What's the Game, then? Opportunities and Challenges for Runtime Behavior Generation

Nicholas Jennings
nicholasjennings@berkeley.edu
University of California, Berkeley
Berkeley, California, USA

Han Wang
hanw@berkeley.edu
University of California, Berkeley
Berkeley, California, USA

Isabel Li
ilyues@berkeley.edu
University of California, Berkeley
Berkeley, California, USA

James Smith
james.smith@berkeley.edu
University of California, Berkeley
Berkeley, California, USA

Björn Hartmann
bjoern@eecs.berkeley.edu
University of California, Berkeley
Berkeley, California, USA

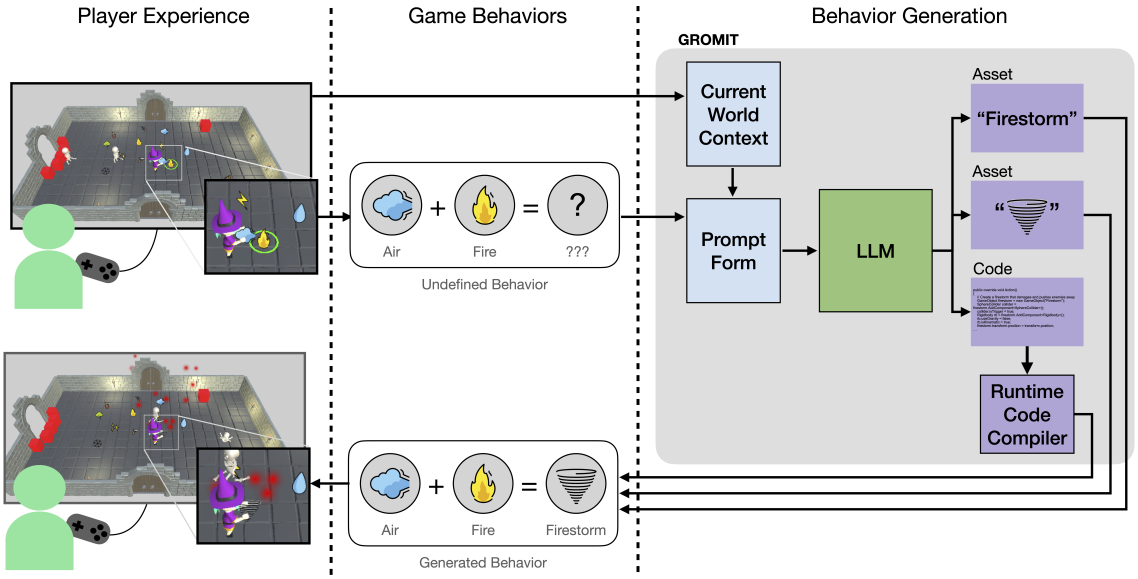


Figure 1: Example of runtime behavior generation in an adventure game. When the player initiates an interaction with no developer-defined output, the generation system is invoked, creating the name, description, and code defining the behavior of a new object to complete the interaction. The behavior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

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 then conduct an interview study with n=13 game developers using GROMIT as a probe to elicit their current opinion on runtime behavior generation tools, and enumerate the specific themes curtailing the wider use of such tools. We find that the main themes of concern are quality considerations, community expectations, and fit with developer workflows, and that several of the subthemes are unique to runtime behavior generation specifically. We outline a future work agenda to address these concerns, including the need for additional guardrail systems for behavior generation.

UIST '24, October 13–16, 2024, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-0628-8/24/10
https://doi.org/10.1145/3654777.3676358

CCS CONCEPTS

• **Computing methodologies** → *Semantic networks*; • **Human-centered computing** → *Interaction design.*

KEYWORDS

Human-AI interaction, Generative AI, Procedural Content Generation

ACM Reference Format:

Nicholas Jennings, Han Wang, Isabel Li, James Smith, and Björn Hartmann. 2024. What's the Game, then? Opportunities and Challenges for Runtime Behavior Generation. In *The 37th Annual ACM Symposium on User Interface Software and Technology (UIST '24), October 13–16, 2024, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3654777.3676358

1 INTRODUCTION

In game development there are well established practices to generate game content algorithmically rather than manually, 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 [8, 14].

A major exception is game behaviors: the programmed rules, mechanics, and actions that define how the game itself is played. Game behaviors have so far largely stayed beyond the purview of PCG systems due their open-ended nature, as well as the difficulties in balancing variety and quality of output [4, 28].

Traditional methods of procedural generation utilize a variety of technical approaches, such as parametric design space exploration, generative grammars, and machine learning based methods to generate assets. When applied to behaviors, these approaches are necessarily restrictive in scope due to their inability to generate arbitrary computer programs [25]. Because of this, interactive behaviors usually require manual implementation in code by a developer.

The emergence of Large Language Models (LLMs) has given developers novel tools to turn high level specifications into running code. It is thus timely to investigate if LLMs can be used for game behavior generation. They don't require a properly defined parametric space, and so are promising candidates for navigating the semantic ambiguity inherent in game behavior requests. The challenge becomes how to formulate prompts to the LLM in order to retrieve code that fits with the rules of the game, and how to integrate those code responses into the game runtime. This necessitates a systems exploration into how to support going from player input or action to coherent game behavior through an LLM.

