LLMR: Real-time Prompting of Interactive Worlds using Large Language Models

Fernanda De La Torre*
dlatorre@mit.edu
Massachusetts Institute of Technology
USA

Cathy Mengying Fang*
catfang@media.mit.edu
MIT Media Lab
USA

Han Huang*
huangh14@rpi.edu
Rensselaer Polytechnic Institute
USA

Andrzej Banburski-Fahey
abanburski@microsoft.com
Microsoft
USA

Judith Amores Fernandez
judithamores@microsoft.com
Microsoft
USA

Jaron Lanier
jalani@microsoft.com
Microsoft
USA

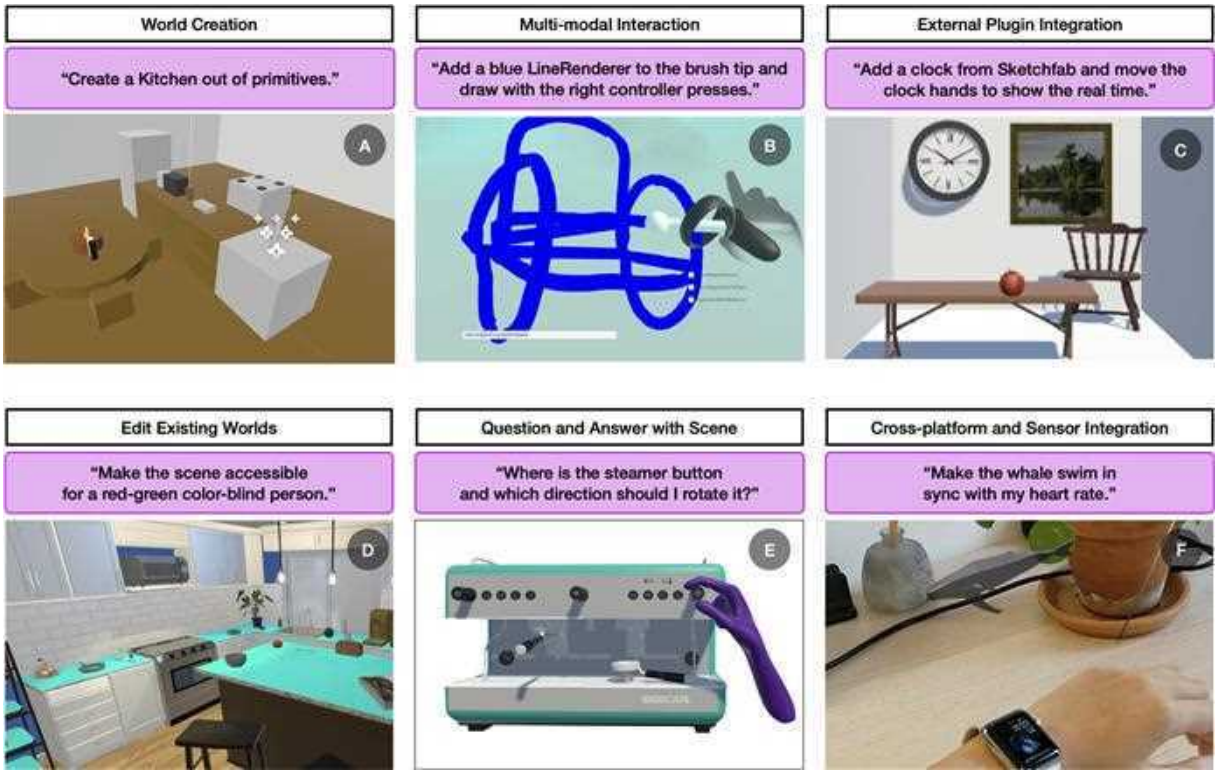

Figure 1: Examples of diverse use cases and functionalities enabled by the Large Language Model for Mixed Reality (*LLMR*) framework. A: Creation of a detailed kitchen scene from scratch using Unity primitives. B: Prompting and drawing objects into existence via multi-modal interactions. C: Integration with external plugins like loading objects from Sketchfab to create high-fidelity scenes and special skills like generating animations. D: Prompting edits of existing VR scenes like changing the color of the objects. E: Automated generation of instructional guides and Questioning and Answering about the scene. F: The framework is compatible across platforms and supports the integration of external sensor data.

ABSTRACT

We present Large Language Model for Mixed Reality (LLMR), a framework for the real-time creation and modification of interactive Mixed Reality experiences using LLMs. LLMR leverages novel strategies to tackle difficult cases where ideal training data is scarce,

*Authors contributed equally to this research and were affiliated with Microsoft.

CHI '24, May 11–16, 2024, Honolulu, HI, USA

ACM ISBN 979-8-4007-0330-0/24/05
https://doi.org/10.1145/3613904.3642579

or where the design goal requires the synthesis of internal dynamics, intuitive analysis, or advanced interactivity. Our framework relies on text interaction and the Unity game engine. By incorporating techniques for scene understanding, task planning, self-debugging, and memory management, LLMR outperforms the standard GPT-4 by 4x in average error rate. We demonstrate LLMR's cross-platform interoperability with several example worlds, and evaluate it on a variety of creation and modification tasks to show that it can produce and edit diverse objects, tools, and scenes. Finally, we conducted a usability study (N=11) with a diverse set that revealed participants had positive experiences with the system and would use it again.

CCS CONCEPTS

• **Computing methodologies → Spatial and physical reasoning**; **Multi-agent systems**.

KEYWORDS

large language model, mixed reality, spatial reasoning, artificial intelligence

ACM Reference Format:

Fernanda De La Torre, Cathy Mengying Fang, Han Huang, Andrzej Banburski-Fahey, Judith Amores Fernandez, and Jaron Lanier. 2024. 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), May 11–16, 2024, Honolulu, HI, USA.* ACM, New York, NY, USA, 22 pages. https://doi.org/10.1145/3613904.3642579

1 INTRODUCTION

Creating 3D virtual worlds is a challenging task that requires both artistic and technical skills. In addition, 3D content often becomes deprecated and has limited interoperability due to platform and device upgrades. Recently, generative AI models have made considerable progress in producing meshes for objects and scenes [17, 18, 22, 25, 26, 41, 43]. However, few works have ventured beyond visual appearances to bring e.g., interactive and behavioral elements into the generated content. In addition, existing rendering-based methods require substantial compute and time to generate and render 3D objects, while the quality and resolution of these generations are limited [11, 35].

On the other hand, the rapid advancement in Large Language Models (LLM) like GPT has shown promise in code generation and reasoning [1, 6, 14, 21, 33]. An integration of LLMs with a game engine, like Unity [50], can enable faster 3D content development and spontaneous user creation, a core element of mixed reality since its inception. In addition, the 3D mixed reality worlds offer rich, spatial, multimodal information (most are post-symbolic or beyond language) that can potentially help LLMs to better situate their reasoning in the reality that humans live in.

This paper presents LLMR(Large Language Models for Mixed Reality), a framework that enables real-time creation and modification of interactive 3D scenes. LLMR can create objects that are rich in both visual and behavioral aspects, or make spontaneous and bespoke edits on an existing environment. For example, we leverage LLMR to spawn interactive tools that are self-contained units designed to perform specific functions in virtual and mixed-reality environments. They can be combined to form more complex interactive systems, extending the range and depth of user and AI-driven experiences. These configurations can be saved and transferred across various environments, serving as the building blocks for versatile interactive experiences.

LLMR is an orchestration of an ensemble of specialized GPTs. At its center is the *Builder* GPT serving as an architect of C# Unity code for crafting interactive scenes. However, the multitude of tasks falling under virtual world creation renders a standalone coder insufficient. For instance, the ability to meaningfully modify an existing virtual world necessitates a profound semantic understanding of the scene. As humans, we have the ability to infer the properties of objects in the world and can refer to objects in the environment using demonstratives. To simulate the benefits of perceptual access, we incorporated the *Scene Analyzer* GPT. It generates a comprehensive summary of scene objects, offering detailed information when requested, including aspects like size, color, and the functionalities of interactive tools previously generated by LLMR. We also implemented the *Skill Library* GPT that determines the relevant skills that are needed for the *Builder* to accomplish the user's request. In addition, we have observed that the code generated by the *Builder* lacks robustness and frequently contains bugs. To remedy this, we introduce the *Inspector* GPT, which evaluates the *Builder*'s code against a predefined set of rules. This evaluation acts as a protective measure against compilation and run-time errors before the code is executed via the *Compiler* in the Unity Game Engine.

To illustrate the efficacy of our framework in the creation and editing of virtual scenes, we tested LLMR on two sets of 150 prompts encompassing a wide array of creation and modification tasks. Our findings demonstrate LLMR's superior performance in contrast to general-purpose LLMs while emphasizing the performance gain achieved with the addition of each module in our pipeline. In particular, LLMR exhibits 4x reduction in code errors in both an empty and an existing scene, when compared to off-the-shelf GPT-4 [34]. In the meantime, LLMR can successfully complete sequences of tasks with varying complexities, while keeping the completion time around a minute. These outcomes underscore LLMR's capacity to execute user instructions in real time with a higher degree of robustness.

To evaluate if our framework can generate not only functional code but also interactive worlds that meet users' instructions, we evaluated LLMR with 11 participants with varying Unity experiences. At a high level, participants found LLMR to be intuitive and easy to use, and they were able to iteratively achieve desired outputs without much manual scripting. While the framework has limitations such as its unpredictability due to generative models' stochastic nature, and thus is not applicable for all contexts (especially ones that require precise and specific control), the output generated by LLMR serves as a starting point for more complex scene generation.

Our paper is organized as follows: we begin by describing prior work and approaches to generating 3D objects and environments for mixed reality in Section 2. In Section 3, we first provide an overview of LLMR followed by details of the function of each module of our framework. We then discuss important extensions of our framework, such as incorporating plugins, memory management, and cross-platform compatibility, in Sections 4,5,6, respectively. We then

present a series of exemplar applications in Section 7 to illustrate the wide range of creations enabled by LLMR. Section 8 Numerical Study includes a comprehensive evaluation of our framework against our design goals: high completion rate, real-time execution, robust against complex tasks, and iterative fine-tuning ability. We follow the Numerical Study with Section 9 User study that evaluates the quality of LLMR's output and presents usability feedback. Finally, in Section 10, we discuss the limitations and future work for others to build upon.

