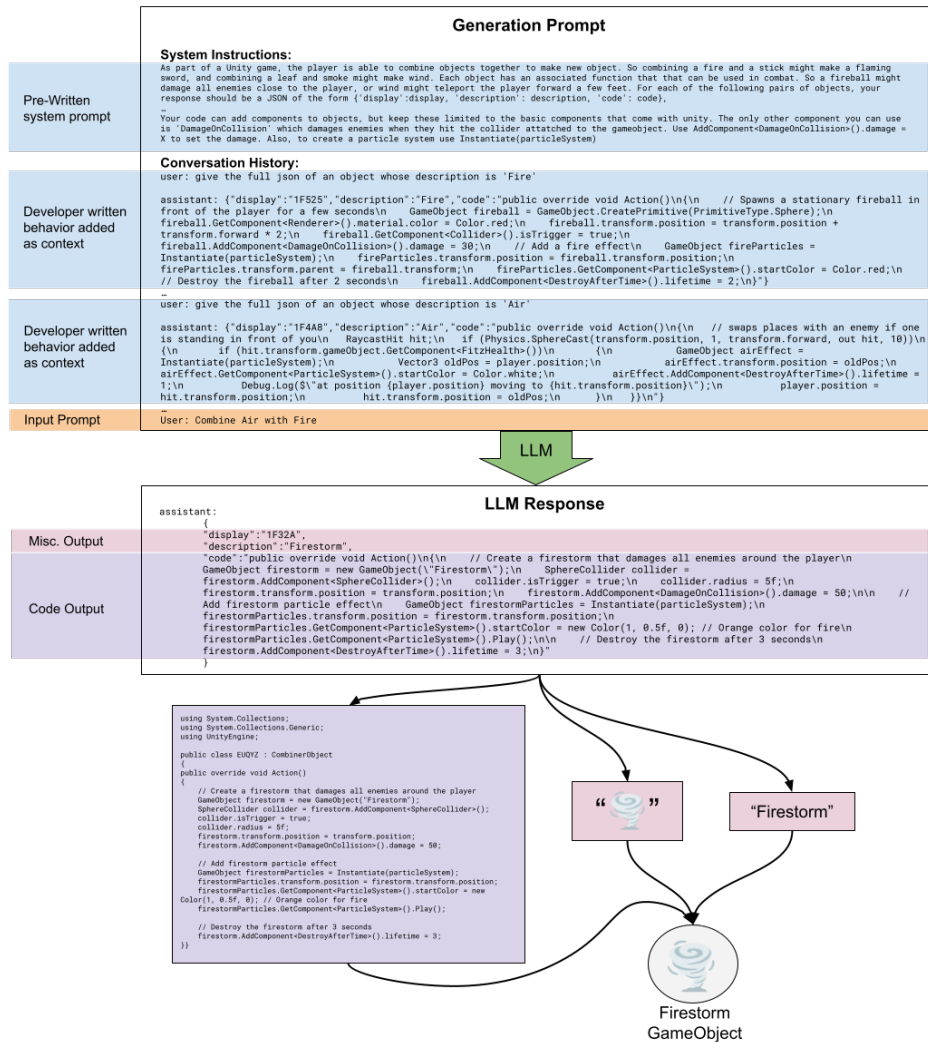


Figure 8: Diagram of an example prompt from the Adventure Game demo with labeled components. When the player combines the Fire and Air spells, a prompt is generated to request a new spell. Although they were manually created by a developer, the existing spells are included in the prompt as if the LLM had generated them so that the JSON format can be reused. The LLM output JSON is then interpreted and the code compiled. The compiled behavior is then combined with the emoji and plaintext output to generate the resulting spell.