There are two ways in which generative systems can be applied to creating game content. The first approach is to create content during the development process. These tools afford the developer access to content with infinite variation, but still rely on developer judgement to choose which generated artifacts to include in the final product. All players that play the game see the same generated asset. The other approach is to generate assets at **runtime**—while the game is running on a player's local machine. This approach allows developers to provide users with experiences that are unique to them, and unique to each time they play through the game, but developers relinquish control in the process. Implementing **Runtime Behavior Generation (RBG)** would allow for more personalized interaction for individual players, as the actions they take can trigger individually unique responses from the game. It also affords player experiences and interactions that the developers did not originally intend, which could lead to positive and negative effects. 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 prototype system built to investigate research questions around RBG. We created a system capable of runtime gameplay behavior generation, and used it to create three working demonstrations of possible use-cases. We evaluated these demonstrations to understand the efficacy of such a system, and highlight certain development pitfalls. Additionally, we conducted interviews with 13 game developers, using the demonstrations to ground our discussion of their attitudes towards runtime behavior generation. We conducted a thematic analysis of these interviews to identify the main reactions of developers.

Through our quantitative analysis, we found that our implementation can generate behaviors with up to a 85% success rate, so long as the rest of the game systems have been built with RBG in mind. We additionally found that our demos that implicitly generated behaviors based on user actions resulted in new behaviors in 71%-76% of cases, and that a small portion of cases (4%-5%) resulted in new behaviors that caused significant downstream gameplay changes. Our developer interviews reveal that these major consequences are not always desired from a design perspective, depending on the game additional guardrails on the RBG system are necessary. We further find that developers have concerns about the quality of current LLM output, and about how using RBG systems can mesh with the expectations of their playerbase.

Our work makes three key contributions.

(1) A system design for integrating runtime behavior generation in a modern game engine.
(2) Evaluations to characterize the quantitative performance of applying RBG to several game and scenario prototypes.
(3) A thematic analysis of interviews with game developers regarding implementing RBG into their workflows.

We conclude with a discussion on the challenges and ways forward to fully embodying RBG concepts into games, and suggest several avenues for future work.

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 [14]. Tools like Material Maker allow developers to quickly generate textures and materials [26]. A large library of systems exists for generating foliage [8, 30]. While these tools can be used while a game is being

	Assets	Behaviors
Devtime	Speedtree [30] Material Maker [26]	Ludi [4]
Runtime	Rogue [3] Minecraft [20]	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.

developed, many of the most notable PCG systems are run without direct developer oversight. Brewer analyzes the 30-year legacy of *Rogue*, an influential game whose procedurally generated dungeons and items enhanced its exploration and replay potential, and inspired the massively popular "rogue-like" video game genre [3]. Games such as Minecraft [20] and No Man's Sky [13] 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 [6]. Nitsche et al. demonstrate a world generation system capable of combining player input with procedural methods [21]. Beyond these straightforward examples, Compton et al. note that generative methods have been used for many systems that may not typically be considered "content" [7].

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 [25]. Togelius and Schmidhuber used a discrete 15 by 15 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 [28]. Browne et al. developed the Ludi system, which generates board games in the style of tic-tac-toe or Go [4]. Chu et al.'s BPAlt system allows a developer to parameterize a design space in the Unreal engine, and then explore that space in a structured manner [5]. 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 [11]. Tools are also being built specifically for game development. Muse is a suite of AI tools for the Unity game engine which allows developers to prototype code, 2D art, animations, and conversation text-trees [27].

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 [24]. 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 [23].

Volum et al. use a large language model to write API code for piloting a character in the popular video game Minecraft [31]. The agent is able to chain API function calls in response to user prompting. It 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 [15]. 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 [22]. 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.

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 [33], 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 [9], allow semantic control over generated content [10], and function in a way to generate and manipulate 3D scenes [32]. We use SSGs to interface an LLM with a 3D environment.

LLMR is a complete GenAI scene manipulation tool, including object and animation generation as well as behavior generation [29]. Where LLMR focuses on explicit prompts to the generative system,

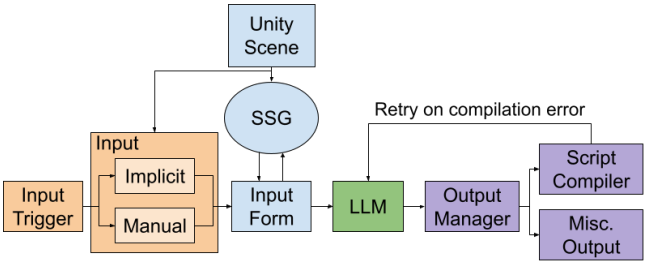


Figure 2: System Diagram for GROMIT. The input system, scene understanding, LLM, and output system are highlighted in orange, blue, green, and purple respectively. Once triggered, the input system combines implicit and manual input. This is then combined with the semantic scene graph to create an input form and 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.

we are interested in how our RBG system affects game developers' workflows.

3 RUNTIME BEHAVIOR GENERATION

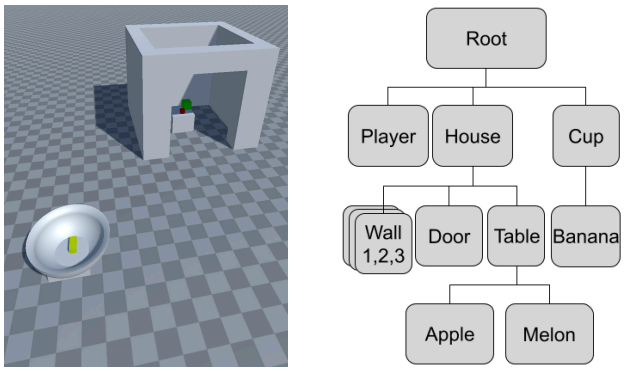
We begin our investigation into RBG systems by determining the different cases in which they can be used. One important distinction in generative systems is whether the output content is first viewed by developers during the game's initial development, or by players afterwards. In this paper we use the terms *devtime* and *runtime* to describe each scenario respectively[1].

Our analysis of related work indicates that Runtime Behavior Generation systems suitable for real-time games have been underexplored. 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.

[1]These 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.



(a) In-Game Scene (b) Semantic Scene Graph

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.