In summary, our main contributions are the following:

(1) We introduced a versatile framework for real-time generation of interactive 3D objects and scenes using LLM modules, designed for easy setup with an OpenAI API key and adaptable across various mixed reality tools, environments, and devices.
(2) We carried out extensive evaluations, including a technical ablation study to gauge the framework's performance and reliability, and a user study to derive design recommendations for optimizing the user experience.
(3) We showcased the expanded capabilities of GPT beyond text inputs, illuminating the broader potential of LLM applications, and demonstrated the framework's broad applicability in domains such as remote training, creativity, and accessibility.
(4) We advocate for the interoperability and longevity of mixed reality applications enabled by AI, and thus we openly share the installation package, code, and prompts used in our application and evaluation so that future work can build on top of our framework.

2 RELATED WORK

Our research on the creation and modification of interactive 3D scenes using natural language is situated at the intersection of large language models (LLMs) and 3D content generation. This section provides an overview of the related work in these areas, highlighting how our work builds upon and extends existing research.

2.1 Generative 3D Assets

The generation of 3D assets has been a significant focus in recent research. The work of Li et al. with 3DDesigner [25], Jun and Nichol with Shap·E [22], and Poole et al. with DreamFusion [35] have demonstrated the potential of text guidance and generative models in creating complex and diverse 3D objects. Lin et al. introduce Magic3D [26], a high-resolution text-to-3D content creation framework that addresses the limitations of slow optimization and low-resolution output inherent in existing methods like DreamFusion. Recently, Holodiffusion by Karnewar et al. [23] furthered the conversation by employing diffusion models for 3D generative modeling. The Instruct-NeRF2NeRF method [15] and advancements like Pointclip v2 [65] as well as the work of Roberts et al. [39] have explored the power of prompting techniques in 3D open-world learning. A comprehensive review of Neural Radiance Field (NeRF) models by Gao et al. [11] adds to our understanding of this rapidly growing field and aligns with our approach of enabling LLMs to interpret non-linguistic or non-symbolic information. Our approach extends beyond visual appearances to incorporate interactive and behavioral elements into the generated content.

2.2 Generative Interactive 3D Environments

In addition to generating objects, the creation of interactive 3D environments has been further explored, with contributions from Wang et al. with Voyager [53], Singer et al. with MAV3D [41], and Höllein, Lukas, et al. with Text2Room [17]. Volum et al. has shown that LLMs can be used to guide NPC interactions with a virtual environment [52]. Wang et al. also introduced Chat-3D [56], a system that focuses on universal dialogues for 3D scenes, which is further augmented by the work of Hong et al. with 3D-LLM [18]. New approaches like Oasis [43] and Procedurally Generated Virtual Reality [44] add novel perspectives. Recent advancements such as Interactive Example-Based Terrain Authoring with Conditional Generative Adversarial Networks by Guérin et al. [13] add a layer of complexity to how terrains can be generated from simple user inputs. Research by Freiknecht and Effelsberg [10], Cao et al. [4], and Song et al. [42] has focused on the balance between realism and algorithmic performance. DeepSpace introduced a novel method of mood-based texture generation from music [45], adding another layer of complexity to asset generation. While these contributions are significant in building interactive 3D spaces, the interplay between AI and mixed reality in these environments remains an open question. Our work tackles this gap by bringing the capabilities of LLMs to a real-time Unity editor for Mixed Reality applications.

2.3 Editor Support for Mixed Reality Development

Mixed Reality (XR) development has been explored by Hirzle et al. [16] and Fidalgo et al. [9], who provide comprehensive reviews at the intersection of AI and XR. Lindlbauer et al. [27] and Cheng et al. [5] focus on the automatic adaptation of MR interfaces, a line of work that is relevant for multi-user XR experiences, as shown by Mandi et al. with RoCo [30]. Thoravi Kumaravel et al. [51] complement these efforts by focusing on bi-directional mixed-reality telepresence. Compared with prior work, we allow users to directly authorize the environment using natural language.

2.4 LLMs Interpreting Spatial, Non-Linguistic Information

Lastly, many have pushed the boundary of LLMs by inputting non-linguistic information (which was not in the training set), such as for visual programming [64] or processing sensor data [28]. More related to our work is using LLM to interface with spatial, embodied data. Work of Zhang et al. with MotionGPT [60], Wu et al.'s work on Embodied Task Planning [57] as well as Richardson et al. with TEXTure [38]. Daffara et al. [7] and Rana et al. [37] further extended these concepts to include demonstrations and task planning. Driess et al. with PaLM-E [8] has shown the potential of LLMs in generating human motion, texturing 3D shapes, and incorporating real-world sensor modalities, respectively. These efforts are complemented by Xu et al. with XAIR [58], which focuses on explainable AI in augmented reality. We hope LLMR contribute to the improvement of LLM's capability of spatial reasoning and world understanding.

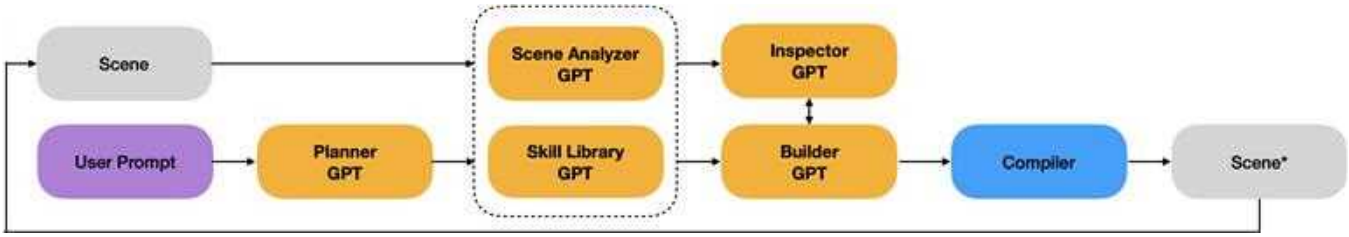

Figure 2: *Large Language Model for Mixed Reality (LLMR) architecture for real-time interactive 3D scene generation.* Starting from the left, a user prompt and the existing 3D scene (Ω) are fed into the *Planner* (P) and *Scene Analyzer* (SA) modules, respectively. The *Planner* decomposes the user prompt into a sequence of sub-prompts, while the *SA* summarizes the current scene elements. These are then integrated with a *Skill Library* (SL) to guide the *Builder* (B) module, which generates the appropriate code. The *Inspector* (I) module iteratively checks the generated code for compilation and run-time errors. Upon receiving the green light from the *Inspector*, the code is compiled using the Roslyn Compiler and executed in the Unity Engine to produce the desired 3D scene and functionalities as specified by the user.

3 LLMR: A FRAMEWORK FOR GENERATING REAL-TIME, INTERACTIVE 3D WORLDS USING LARGE LANGUAGE MODELS

Large language models are capable code generators, and their ability to synthesize programs has been extensively tested [1, 6, 14, 21, 33]. Scripting in a game engine, however, is especially challenging given the multitude of tasks and the complexity of the development environment. For a non-comprehensive list, generating a realistic 3D world may involve object creation, texturing, behavior programming, event scripting, animations, particle effects, lighting, and user interface [3]. Prompting these elements in real time requires a framework that understands the virtual scene, interprets user intention, and generates high-quality code. To this end, we present Large Language Model for Mixed Reality (*LLMR*), a framework that enables real-time creation and modification of interactive 3D scenes using natural language.

LLMR is an orchestration of language models, each contextualized with a distinct metaprompt to outline its role, as illustrated in Figure 2 and Algorithm 1. A metaprompt is a specially crafted input sequence or context that guides an LLM's behavior or output, enabling more focused or nuanced responses than standard prompts. We start with the *Planner*, which breaks down the user's request into a sequence of appropriately scoped instructions. These instructions, along with a concise summary of the existing scene from the *Scene Analyzer* and extra knowledge for specialized skills from the *Skill Library*, are used as inputs to the central module called *Builder*, which generates code to fulfill these instructions. In addition, we use a separate *Inspector* module to check the *Builder*'s generated code against potential compilation and run-time errors before finally executing the code.

The task of generating interactive 3D scenes boils down to generating and executing appropriate code snippets to accomplish the user's prompt. Formally, denote the user's request by u and the current 3D world by Ω (which may be empty), we wish to draw sample $x \sim \mathcal{P}(x|u, \Omega)$, where $\mathcal{P}$ is the distribution of syntactically valid, request-fulfilling code. We then compile and execute x at run-time under the Unity Engine [50], a development platform for creating virtual scenes that suits our needs. Below, we detail each module and explain the design choices that enable various aspects of prompting a virtual world into existence.

Algorithm 1: LLMR

Input : u: user's request; Ω: current scene
Require: $\mathrm{A}(s|u, \Omega)$: Scene Analyzer;
$\mathrm{P}(u_1, ..., u_N|u, s)$: Planner;
$\mathrm{L}(h|u)$: Skill Library;
$\mathrm{B}(x|u, s, h)$: Builder;
$\mathrm{I}(r, v|u, s, x)$: Inspector.