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

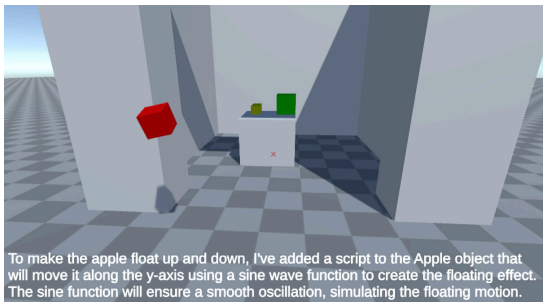
The key technical contribution of our work is the design and implementation of a functional RBG system. It addresses three core challenges:

(1) integrating knowledge of the environment into the LLM prompt;
(2) handling recompilation in the presence of errors in generated code; and
(3) generating code at runtime in response to player input.

We describe our system architecture in Figure 2, and details of our working prototype in Section 3.2.

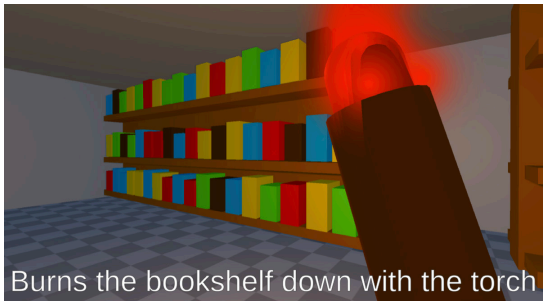
3.1.1 Input Processing. The system is initiated through some type of trigger or action. In our investigation we focus on instances where a player action triggers the system, but RBG systems are not generally limited by this. We consider two types of input, which type is used is dependent on the application. **Explicit** initiations are where the user is directly prompting the system for a behavior, such as asking the system to move an object from one place to another. These are explicit because the action the user takes is to directly prompt the system. **Implicit** initiations come from user actions that indirectly create prompts to the system, such as trying to combine two objects into a new object. The actions users take in this case are of a different form than the prompt sent to the system. See Figure 5 for an example of how implicit inputs are processed.

3.1.2 Environment. In order for the LLM to create a coherent output, some representation of the environment in which the action is being taken must be provided. This is because many user actions

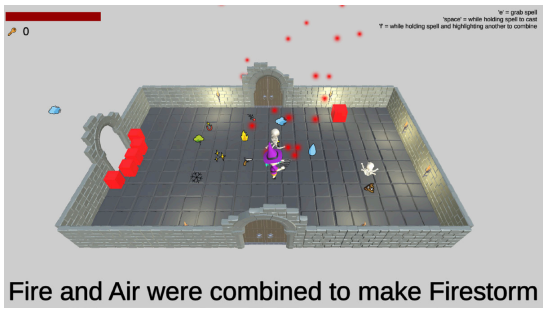


(a) Blockland Scene

(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.

are often associated with objects in the environment (see the two examples in the prior section). There are many ways to represent the environment, such as object relation hierarchy as in the LLMR work [29] as well as many examples of using semantic scene graphs (SSGs) [2, 10, 32]. We utilize SSGs in our work for their ability to describe object relationships without requiring a strict hierarchical relationship. This also allows objects to have semantic relationships to many objects in a scene.

3.1.3 Large Language Model. The input actions and environment are fed into an LLM with a custom designed prompt on a **per application** basis. The request is always framed as requiring code as part of the response, but may also require secondary information as shown in Figure 8 (Appendix). In our implementations, these pre-written prompts are only concerned with creating the immediate behavior required of the system, and don't require the LLM to consider any long-term goals. The high generality of the natural language and C# programs managed by the LLM ideally allows GROMIT to adapt to many different requests.

3.1.4 Output Processing. The output of the prompt can be anything from descriptions, labels, code, and more. If requesting descriptive information, the LLM can be asked to return its reply in JSON format so that the data can be read and processed directly. When code is the output, that code must be compiled and loaded into the software's runtime. Errors encountered during this compilation can be dealt with by re-prompting the system with the error information, and asking for an updated code snippet. This process is repeated until a functional code sample can be compiled.

3.2 Implementation: GROMIT

To gain insights into the specific properties of a runtime behavior generation system we built an example of the previously described system, which we call GROMIT[2]. GROMIT generates behaviors by making requests to a LLM for program code which it then compiles and runs. 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 and their common use in prototyping in the research community. GROMIT 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.

Unity uses C# as a scripting language so our prompts request that returned code snippets are in that language. The C# compilation system was adapted from a project by Lague [17], and contains several compilation tricks to compensate for programming errors GPT-4 consistently makes such as forgetting import statements and forgetting class wrappers.

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

An example of the SSG datastructure that we generate can be seen in Figure 3. The SSG represents each object in the Unity 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.

This structure facilitates the conversion of the 3D scene into a JSON format, making it a text-based representation that is compatible with LLMs. In the spirit of Retrieval-Augmented Generation [18], the SSG can be filtered to only include nodes related to the input request. 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.

Depending on it's 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. We discuss this possibility more in section 6.2.1.

3.3 Demo: Sandbox

In the Sandbox demo, GROMIT is used as a general tool for manipulating the environment. Screenshot of the scenes used for the Sandbox demo are shown in Figure 4a and Figure 4b. Players click a button to begin recording input, and then provide a command to the system. For example, if a players says "make the apple spin", GROMIT will attach a script to the apple that makes it rotate over time. The primary medium for these instruction is spoken audio processed with a Whisper audio-to-text model. We utilized the Whisper-for-Unity asset [12], 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 [19]. 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. Example interactions with GROMIT for each demo can be found in the video figure.

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.

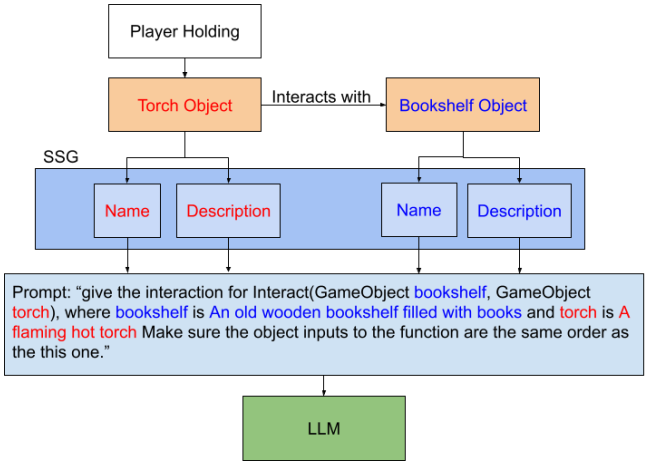


Figure 5: Implicit prompting process in the Escape Room demo. When a player performs an interaction with no existing behavior, the scene context is used to automatically prompt the LLM.

3.4 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 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.

3.5 Demo: Adventure Game

While we initially thought that the previous two demos gave sufficient coverage of RBG games for our purposes, after our first 5 developer interviews (discussed in sections 5 and 6) it became apparent that developers were particularly interested in the third category of RBG games, **Games with Generative Mechanics**. We created the Adventure Game demo as an example of this case in particular. 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 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.

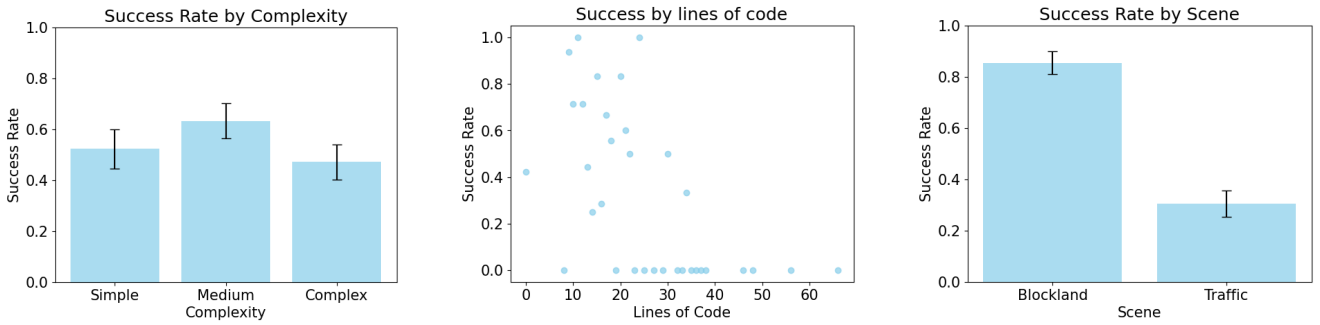
Comparing the success rates of requests by the test scene shows a clear difference ($p < .0005$, see Fig. 6c). Requests made in Blockland, which was designed with GROMIT in mind, tend to succeed with a 85% success rate, whereas requests made in the Traffic scene tend to fail with a 30% success rate. 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 with a standard deviation of 7.5 seconds. The average time drops to 5.2 seconds with a standard deviation of 3.6 seconds when using the gpt-4o model. Game developers often handle other high-latency tasks such as scene loading, so the request generation time is a significant but not insurmountable design challenge.

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



(a) Success By Complexity (b) Success by Lines of Code (c) Success by Scene

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.

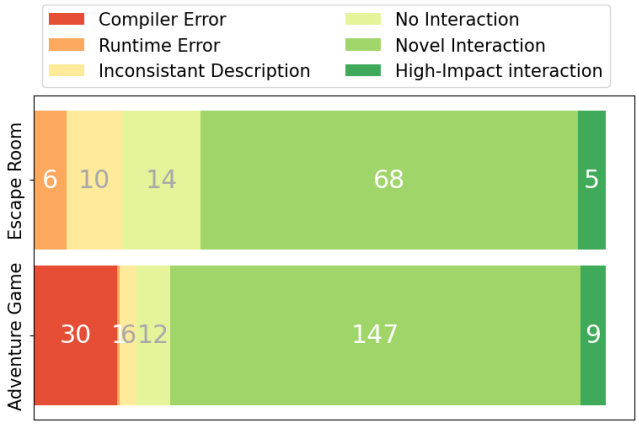


Figure 7: Results of automatically generating interactions with GROMIT. A result was considered a compiler or runtime error if the behavior threw an error that prevented it from compiling or running properly. A result was categorized as an "Inconsistent Description" if it ran without errors but had a 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".

behavior ran without errors and aligned with it's 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 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 it's 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 errors occurred when the request "Implement a day-night 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. Although these results can only be used as a proof of concept that GROMIT can support implicit interactions, we also identify a number of cases where the generated behaviors have downstream consequences discussed further in our follow up study with developers. On average, GROMIT completed all requests in 5.6 (SD = 3.8) and 14.1 (SD = 2.9) seconds for the escape room and

Participant	Experience
P1	Student, prior experience in Indie
P2	Student, prior experience in AAA
P3	Indie
P4	Student
P5	Indie, prior experience in AAA
P6	Student
P7	Hobbyist
P8	Indie
P9	AAA
P10	Hobbyist
P11	Hobbyist and Student
P12	Indie and AAA
P13	Indie and Student

Table 2: Background of interviewed game developers. Here, AAA refers to experience at mid-to-large sized game companies, and Indie refers to experience working individually or with smaller teams.

adventure game scenes respectively. The average times drop to 2.6 (SD = 1.3) and 6.1 (SD = 2.2) seconds when using the gpt-4o model.