$s \sim \mathrm{A}(\cdot|u, \Omega)$;
$(u_1, ..., u_N) \sim \mathrm{P}(\cdot|u, s)$ `/* Decomposes the request into suitable instructions. */`
$\Omega_1 \leftarrow \Omega$;
for $i = 1 : N$ **do**
  $s_i \sim \mathrm{A}(\cdot|u_i, \Omega_i)$ `/* Analyze the current scene. */`
  $h_i \sim \mathrm{L}(\cdot|u_i)$ `/* Retrieve required skills. */`
  $x_i \leftarrow \mathrm{GenerateCodeWithInspection}(u_i, s_i, h_i)$;
  $\Omega_{i+1} \leftarrow \mathrm{CompileAndRun}(x_i, \Omega_i)$;
end

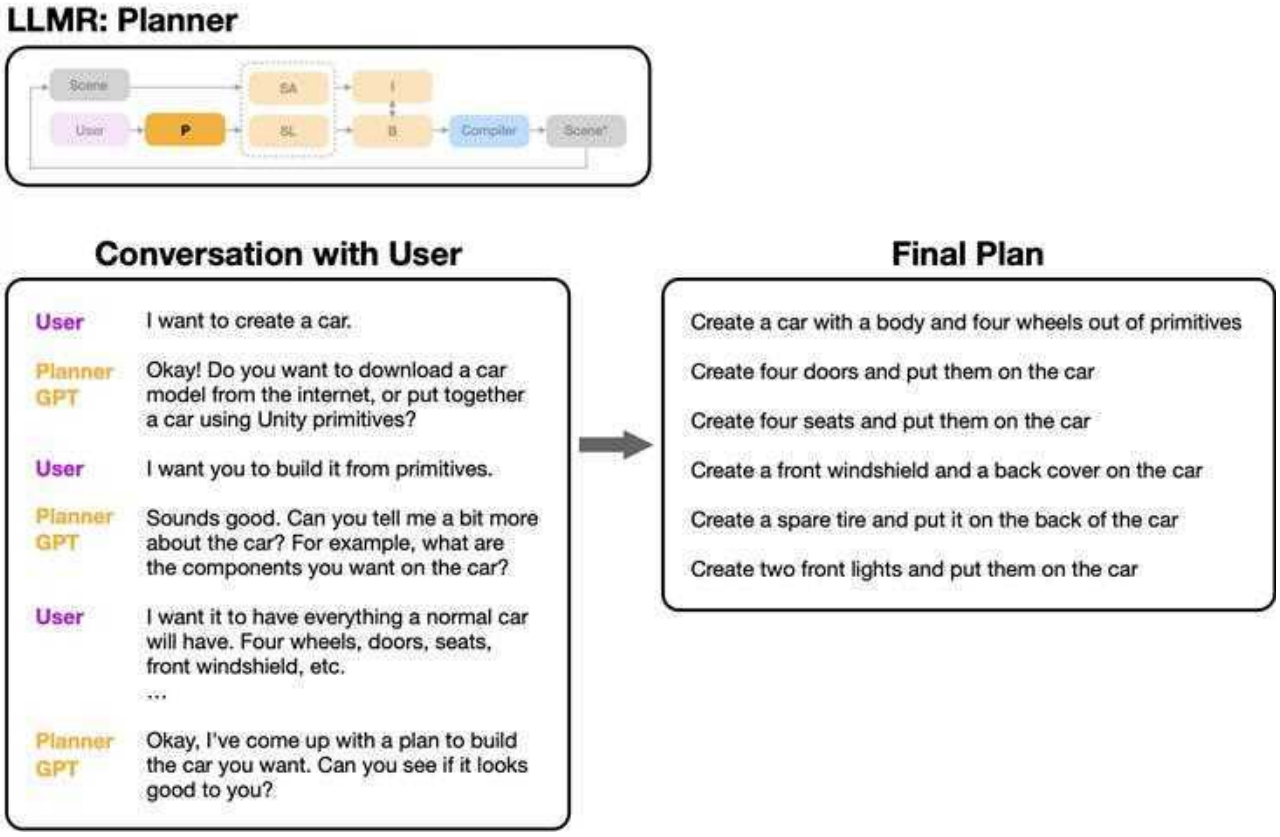

Figure 3: The *Planner and its role in breaking down a user's high-level request into a sequence of manageable subtasks* $(u_1, u_2, \ldots, u_n)$. The *Planner* engages in a user-oriented conversation to determine the appropriate scope and granularity of each subtask. Following this, the *Builder* executes the plan by generating code $(x_1, x_2, \ldots x_n)$ for each subtask, effectively carrying out the user's initial request.

3.1 Planner

Prompting a world into existence can be a hefty task. "Create a city and all its denizens" is a valid request, albeit one that is overly ambitious to achieve in a single step. Following the common wisdom "nothing is particularly hard if broken into small jobs", instead of directly sampling from $\mathcal{P}(x|, u, \Omega)$, we propose a *Planner* $P : u \mapsto (u_1, u_2, ..., u_N)$ to decompose each prompt into subtasks within an appropriate scope, then use autoregressive sampling to carry out these subtasks via a sequence of generated code $(x_1, x_2, ..., x_N)$:

$$\begin{aligned}\mathcal{P}(x_1, x_2, ..., x_N|u_1, u_2, ..., u_N, \Omega) &= \mathcal{P}(x_1|u_1, u_2, ..., u_N, \Omega) \times \\ \times \prod_{n=1}^{N-1} \mathcal{P}(x_{n+1}|\boldsymbol{x}^n, u_1, u_2, ..., u_N, \Omega) \\ = \mathcal{P}(x_1|u_1, \Omega) \prod_{n=1}^{N-1} \mathcal{P}(x_{n+1}|\boldsymbol{x}^n, u_{n+1}, \Omega)\end{aligned} \tag{1}$$

where $\boldsymbol{x}^n := (x_1, ..., x_n)$. The second quality follows by assuming independence of code generations and requests at different steps, $x_n \perp\!\!\!\perp u_m, \forall m \neq n$. An illustration for this procedure is provided in Figure 3. However, sampling from $\mathcal{P}(x_{n+1}|\boldsymbol{x}^n, u_{n+1}, \Omega)$ may be difficult for a language model, because it has to *infer* the effect of $(x_1, ..., x_n)$ on the initial world Ω before writing code x_{n+1}. To remove the guesswork, we leverage a runtime compiler R to execute $(x_1, ...x_n)$ in order, each time getting a new world state $\Omega_{n+1} = R(x_{n+1}, \Omega_n)$. We can then rewrite:

$$\mathcal{P}(x_{n+1}|\boldsymbol{x}^n, u_{n+1}, \Omega) = \mathcal{P}(x_{n+1}|x_n, u_{n+1}, \Omega_n), \tag{2}$$

where we assume $\{x_i\}_{i=1}^n$ is Markovian when conditioned on Ω_n. That is, the current world state is rich enough to capture all previous executions past the most recent one.

In principle, it is possible for the user to limit their prompts within a certain difficulty so that the decomposition is unnecessary. However, the user may not know the appropriate task scope a priori (if creating a city is too hard, how about a single house? Or a room in the house?) As a result, having a properly configured *Planner* makes the framework robust to prompts of varying difficulty. In addition, the user may have different levels of details in their prompt. For example, "Creating a car" is a valid request that nevertheless does not specify its appearance or functionality. Here, the *Planner* serves as a conversational assistant that interacts with the user to devise a plan with an appropriate scope and granularity, which significantly improves the user experience.

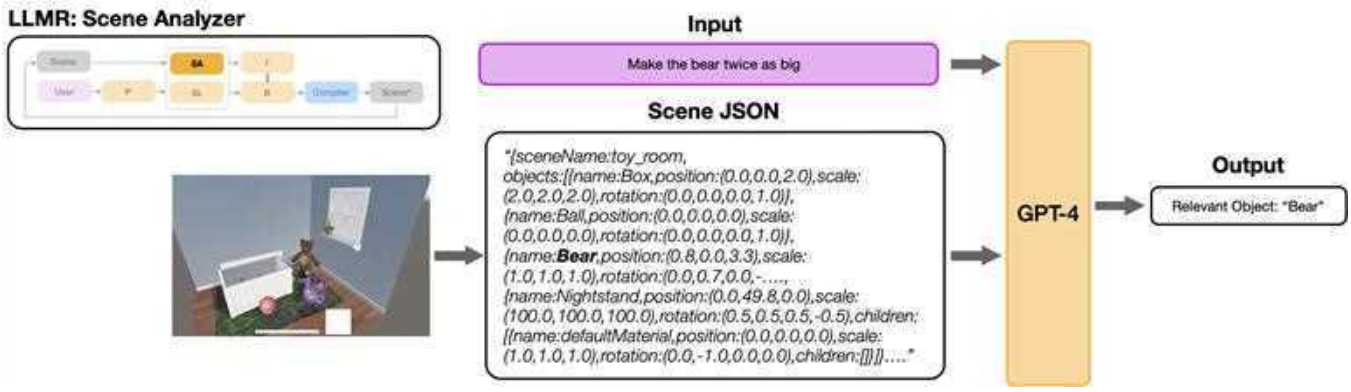

Figure 4: *Scene Analyzer module.* The virtual scene, depicted in the bottom-left corner, is converted into a parsed scene hierarchy in JSON format. This, along with the user request, serves as input to the *Scene Analyzer*. The output is a filtered, relevant summary of the scene, which is then used for conditioning subsequent modules like the *Builder*. The process optimizes the utilization of the language model's fixed context window and enhances focus on objects relevant to the user prompt.

3.2 Scene Analyzer

There are many possible representations of a virtual world Ω that may include visual, behavioral, and auditory elements. In this work, we derive Ω from the Unity scene hierarchy, which contains all existing game objects, their attached components, and their parent-child relations. The hierarchy is parsed into a JSON string and can then be used as input to language models. However, directly using the raw JSON string as input proves to be infeasible in practice. First, most prompts only require interactions with a small subset of Ω, so it is unnecessary and even distracting to use its entirety as input. Second, LLMs have a fixed context window W that serves as its short-term memory, which has to contain its metaprompt, few-shot examples, user prompt, and generative output [62]. For example, GPT-4 supports either 8k or 32k tokens for maximum number of token at a time[34], but even the 32k token limit can be insufficient, particularly for intricate scenes containing numerous objects, each consisting of multiple components.