5 DEVELOPER INTERVIEW METHODS

Our system evaluation shows that GROMIT can be a useful tool for Runtime Behavior Generation, but our findings are limited to the example demos that we created and evaluated, and provides only a few design guidelines in how to employ such systems. To learn more about how RBG could be utilized in games and to gain an understanding of the current industry view of such systems, we conducted interviews with practicing game developers with a variety of backgrounds and experience.

5.1 Recruitment

We recruited 13 game developers from a variety of backgrounds. 4 developers had experience in AAA development, 6 developers had experience as indie developers, 5 developers were university students intending to work in the games industry, and 3 developers were hobbyists. Several developers had experience in multiple of these categories. A complete list of developer backgrounds can found in table 2. Participants were recruited through a mix of posts on online game development spaces, and direct emails to developers with a strong internet presence. Interviews lasted 60 minutes, and participants were compensated with a $20 Amazon gift card.

5.2 Protocol

Interviews consisted of a 60-minute recorded Zoom session. At the beginning of each interview we collected the developers consent to be recorded, as well as some demographic information and background on their history in game developement and prior experience with GenAI and PCG tools. To ground our discussion, we then showed the interviewee each of the GROMIT demos and answered any technical questions they had[3]. We then spent the remainder of the interview discussing the developers perspective on RBG as it applied to their game development process. This semi-structured portion of the interview focused on the developer's creation process and their expectations of the reception of such games.

5.3 Thematic Analysis

Interviews were recorded and transcribed using automatic software to assist in analysis. We then synthesized the recorded interviews into codes and themes, focusing on the broad categories of use-cases proposed by developers as well as barriers identified by them. After coding was completed, we discussed and organized the themes through a series of group meetings.

6 DEVELOPER INTERVIEW FINDINGS

All developers we interviewed expressed positive interest in seeing RBG used in either their own games or games in general, but subsequently wanted more information about the degree of control and fine-tuning they are able to achieve when crafting a game using runtime behavior generation. Three primary themes emerged from our qualitative analysis of the interviews: RBG's fit into developer workflow; game community and expectations; and technical quality of runtime generated behaviors.

6.1 RBG and Developer Workflow

6.1.1 Scope of RBG. Devtime behavior generation tools, such as ChatGPT, were already used for ideation and prototyping by several of the interviewees, and it's apparent that runtime tools could also be useful in this process. Of our 13 interviewees, all 13 mentioned that they could use behavior generation to help them think of game features. A particular advantage of the generation occuring at runtime is that it allows player input to be incorporated. P12 proposed adding an RBG system to a game for playtesting, and tracking what behaviors players were likely to generate. These desire paths could then be manually implemented with higher quality for the game's full release. P5 similarly described an "open alpha sandbox" where the developers would continuously work to fix quality issues with the RBG output produced by players.

Developers had many ideas of games they could make using RBG. We organize these by the scope of RBG within the game, reusing the categories we mention in section 3. The first category, **Fully Generative Games**, are games made almost entirely with generative systems. A GROMIT-style behavior generator could be combined with other generative tools to make a sandbox that adapts to the needs of the player. Some form of this idea was expressed by every interviewee and is roughly illustrated in our Sandbox demo.

The second category, **Partially Generative Games**, has a developer manually create the key portions of a large-scale game, and fill in the rest using RBG. P9 felt that the current way PCG is used in large-scale games led to "sparse" environments that could be made more interesting if behaviors were generated. P6 had a similar opinion, but felt specifically, citing quality concerns, that

[3]The first 5 interviews were conducted prior to the creation of the Adventure Game demo, so those interviewees (P1-5) were only shown the Sandbox and Escape Room demo.

generated behaviors needed a high "degree of freedom" to be implemented imperfectly but still be fun for players. Since core game mechanics had a lower degree of freedom, they reasoned that RBG systems should never be the "star of the show" for a game. In contrast, several developers (P8,P11,P13) thought that implementing RBG required a large commitment from developers, and therefore it only made sense to use RBG as a major system in the game. While a smaller-scale RBG system that makes only minor behavior changes is technically feasible, the cost of such a system may not be worth the outcome. P8 said "I think I would only use this technology if it was the center point of whatever game I was making, because it feels like too much overhead to not do that."

The third category, **Games with Generative Mechanics**, incorporates the RBG system into a single mechanic or domain of a larger game that is otherwise mostly human-made. P3 suggested that inconsequential systems, such as the interaction of set dressing in a first-person shooter (FPS), would make good candidates for behavior generation. P5 felt that the success of the recently released Baldur's Gate 3 suggested an RPG with generated mechanics could be popular as long as the story remained hand-crafted. This category is not well represented by either the Sandbox or Escape Room demo, in which GROMIT has complete direct access to the entire game scene. To discuss the concept more concretely in later interviews, we developed the Adventure Game demo to demonstrate what a game might look like with behavior generation only applied to one subsystem.

6.1.2 Authoring Guardrails. A major challenge of developing a game in the second or third categories described in the previous section is defining and restricting the generated behaviors to their given subsystem. Developers who created games which focused on problem-solving, puzzles, and challenges as opposed to free exploration raised the issue of RBG creating "game-breaking" mechanics (P1, P2, P7, P10, P11). Game-breaking mechanics usually referred to generated mechanics which allowed players a shortcut around a core developer-designed game experience, depriving the player of a satisfying solution or key part of the game narrative. P13 noted that in some cases any amount of behavior creation could be unwanted, since they could distract the player from the actions needed to progress in the game. Developers described several ways to demarcate the influence of runtime behavior generators, which we list here.

Limitation by Style/Genre P3's portfolio of games have a particular visual style they had become known for, and their primary concern with using a runtime system was that the behaviors would not match this style. Beyond simply ensuring high quality, they noted that their plan to retain control over all interaction animations (discussed further in section 6.3) would also constrain all behaviors to their own visual style.