To tackle these issues, we created a separate module termed the *Scene Analyzer*, which is a properly prompted LLM $\mathrm{A}(s|u, \Omega)$ that outputs a succinct summary of Ω conditional on the user request. At a high level, one can think of the *Scene Analyzer* as a means of perception that relays an abstraction of the environment for downstream processing. An illustration of the module is provided in Figure 4. Concretely, the output $s_n \sim \mathrm{A}(\cdot|u, \Omega_n)$ is used to reparametrize the density at each sampling step:

$$\mathcal{P}(x_{n+1}|x_n, u_{n+1}, s_n) \sim \mathcal{P}(x_{n+1}|x_n, u_{n+1}, \Omega_n) \tag{3}$$

3.3 Builder-Inspector

Central to LLMR is the *Builder* $\mathrm{B}(x|u, s)$, a module responsible for generating code conditional on the user prompt. It serves as our main apparatus for approximating $\mathcal{P}$. In other words, we hope

$$\mathrm{B}(x|x_n, u_{n+1}, s_n) \approx \mathcal{P}(x_{n+1}|x_n, u_{n+1}, s_n), \tag{4}$$

holds with a carefully crafted metaprompt and enough in-context demonstrations. In practice, however, the complex nature of creating a virtual world makes the approximation unsatisfactory even with as many examples as the context length allows. This is largely because the *Builder* module is asked to accomplish the instructions with some creativity while faithfully following an extensive list of specific guidelines that align the output, which causes to *Builder* to have a "cognitive overload".

Algorithm 2: Generate Code With Inspection

Input : u: user's request, s: scene summary, h: additional hint.
Require: $\mathrm{B}(x|u, s, h)$: Builder;
$\mathrm{I}(r, v|u, s, x)$: Inspector;
T: maximum number of inspections.

$t \leftarrow 0$;
$r_0 \leftarrow \emptyset$;
$v_0 \leftarrow$ False;
while $t < T$ *and* v_t *is false* **do**
  $x_t \sim \mathrm{B}(\cdot|u, s, h, r_t)$ /* Builder writes code x_t */
  $(r_t, v_t) \sim \mathrm{I}(\cdot|s, x)$ /* Inspector checks code, outputs verdict v_t and suggestion r_t */
  $t \leftarrow t + 1$;
end
return x

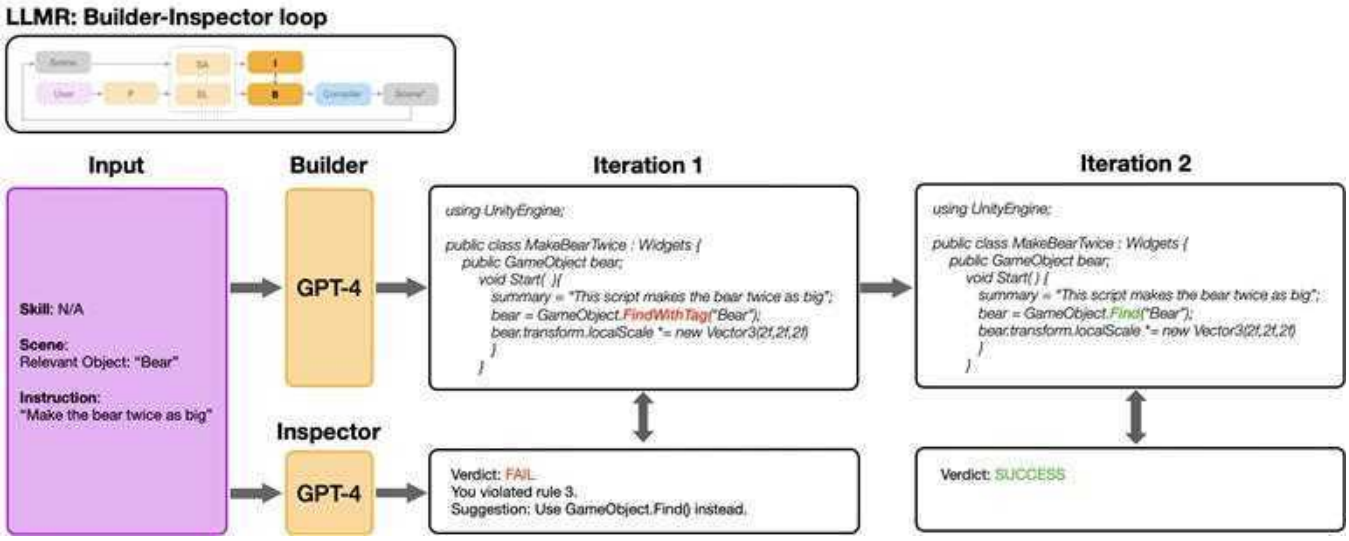

Figure 5: *Builder-Inspector paradigm in LLMR.* The *Builder* module $\mathrm{B}(x|u,s)$ generates code based on user input and current state. The generated code is then inspected by the *Inspector* module $\mathrm{I}(r,v|x,s)$ for compilation and run-time errors. If errors are found, indicated by verdict v, the *Inspector* provides suggestions r for corrections. The process iterates until either the code passes inspection or a maximum number of inspections T is reached. This feedback loop significantly enhances the quality of the generated scripts.

To ameliorate this, we introduce another module, the *Inspector* $\mathrm{I}(r,v|x,s)$, that checks the *Builder*'s generated code for compilation and run-time errors. In the case of a failed inspection indicated by verdict v, the *Inspector* outputs a suggestion r for potential fixes and prompts the *Builder* to make another attempt. As a result, the *Builder* and *Inspector* work in tandem to write and self-debug code, forming a feedback system that significantly improves the quality of the generated scripts. We outline this paradigm in Algorithm 2 and illustrate it in Figure 5. Interestingly, the *Inspector* excels at catching errors even if the same guidelines in its metaprompt are present in the *Builder*. One possibility is that this is due to providing a more extensive list of negative and positive examples to the *Inspector*. Still, when the *Builder* is provided with the same examples, performance is not as high. Our intuition for this is that verifying a snippet of code is easier than writing the said code, or the two tasks bear different failure modes that can be effectively hedged.

3.4 Compilation, Save and Reload

After the *Builder*-generated script passes the inspection, we follow the approach in [39] to compile and execute the scripts at runtime through the Roslyn C# compiler [49]. The inclusion of run-time compilation elevates LLMR from an offline development tool to a real-time generative framework.

To enable iterative design, users can save their generations and selectively reload the saved generations in the existing or new scene without having to repeat the prompting process. The generated output is saved as C# scripts and reattached to the Compiler to be compiled at runtime. A one-sentence summary of each script's function is saved, so alternatively, the output can also be regenerated by the framework based on the summary.

3.5 Skill Library

The creation of the *Skill Library* Module is motivated by two primary challenges. The first is the token size limitation imposed by the GPT architecture on the context, or the "metaprompt," provided to the *Builder*. Typically, the *Builder* is presented with a comprehensive list of various APIs and plug-ins that could be employed to meet the user's needs. As the range of available skills expands, this list lengthens, eventually surpassing GPT's token size limit for public users. The second challenge lies in the *Builder*'s attention capacity, which appears to be limited. Even when we attempt to condense all the available skills into the *Builder*'s metaprompt, it struggles to keep track of a specific skill when the list becomes too lengthy. This limitation is further exacerbated by the necessity to include precise coding examples for each plugin to ensure their effective utilization by GPT. To address these challenges, we created the *Skill Library* module, denoted as $\mathrm{L}(h|u)$, which serves as a centralized repository for all available skills and as an attention mechanism that retrieves only the skills relevant to a specific user prompt. We illustrate this module in Figure 6.

Formally, a specialized GPT is provided with a metaprompt containing two essential pieces of information: 1) a high-level summary of the available skills, and 2) the user's prompt. The GPT model is tasked with identifying either a single skill or a subset of skills that are most pertinent to the user's request. The *Skill Library* remains efficient and small in token size because it only needs the high-level descriptions of each skill, while the specific usage details,

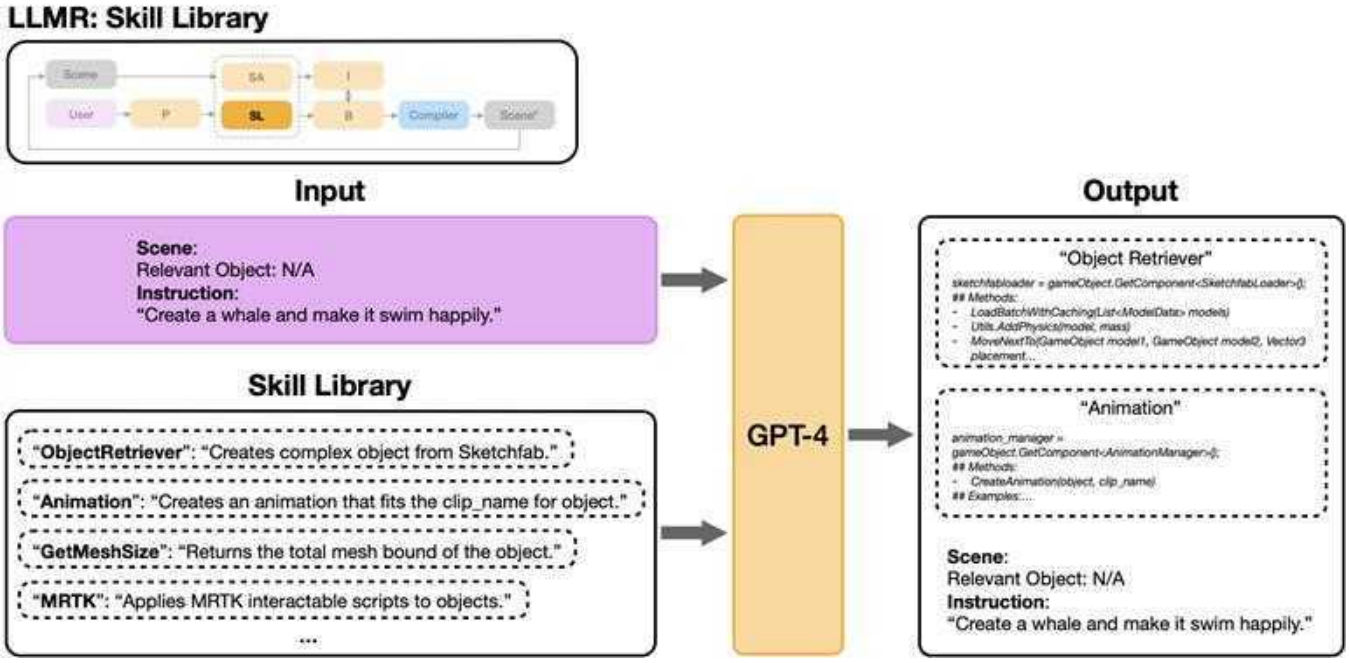

Figure 6: *Skill Library module workflow.* On the left, the module receives inputs from the *Scene Analyzer* and a user prompt "create a whale and make it swim happily". A list of skills is provided to the SL GPT module in its metaprompt, which also contains a high-level summary of available skills such as object retrieval and animation. The module then identifies and outputs the most relevant skills (in this case, object retriever and animation) to the *Builder*, which subsequently utilizes these tools for implementation.