Limitation by Area One straightforward approach mentioned by P2 and P5 was to only enable the system in certain regions of the game. P2 imagined a single level in an RPG where most interactions were generated by a GROMIT-like system, but where any generated objects were marked and removed from the player's inventory once they left the level.

Limitation by Mechanic When shown the ways in which generated behaviors could bypass the intended solution to the Escape Room demo, several developers (P1,P2,P8,P10) characterized the problem as certain generated mechanics (e.g. fire burning down the bookshelves) encroaching on the territory of a non-generated mechanic. P2 said they would consider using an object interaction system similar to the one in the Escape Room demo as a puzzle mechanic in a horror game, but only if they could be sure the player still had to engage with the horror mechanics.

Limitation by Quantity Limiting the number of interactions and/or objects that can be directly affected by the GenAI system would reduce the possibility that these changes might effect other parts of the game. The example given by P3 was a FPS game where the RBG system can only directly affect set dressing items that wouldn't impact the core gameplay. In a more general strategy, P10 proposed that each mechanic/domain could be given numbered quotas of new behaviors, with more disruptive mechanics receiving lower quotas.

Limitation by Depth Level This is a limitation specific to objects created by behavior generation systems such as in the Escape Room and Adventure Game demo. Multiple developers (P1, P2, P4, P13) noted that objects could be categorized based on how many initial objects were combined to create the result. Objects made by a developer have a depth of 0. Objects generated by combining depth 0 objects are depth 1, and so on. Limitations on objects could be determined by their depth level. For example, enforcing a rule that "only objects with levels N and above can possibly open a door" can control the complexity of generated solutions. P13 noted that limiting generations past a certain depth level could be used to reduce overall calls to the LLM API and save cost.

Responding to the GROMIT demos, P13 framed guardrails as being necessary to incorporate "macro design" into systems that operated on a behavior-level. They described defining the guardrails as the "ultimate question" regarding implementing an RBG system. Some developers (P4, P5, P12) were hesitant to say that any form of limitation could actually work. P5 said "I'm filled with doubt that for something this powerful you can define enough constraints" and suggested that players and developers would need to "tolerate a certain amount of jank and strangeness".

6.2 Game Community and Expectations

Developers speculated about how players would receive games with generated behaviors. They also discussed what types of gameplay genres they believed could effectively incorporate RBG.

6.2.1 Player Expectations around GenAI Games. Developers were excited about the potential player experiences GROMIT enabled. P9 believed that RBG, with ideal output, could produce "magical" interactions where players feel like their imagination was directly translated into the game experience. Still, some developers theorised that there's a type of player who wants linear stories told by people, and so wouldn't enjoy any significant influence of any GenAI tool (P4, P5, P11).

Regarding how RBG games will be presented to the game community, developers (P1, P4, P5, P12) expressed that players should be made aware of GenAI use in their games to respect the ethical opinions of the playerbase. P4 said, "I think in order to be ethical these days ... the go-to is that you should say whether or not you've used Generative AI, especially in asset development ... Because

they're trained on other work, and you want to make transparent what it was trained on." In contrast, P3 felt that informing players how the system works would cheapen their experience, stating: "You want it to feel like magic, you know?... The more invisible the game is to the player, how it works, the more interesting it is". They were also concerned that, if players knew a runtime system was present, they would focus more on experimenting with the system than playing the game.

A number of developers (P4, P5, P10, P11, P12) believed players should be warned that the behaviors made by the generative system might change the theme and maturity rating of the game. P11 brought up Infinite Craft as an example, saying: "Let's say I thought they had a database back there, with all the different combinations. Then I'd be like, why did they choose this as a combination? ... Maybe I'm thinking from the point of reputation." Announcing that content was AI-generated could serve as both a content warning and an explanation for any designs made by the generative system.

6.2.2 Interaction with New and Existing Genres. Most of the interviewees felt that adding RBG to a game would change its genre (P1, P2, P4, P5, P7, P8, P10, P11, P12). One way this could happen, mentioned in the previous section, was for the generated behaviors to differ thematically from the original game content.

More broadly, some developers (P1, P2, P8, P10, P11) felt that if the system is given significant power over the behaviors, it changes the genre of the game regardless of the theming of the generated output. By creating more behaviors the player might interact with, P10 felt the game would shift its genre. They felt that any game with some significant RBG element would be "not quite open, and like free to expiration, but like leaning towards [that]. You have a little bit more freedom of choice there, as a player.".

Developers also discussed game genres where they thought RBG may not align well with key characteristics of the genre. P6, who creates puzzle games centered around cognitive problem-solving, said, "Puzzle games require a lot of intentionally, psychological thought, a flow of where the players attention goes...[GROMIT] is probably not trustworthy to use on runtime."

6.3 Quality of Runtime Generated Behaviors

Developers discussed their needs related to the quality of RBG game content, especially in relation to their existing content styles.

Many developers (P1, P3, P4, P5, P8, P10, P11, P13) expressed concern that GROMIT could deliver output that was detrimental to the player experience. Some developers worried about GROMIT's ability to crash or break the game with poor output. P4 and P5 thought that GROMIT's code synthesis was a serious security threat to a game's basic functionality. They were especially concerned about scenarios where players can directly prompt the system to have code written or changed, allowing them to either accidentally or maliciously alter the game and render it unplayable.

Many developers (P3, P4, P5, P8, P10, P13) didn't trust current LLMs to consistently produce satisfactory output that aligned with the goals and creative direction of the game. P8 felt that GPT-4 was "trained to do whatever is most predictable, and not necessarily follow my narrative in any way". P13 similarly said that while the Escape Room demo in particular was "promising" they still overall "don't trust these systems to be wild and creative." To improve the model, P5 suggested using a separate LLM to assess the output quality. P8 thought a fine-tuned model for behavior generation could improve quality, but wasn't sure it would be worth the effort.