as well as positive and negative examples, are stored separately. Once the relevant skills are identified, their detailed information and usage examples are fetched and passed on to the *Builder* for implementation.

$$h_i \sim L(\cdot|u_i) \quad \text{(Retrieve required skills, if any.)} \tag{5}$$

As an illustrative example, consider a skill we created for GPT's use, which leverages a combination of generative and contrastive models along with the Sketchfab API to source and integrate 3D models into a scene. We have also created skills that allow the generation of animation of a rigged object in real-time [19]. While we delve into the specifics of a skill in the next section, it is worth noting that the *Skill Library* only receives a high-level summary of how this particular skill functions, along with similar descriptors for other skills. The actual examples needed to use this skill are then retrieved and supplied to the *Builder* for execution.

$$\mathrm{B}(x|u,s,h) : \text{Builder}; whereh = \text{retrieved skills from} L \tag{6}$$

This approach ensures that the *Skill Library* and the *Builder* work in tandem to efficiently and effectively generate code that fulfills the user's request while overcoming the token size and attention capacity limitations of LLMs.

4 INCORPORATING EXISTING OPEN-SOURCE 3D ASSETS

The process of generating interactive 3D scenes often involves the creation and placement of various objects. For instance, a request to create an office space might be decomposed into the generation of a desk, a chair, a lamp, and a clock. While it is possible to generate these objects using primitives, a method that works well even for composite objects like a car or an entire room (depicted in the car of Figure 8 and the kitchen of Figure 1), there is a need to leverage the intricate objects created by artists and 3D developers that exhibit high real-world fidelity. Previous work has utilized objects from Sketchfab [39, 40] and used the priors of GPT to size them accordingly to the real world. However, this approach encounters challenges when the user prompts an object, say a clock, and Sketchfab offers 50 different clocks, only three of which are suitable for an office setting.

To address this issue, we introduce the *Object Retriever*, a skill that employs other AI models to identify the 3D object that the user most likely intended. The workflow of the *Object Retriever* can be formalized as follows: given a user prompt u, the *Object Retriever* identifies an object o contained in u and calls the Dall·E-2 [47] API for the object o, generating a "target image" T. Concurrently, the same object-prompt o is used to download N screenshots of 3D objects freely available on Sketchfab, denoted as $\boldsymbol{S} = \{s_1, s_2, ..., s_N\}$. We then employ CLIP [36] to map out similarity spaces in the language domain L and the visual domain V. We select the top

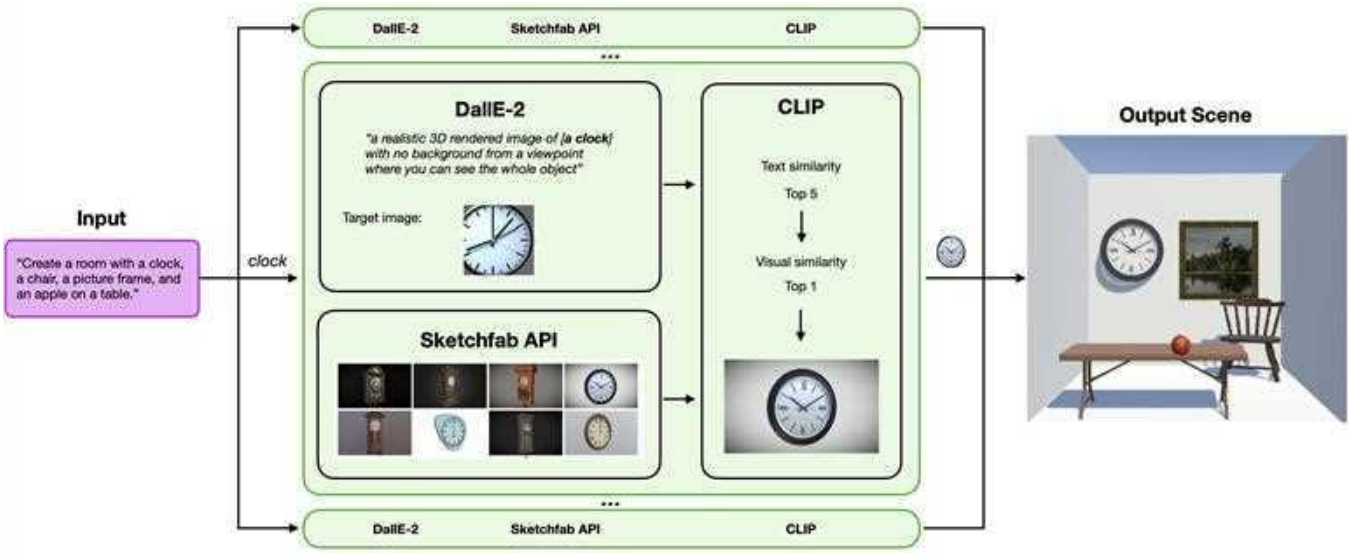

Figure 7: *Object Retriever pipeline for generating a 3D scene.* The user provides a prompt for a scene containing a clock, a picture frame, a chair, and an apple on a table. For each object (e.g., clock), the pipeline uses DALL-E 2 to create a target 3D image. Concurrently, multiple screenshots of potential matches from open-source Sketchfab models are downloaded using the object label as the query. CLIP is employed to generate embeddings for these images which includes the target image. The top 5 candidates in the language similarity space are selected. The final object is then chosen based on the highest visual similarity to the target image. This sequence is repeated for each object in the prompt to assemble the complete 3D scene, as shown on the far right.

5 images $S' \subset S$ that are closest to the object-prompt o in the language similarity space L, and from these, we select the image s^* that is closest to the target image T in the visual similarity space V. Formally, let $L(o, s_i)$ and $V(T, s_i)$ denote the language and visual similarity between the object-prompt o and the screenshot s_i, and the target image T and the screenshot s_i, respectively. The *Object Retriever* operates as follows:

This process is repeated to generate entire scenes. Algorithm 3 and Figure 7 describe this pipeline. There is potential for further exploration to improve this pipeline. For instance, selecting from the visual similarity space before the language similarity space might yield better results. Future work will involve human feedback to identify the workflow that maximizes the likeness between the 3D object loaded and the user's intended object.

Algorithm 3: Retrieving 3D objects

Input : u: user's prompt
Require: o: object in u;
T: target image;
S: screenshots;
$L(o, s_i)$: language similarity;
$V(T, s_i)$: visual similarity.
$S' \leftarrow$ Top $5 s_i \in S$ with highest $L(o, s_i)$;
$s^* \leftarrow \arg\max_{s_i \in S'} V(T, s_i)$;
return s^*

5 MEMORY MANAGEMENT

By default, language models generate new words based on all previously sampled tokens, a configuration that may not be ideal due to their finite context length. For instance, this may hinder the model's ability to engage in extended conversations. To mitigate this, techniques such as dialogue summarization and distillation can be employed [2, 20, 54]. Additional research has delved into leveraging persistent memory and retrieving in-context examples from databases to enhance few-shot performance [55, 63].

We sought to deploy a protocol that alters the contents within the LLM's context window while the framework is in continuous use. We explored three memory modes for each module within LLMR : full memory, limited memory, and memory-less. We document the memory modes used for each module in Table 1. These modes pertain to the retention of all, a few, or none of the historical instructions and generated code within the model's context. Define an episode of interaction as the input and output to the module for

Module	Memory Mode
Planner	Memory-less
Scene Analyzer	Memory-less
Builder	Limited-memory
Inspector	Memory-less
Skill Library	Memory-less

Table 1: Memory mode for each module. Note that no module uses full memory, the default GPT paradigm.

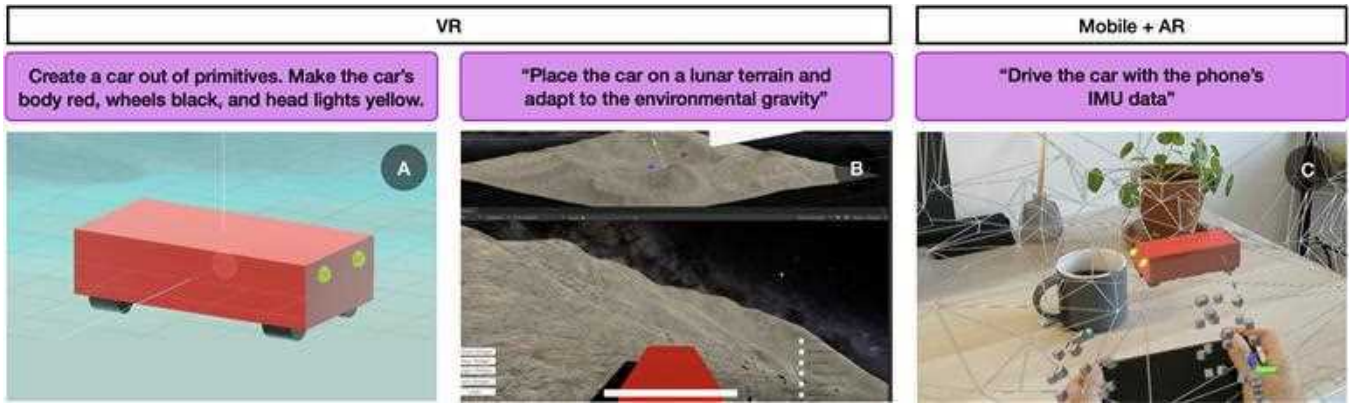

Figure 8: *Cross-Platform and Cross-Scene Transferability made possible by LLMR.* The left panel shows a car automatically created by LLMR using Unity primitives, complete with color and composite features (e.g., wheels and headlights), controllable via keyboard inputs. The middle panel displays the same car transferred to a different Unity scene featuring moon-like gravity and terrain. The right panel showcases the framework's adaptability across platforms by illustrating how the car can collide with objects in the physical world and can be controlled using IMU data from a user's mobile phone.