Developers raised the need of matching generated behavior to high quality game art and assets. While discussing the Escape Room demo, P3 proposed that rather than using GROMIT to devise any possible interaction between objects, they would limit the generative system to intelligently choose from a set of pre-defined interactions for which assets (animation, visual effects, art, etc.) would already be created. This would ensure that generated behaviors could still be associated with high quality assets that stylistically matched the game's existing visual effects. P9, who was both a programmer and technical artist, speculated that in a high-production game they would need 3D models, texturing, and other complex assets to be generated alongside behavior at runtime, and were unsure how this might mesh with manual asset pipelines.

7 DISCUSSION AND FUTURE WORK

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.

Finding a process to better control RBG systems was a primary concern for many of the interviewed developers, 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 a 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.

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 the developer opinions expressed in section 6.2.1 would help both to characterize the relationship between developers and players with this technology, and to inform developer decisions on making games with RBG.

8 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 further used the example demos to ground interviews with 13 game developers, where we discussed potential design challenges to incorporating RBG in future projects. We highlight potential future work that could address these challenges.

ACKNOWLEDGMENTS

This work was in part supported by a gift from Accenture. We used the Mobile Traffic System 2.0 asset from the Unity Asset Store for the traffic demo. The adventure game demo used assets from Quaternius and Fornax and emojis from OpenMoji. The authors would like to acknowledge Cathy Wang, Roshan Nagaram, and Alvin Bao for their help in developing an early prototype of GROMIT.

REFERENCES

[1] Neal Agarwal. 2024. Infinite Craft. https://neal.fun/infinite-craft/
[2] Trevor Ashby, Braden K Webb, Gregory Knapp, Jackson Searle, and Nancy Fulda. 2023. Personalized Quest and Dialogue Generation in Role-Playing Games: A Knowledge Graph- and Language Model-Based Approach. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery. https://doi.org/10.1145/3544548.3581441
[3] Nathan Brewer. 2017. Computerized dungeons and randomly generated worlds: From rogue to minecraft [scanning our past]. *Proc. IEEE* 105, 5 (2017), 970–977.
[4] Cameron Browne and Frederic Maire. 2010. Evolutionary Game Design. *IEEE Transactions on Computational Intelligence and AI in Games* 2, 1 (2010), 1–16. https://doi.org/10.1109/TCIAIG.2010.2041928
[5] Eric Chu and Loutfouz Zaman. 2021. Exploring alternatives with unreal engine's blueprints visual scripting system. *Entertainment Computing* 36 (2021), 100388.
[6] Kate Compton and Michael Mateas. 2006. Procedural level design for platform games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, Vol. 2. 109–111.
[7] Kate Compton, Joseph C Osborn, and Michael Mateas. 2013. Generative methods. In *The fourth procedural content generation in games workshop, pcg*, Vol. 1.
[8] Armando de la Re, Francisco Abad, Emilio Camahort, and M Carmen Juan. 2009. 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, 801–810.
[9] Yannick Dennemont, Guillaume Bouyer, S. Otmane, and M. Mallem. 2012. 3D Interaction assistance through context-awareness. In *2012 IEEE Virtual Reality Workshops (VRW)*. 103–104. https://doi.org/10.1109/VR.2012.6180903
[10] Helisa Dhamo, Fabian Manhardt, Nassir Navab, and F. Tombari. 2021. 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)*. 16332–16341. https://doi.org/10.1109/ICCV48922.2021.01604
[11] Beth Elderkin. 2024. GDC 2024 state of the game industry: DEVS discuss layoffs, Generative AI, and more: News: GDC: Game Developers Conference. 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
[12] Alex Evgrashin. 2023. Macoron/whisper.unity: Running speech to text model (whisper.cpp) in unity3d on your local machine. https://github.com/Macoron/whisper.unity
[13] Hello Games. 2024. No man's sky. https://www.nomanssky.com/
[14] Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. 2013. Procedural content generation for games: A survey. *ACM Trans. Multimedia Comput. Commun. Appl.* (2013). https://doi.org/10.1145/2422956.2422957
[15] Inworld.ai. 2023. https://inworld.ai/
[16] Rilla Khaled, Mark J Nelson, and Pippin Barr. 2013. Design metaphors for procedural content generation in games. In *Proceedings of the SIGCHI conference on human factors in computing systems*. 1509–1518.
[17] Sebastian Lague. 2019. Seblague/runtime-csharp-test. https://github.com/SebLague/Runtime-CSharp-Test
[18] Huayang Li, Yixuan Su, Deng Cai, Yan Wang, and Lemao Liu. 2022. A survey on retrieval-augmented text generation. *arXiv preprint arXiv:2202.01110* (2022).
[19] David McNeill. 1992. *Hand and mind: What gestures reveal about thought*. University of Chicago press.
[20] Mojang. 2024. Minecraft. https://www.minecraft.net/en-us
[21] Michael Nitsche, Calvin Ashmore, Will Hankinson, Robert Fitzpatrick, John Kelly, and Kurt Margenau. 2006. Designing procedural game spaces: A case study. *Proceedings of FuturePlay* 2006 (2006).
[22] Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. 1–22.
[23] Proxima. 2023. Hilarious vampire game. https://www.playsuckup.com/
[24] Bernhard Rieder. 2018. Using procedural content generation via machine learning as a game mechanic. *Austrian Marshall Plan Foundation* (2018).
[25] Noor Shaker, Julian Togelius, and Mark J Nelson. 2016. Procedural content generation in games. (2016).
[26] Rodolphe Suescun. 2023. Material maker. https://www.materialmaker.org/
[27] Unity Technologies. 2024. Unity muse. https://unity.com/products/muse
[28] Julian Togelius and Jurgen Schmidhuber. 2008. An experiment in automatic game design. In *2008 IEEE Symposium On Computational Intelligence and Games*. 111–118. https://doi.org/10.1109/CIG.2008.5035629
[29] Fernanda De La Torre, Cathy Mengying Fang, Han Huang, Andrzej Banburski-Fahey, Judith Amores Fernandez, and Jaron Lanier. 2023. LLMR: Real-time Prompting of Interactive Worlds using Large Language Models. arXiv:2309.12276 [cs.HC]
[30] Interactive Data Visualization. 2024. SpeedTree. https://store.speedtree.com/
[31] Ryan Volum, Sudha Rao, Michael Xu, Gabriel DesGarennes, Chris Brockett, Benjamin Van Durme, Olivia Deng, Akanksha Malhotra, and Bill Dolan. 2022. 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)*, Marc-Alexandre Côté, Xingdi Yuan, and Prithviraj Ammanabrolu (Eds.). Association for Computational Linguistics. https://doi.org/10.18653/v1/2022.wordplay-1.3
[32] Johanna Wald, Helisa Dhamo, N. Navab, and Federico Tombari. 2020. Learning 3D Semantic Scene Graphs From 3D Indoor Reconstructions. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 3960–3969. https://doi.org/10.1109/cvpr42600.2020.00402
[33] Terry Winograd. 1972. Understanding natural language. *Cognitive psychology* 3, 1 (1972), 1–191.