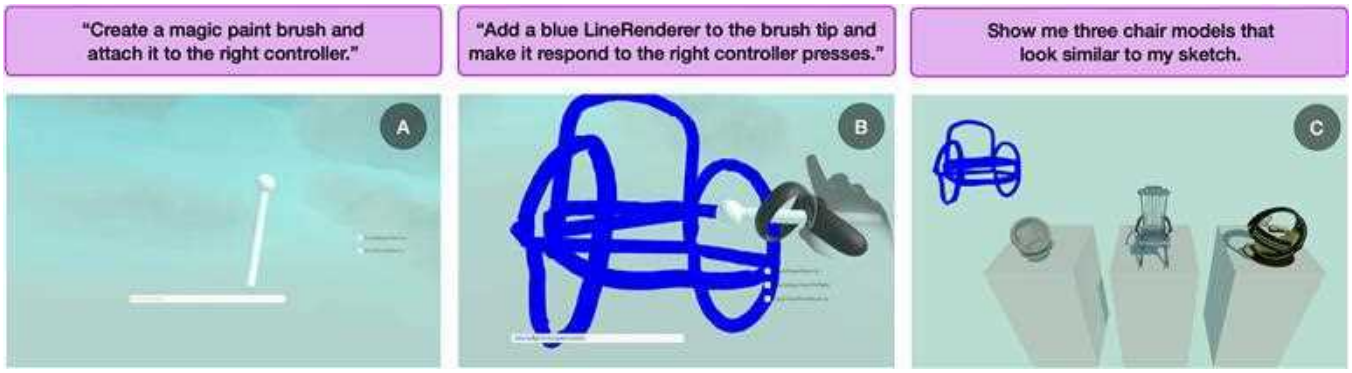

Figure 9: *Sketching objects into existence with LLMR.* In the left panel, a user requests a "magic paintbrush" to be attached to a VR controller. The middle panel illustrates the automatic conversion of the line renderer into a paintbrush, where the user is shown drawing a chair. The right panel demonstrates the 2D-to-3D transformation using 2D-3D ControlNet [59] and our Dall·E-CLIP Sketchfab API. This enables the generation of multiple chair models that can then be transferred across different platforms using LLMR for further interaction.

a single user prompt to LLMR. To implement a memory-limited module, for example, we clear its context of all but the most recent N episodes after every prompt, where $N = 1$ typically.

An effective memory management protocol offers three distinct advantages:

Token limit: Trimming old memory reduces token consumption and enables prolonged usage of LLMR, a critical feature for gradually constructing intricate scenes. Notably, the *Scene Analyzer* benefits from having no memory of prior interactions, as it is susceptible to token constraints. As an example, the first AI2-THOR scene hierarchy measures around 7k GPT-4 tokens [24]. Hence, a full memory *Scene Analyzer* with 8k tokens can only fulfill a single instruction before its context is depleted, rendering the framework essentially unusable outside of a memory-less setting.

Performance: Certain modules perform better with reduced memory, as they may be prone to be confused by earlier interactions. For example, our empirical observations indicate that the *Inspector* module exhibits increased leniency in repeated inspections, allowing the proposed code to pass before all errors are rectified.

Interpretability: A memory-limited framework provides clearer error attribution. For instance, when a sequence of prompts is sent, and the generation fails at the final step, maintaining all memory makes it challenging to discern whether the last prompt posed a unique challenge or if the framework became perplexed by aspects of an earlier task. Improved transparency facilitates swift debugging and iterating on our framework.

We believe the choice of memory mode is a crucial aspect of any LLM orchestration pipeline, and our design choices may offer insights for the development of LLM systems beyond the task of creating virtual worlds.

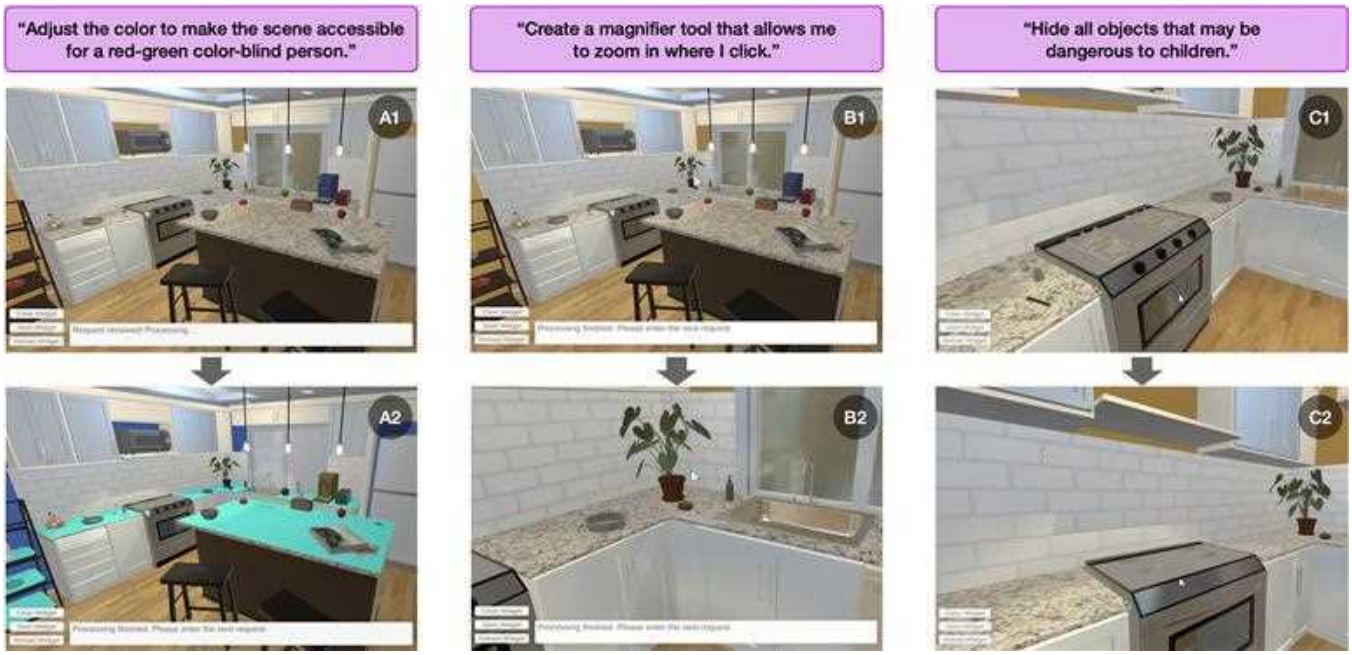

Figure 10: ***Accessible Interface Features in Action.*** **A1 and A2 show how a user can prompt the system to adjust the color scheme of a kitchen scene for red-green color-blind compatibility. B1 and B2 demonstrate the activation of a magnifier tool. C1 and C2 reveal the option to hide objects deemed not kid-friendly.**

6 CROSS-PLATFORM COMPATIBILITY AND INSTALLATION

We show that our framework can be deployed in various types of platforms (e.g., Web, Mobile, AR, and VR) and on various devices (e.g., Meta Quest, HoloLens 2). To keep the framework lightweight, we deploy our framework's run-time compiler on a PC that acts as the server, and we build upon existing remoting protocols and frameworks [32, 46] to stream the generated results to the client device (e.g., holographic remoting for a HoloLens 2). Platform dependencies, such as namespaces and other packages can be added as a "Skill" to the framework's *Skill Library*, which allows the user to quickly enable interaction modalities such as pinch and input modalities like speech and controller.

Interactive elements built within one scene can be saved as self-contained units by storing the source code that created them. We can then re-execute the cached code to load and adapt the prompted objects into novel scenarios, which can be as simple as a different scene with adjusted physics or a project with completely new APIs, as depicted in Figure 8. Our experiments with LLMR suggest that translating interactive elements between independent SDK platforms is possible and suggests an application of adapting existing pieces of software (perhaps ones written with obsolete, no-longer working code) to newer SDKs. We leave this for future explorations.

6.1 Installation

Our framework can be easily added to any existing Unity scenes. The framework consists of a unity package and a few additional open-sourced packages (such as GLTF loader and OpenAI), and the installation process takes only a few steps. This enables anyone with an OpenAI API key to try our framework. We are strong proponents of the adaptability of our framework, and so we have open-sourced the foundational framework along with several examples on GitHub (https://llm4mr.github.io/). Readers who wish to try our framework can try out the example playground scenes or can easily add our Unity package to their existing Unity projects. They would need to obtain an OpenAI API key, a copy of the Roslyn compiler and optionally an account for Sketchfab if they wish to automatically load existing assets. In the Appendix, we also provide the metaprompts used for each LLMR's modules for transparency.

7 EXAMPLE PROMPTED INTERACTIVE WORLDS AND USES

In this section, we illustrate the wide range of objects, tools, and scenes one can construct with LLMR. We highlight that our framework is modular, real-time, adaptive, interactive, and multi-modal, which differentiates this approach from other generated 3D worlds that primarily focus on visual appearance. For all of the examples below, it is important to stress that all of the results are achieved simply by prompting the system, without the need for manual intervention.

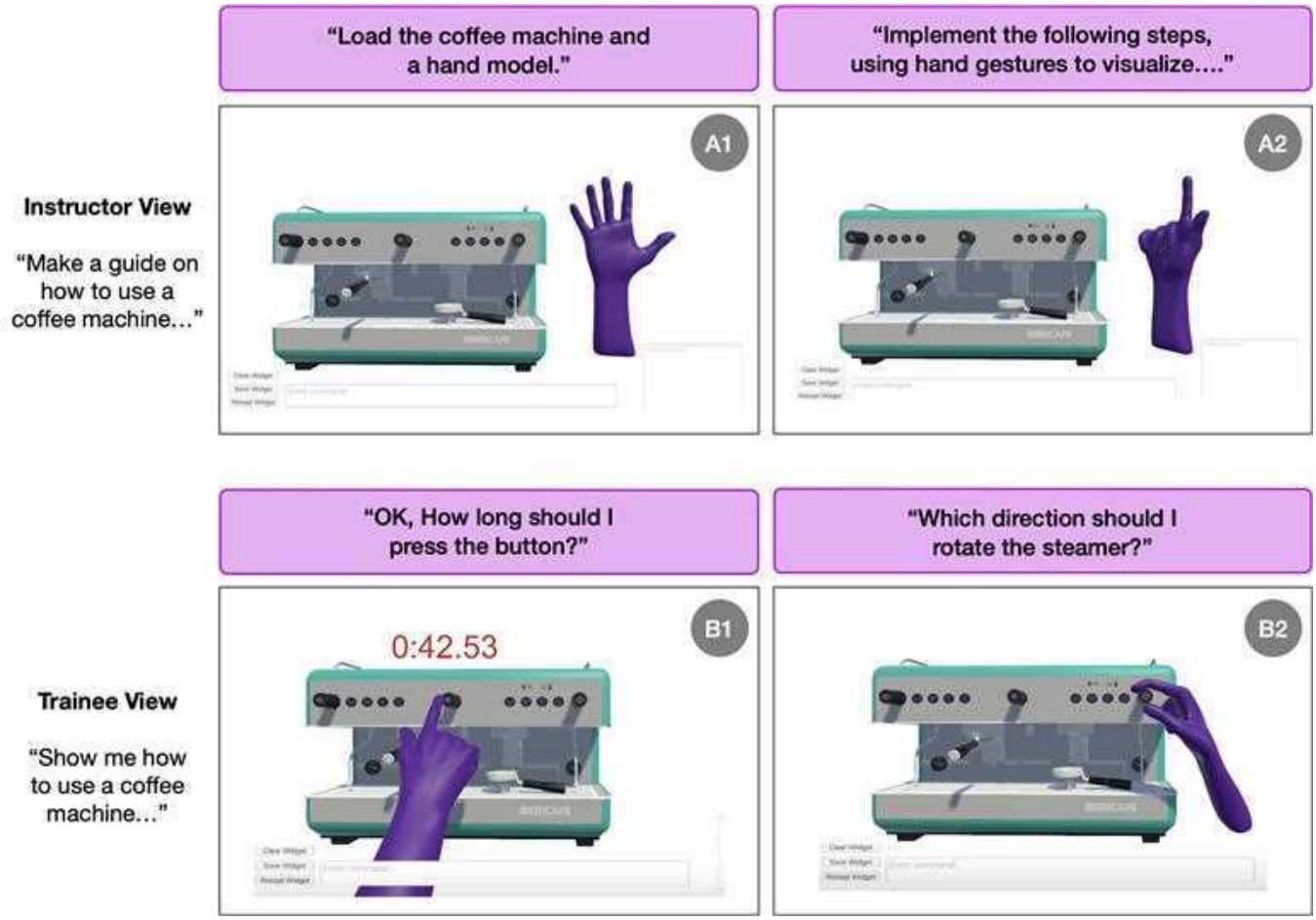

Figure 11: *Spontaneous Creation of Teaching Guides.* A demonstration of creating a guide for operating a coffee machine in which LLMR animates a hand model to point out the various steps of the operation. Our framework allows for the rapid creation of such guides and furthermore allows users to ask questions that were not predicted by the instructor beforehand, with appropriate motions being animated on the fly.

7.1 Game Design and Creativity

An immediate application of our framework is the creation of games, in particular, scenes. A scene sets the context of a game, and it usually involves numerous assets that are difficult and tedious to set up manually. A game designer can use the *Planner* to create a draft environment, and add interactive components like "players" and "opponents" with responsive behaviors to mock up the gameplay logic. In addition, game designers can expand gameplay in multiple environments. For example, a toy car can be created and reloaded in a moon simulation environment in VR (Figure 8 B) or be spawned in the physical world and driven around with a mobile phone (Figure 8 C). Besides "prompting" objects into existence, we show that our framework also allows users to "draw" things into existence. Here the user wishes to design a chair (Figure 9). They can do so by simply prompting "a magic paintbrush", which has functions similar to that of TiltBrush [12], a popular 3D drawing application, and then turn the drawing into a 3D model with the integration of Dall-E 2, CLIP, and Sketchfab, through a similar process illustrated in Figure 7.

7.2 Accessibility and Adaptive Interface

Similar to the accessibility feature in 2D documents, our framework can also be prompted to make a 3D scene accessible and adaptive to different user needs and preferences. Figure 10 shows three examples of editing an existing virtual kitchen scene to different requests. For example, one can request to make the scene to be more friendly to red-green color-blind users. For someone who is near-sighted, they can prompt a magnifier tool that zooms into a particular part of the room. An architect can use our framework to figure out if the space is friendly for wheelchair users or make sure objects in the room are child-proof. These examples show how our framework leverages LLMs' prior knowledge and puts the knowledge into the context of a spatial world at a human scale.

Figure 12: *Simulated Rescue Plan.* The HoloLens 2 displays the automated generation of a simulation of a rescue plan using our framework. The guide shows an interactable 3D terrain, helicopter, and simulated wind, allowing rescue workers to visualize the flight path under different weather conditions.

7.3 Remote Assistance and Planning

In a remote training scenario, typically, creating such a 3D interactive training guide requires custom creation, from rigging a gesture to placing a UI element. An instructor can use our framework to automate the generation of a training guide from a list of instructions. (Figure 11). The trainee can then, for example, use an AR device that overlays information on the machine. As the trainee advances through the steps, they can ask questions directly to the guide where answers can be generated in the context of the trainee's learning progress. In another scenario of remote rescue planning, helicopter operators can prompt a simulation of the flight path given several target locations and see how the flight path might be affected by different wind conditions (Figure 12).

8 NUMERICAL STUDY

As an orchestrated pipeline, LLMR augments an LLM coder with multiple modules to enhance its reliability. To empirically justify the inclusion of each module in our framework, we quantitatively evaluate LLMR's generative performance against a variety of prompts and baselines. In addition to success in compiling the generated code, we evaluate how our framework meets our design goals: real-timeness, complexity of interaction, and iterative fine-tuning ability.

This section is organized as follows: we begin by evaluating LLMR on single prompts in an empty and existing scene, highlighting the impact of each module and overall performance compared to standard LLMs. We also discuss the framework's performance at completing tasks with different complexity. Then, we conduct a similar experiment on sequential prompts to illustrate LLMR's capacity for iterative designs. Lastly, we present an analysis of the real-time aspect of our framework.

8.1 Error Rate

8.1.1 Experiment Setup. We start by investigating LLMR's ability to carry out single, independent requests in either an empty or existing scene. To this end, we created two datasets each with 150 prompts. The first set is used as inputs in an empty scene and is mainly creative in nature as there is nothing to modify or interact with in the world. An example is "creating a cat and mouse out of primitives. The cat should chase the mouse, who flees in an erratic pattern." The second set is used as inputs in an *existing* scene shown in Figure 13. The scene was downloaded from Sketchfab [48] and was chosen as it is sufficiently complex (around 35 objects). A few example prompts are shown in Figure 13, which involve visual and semantic alteration of the space. To promote fairness and diversity in our test prompts, we use a separate, properly prompted GPT to generate two evaluation datasets. The authors created 15 prompts as demonstrations for the prompting GPT. The full evaluation datasets can be found in **Appendix**.

To assess the efficacy of LLMR, a proper metric is required. Given the subjective nature of tasks such as "make the room more uplifting," it is difficult to systematically determine if a prompt has been met successfully. However, the presence of run-time or compilation errors in the generated code can be considered a clear indicator of failure. Therefore, we have selected the 'error rate' – the proportion of outputs with bugs – as the criterion for assessing the framework's performance.

To evaluate the efficacy of each module of LLMR, we created three model conditions, each with adding one additional GPT module, besides GPT-4 zero shot and GPT-4 few shot as our baselines. This makes a total of 5 model conditions. We conducted 5 runs of 150 prompts with each model condition and for each scene condition (empty scene and existing scene).

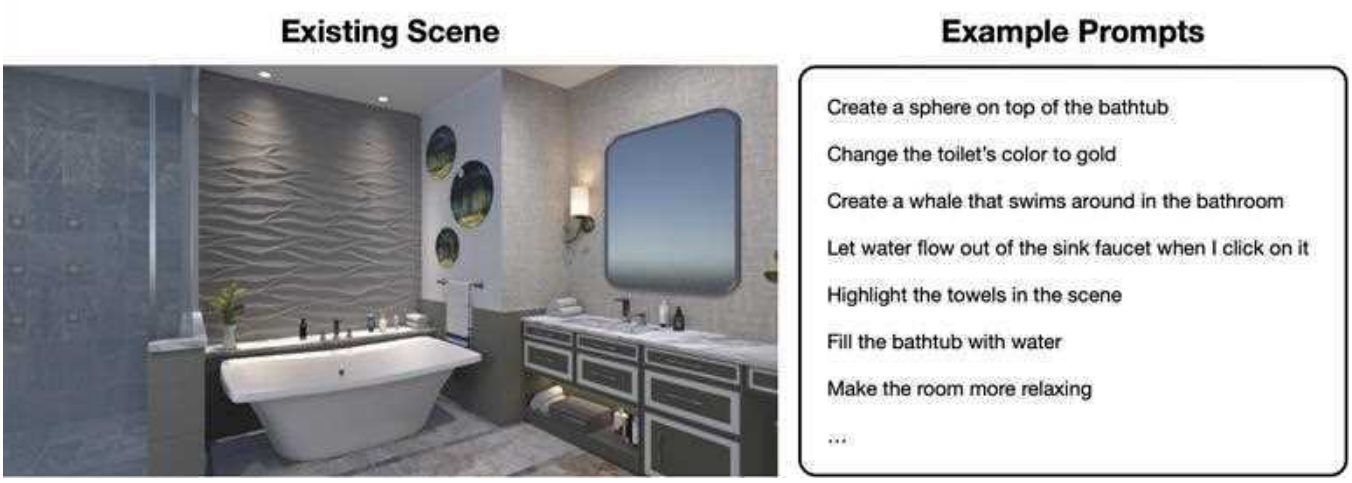

Figure 13: *An illustration of our experimental setup.* We provide the bathroom scene (left) and a subset of the 150 prompts (right) used in this space for the evaluation provided in Figure 14.