A APPENDIX

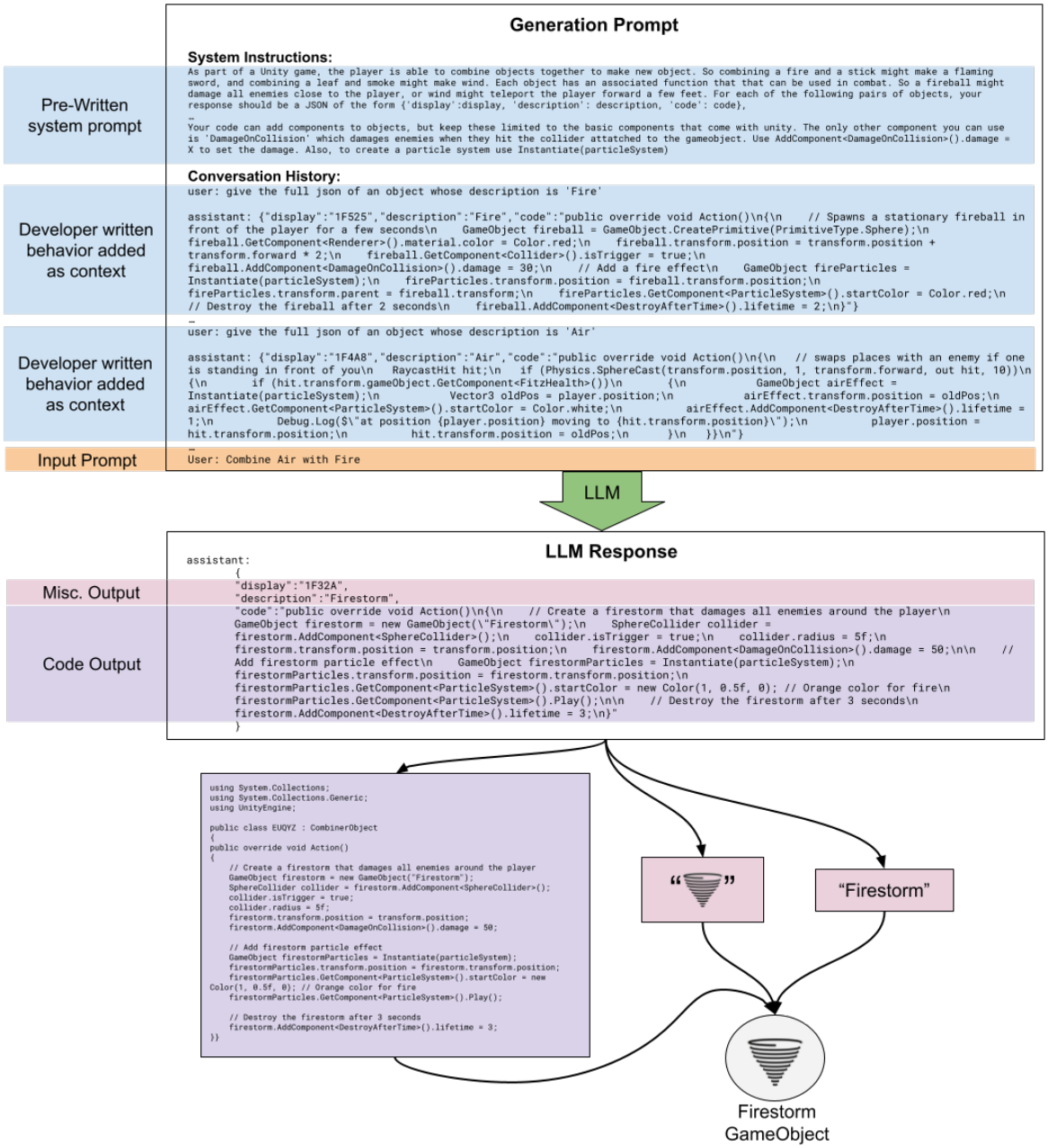


Figure 8: Diagram of an example prompt from the Adventure Game demo with labeled components. When the player combines the Fire and Air spells, a prompt is generated to request a new spell. Although they were manually created by a developer, the existing spells are included in the prompt as if the LLM 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.