8.1.2 Results and Discussion. We provide a summary of error rates for our model and various baselines in Figure 14. To underscore the benefit of each LLMR module, we add each component incrementally to tease out its marginal impact. Starting with the off-the-shelf GPT-4, we see that standard in-context learning techniques increase performance in both settings, yet only to the extent that roughly half of the requests fail. From here, we augment the standard GPT-4 with components developed in this work, starting with the *Scene Analyzer*, then the *Skill Library*, and finally the *Inspector*. As a result, the generated errors drop substantially to only 20.5% and 25.2% of the error rate observed in the original GPT-4 for the empty and existing scene, respectively, which attests to the effectiveness of our pipeline over standard, off-the-shelf LLMs for the task of generating interactive scenes.

We now discuss the impact of each LLMR module in detail. As explained in Section 3.2,the *Scene Analyzer* allows LLMR to parse and understand the virtual scene and is thus indispensable for meaningful manipulations of existing environments. Consequently, enhancing GPT-4 with the *Scene Analyzer* results in a significant performance enhancement in the Bathroom scene. Secondly, the *Inspector* module enables LLMR to perform self-debugging and effectively prevents the generation of erroneous code, further reducing the error rate in both scenarios. Although we integrated the *Inspector* at the final stage, it is compatible with any combination of modules and will consistently reduce the output error rate. As an example, we added *Inspector* to GPT-4 with few-shot prompting in the empty scene and observed the average error drops from 45.0% to 13.1%. We also observe the *Skill Library* has a marginal impact on the error rates. This is expected, since the *Skill Library* is designed to handle more specialized tasks, which we discuss in more detail in the following subsections. Lastly, the *Planner* is not included as it alters the input prompt with step-by-step decomposition, making the results incommensurable. We include in the Appendix an example where the *Planner* is used to build a virtual kitchen, underscoring the benefit of decomposing difficult tasks into incremental steps.

8.2 Error Rate by Levels of Difficulty

8.2.1 Methodology. To explore the relationship between the complexity of prompts and the completion rate of LLMR, we performed an ad-hoc analysis of the results from the previous section. We classified our prompts for single prompt task into levels of difficulty from 1 to 10. To achieve this, we utilized a non-contextualized LLM devoid of any meta-prompting, asking it to assign a difficulty level to each prompt. This process was repeated ten times for each prompt, and the average difficulty level was then calculated (one which had a small standard deviation). The aggregated results, categorized by difficulty level, are illustrated in Figure 15. The prompt given to this LLM (GPT-4) was "*The above are prompts that are given to a system that can code and execute commands inside of Unity. We want to measure how good this system is at coding in C# for Unity purposes. Given your knowledge of Unity, please rate all of the prompts above on a level of difficulty from 1 to 10*". The rationale behind employing a non-contextualized LLM (without any meta-prompting) lies in the subjective nature of assessing difficulty levels. Being the developers of the system, our judgment might be inherently biased, influenced by our understanding of the system's capabilities and limitations. Furthermore, engaging Unity experts to determine the difficulty levels presents its challenges. The variability in the expertise and experience levels among Unity developers could lead to inconsistent evaluations and difficulty in standardizing the experience of the evaluators without a comprehensive and uniform examination framework.

8.2.2 Results and Discussion. In this section, we analyze the performance of various architectures in executing Unity tasks, differentiated by difficulty levels that range from Easy to Hard. These levels were determined based on a 1-10 scale assigned by GPT-4. Figure 15 shows the error rate of the different architectures on two panels. On the left, we have the results for the empty scene and on the right for a scene with a bathroom containing various objects.

Across all levels and scenes, LLMR (orange line) consistently outperforms other architectures, underscoring its robustness. In

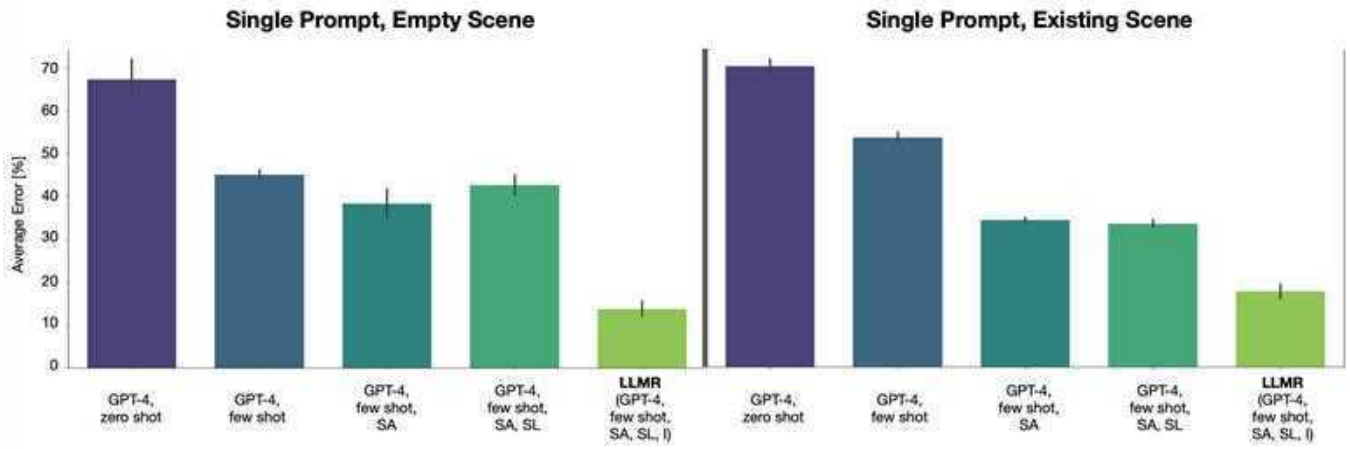

Figure 14: *Comparison for average compilation and run-time error rate.* SA stands for the *Scene Analyzer*, SL stands for the *Skill Library*, and I stands for the *Inspector*. Overall, in both creating from scratch, as well as editing existing scenes, LLMR outperforms GPT-4 by 3x in the case with few-shot prompting, and gives over 4x improvement compared to the performance of zero-shot GPT-4.

the empty scene on the left, a noticeable trend is that the error rate generally increases with the task difficulty. This trend aligns with expectations, except in the case of GPT-4 Zero Shot. A notable point here is that the Easy category only contains a single prompt, which is a basic "Hello World" console display. The simplicity of this task explains its solitary placement in this category. For the bathroom scene, the error rates for Medium and Somewhat Hard tasks show minimal variation, suggesting a plateau in difficulty perception. An interesting observation is the drop in error rates from Easy to Somewhat Easy tasks, although this is not consistent across all models. The integration of the *Skill Library* shows mixed effects (dark blue line). In some instances, it enhances performance, while in others it seems to hinder it.

Estimating the difficulty of tasks, especially in scenarios involving modifications to an existing scene rather than building from scratch, presents challenges. This is exemplified in the bathroom scene, where adding new objects (difficulty levels 3-4) did not require scene understanding, contrary to the tasks in the Easy category, which involved moving objects and thus relied more on scene comprehension. Our analysis of the prompts indicates that the nature of the scene significantly influences the perceived difficulty. For instance, in the bathroom scene, certain tasks categorized as Easy in theory turned out to be more challenging in practice. The **Appendix** offers a more comprehensive analysis, including variations in architecture, such as the combination of *Scene Analyzer* (SA) and *Inspector* modules.

In conclusion, LLMR demonstrates superior performance across various scenarios, underscoring its effectiveness in handling tasks of varying complexity in Unity environments. This analysis also highlights the intricate relationship between task difficulty, scene context, and architectural components, paving the way for further exploration in optimizing task-specific architectures.

8.3 Task Complexity

Complexity can manifest through different aspects. To supplement the ad-hoc analysis above, we now provide a more comprehensive discussion by breaking down the concept of complexity through the following aspects and share findings that emerged throughout our experimentation:

Specific Skill Requirement – Certain tasks are inherently more difficult. For example, deforming the mesh of an object is much more complicated than adding an object to the scene. A human developer may need to look up examples and documentation to achieve a complex task; LLMR can reduce the complexity of the task by starting a templated script. However, LLMR is not error-proof. As shown in the previous section, LLMR's error rate increases (but not above 40%) as a task becomes more difficult. The error rate can be further brought down by adding relevant skills to the *Skill Library* by an experienced developer to help LLMR achieve a higher success rate and reduce possible rounds of iteration between the *Builder* and *Inspector*. LLMR can save the time of experienced developers by generalizing beyond the examples provided in the *Skill Library*.

Token Requirement – The amount of tokens required grows as the scene or the object becomes more complex. For example, if the existing scene has a tree object with many leaves, where each leaf is considered a child game object, the scene summary could easily exceed the maximum token allowed (at the time of writing, the maximum number of tokens was 8k, though this has now grown significantly). In anticipation of this, our *Scene Analyzer* module only fetches the top-level game object name to filter out the relevant game object based on user query and task. This allows our framework to handle a scene as complex as the Kitchen scene (example in Figure 10) and the Bathroom scene (used in Numerical Study, Figure 13). Besides the complexity of objects and scenes, a task itself can also require a lot of tokens. One such example is generating an animation [19] (with the help of a skill written for the *Skill Library*) that involves generating time-series of numerous joint positions

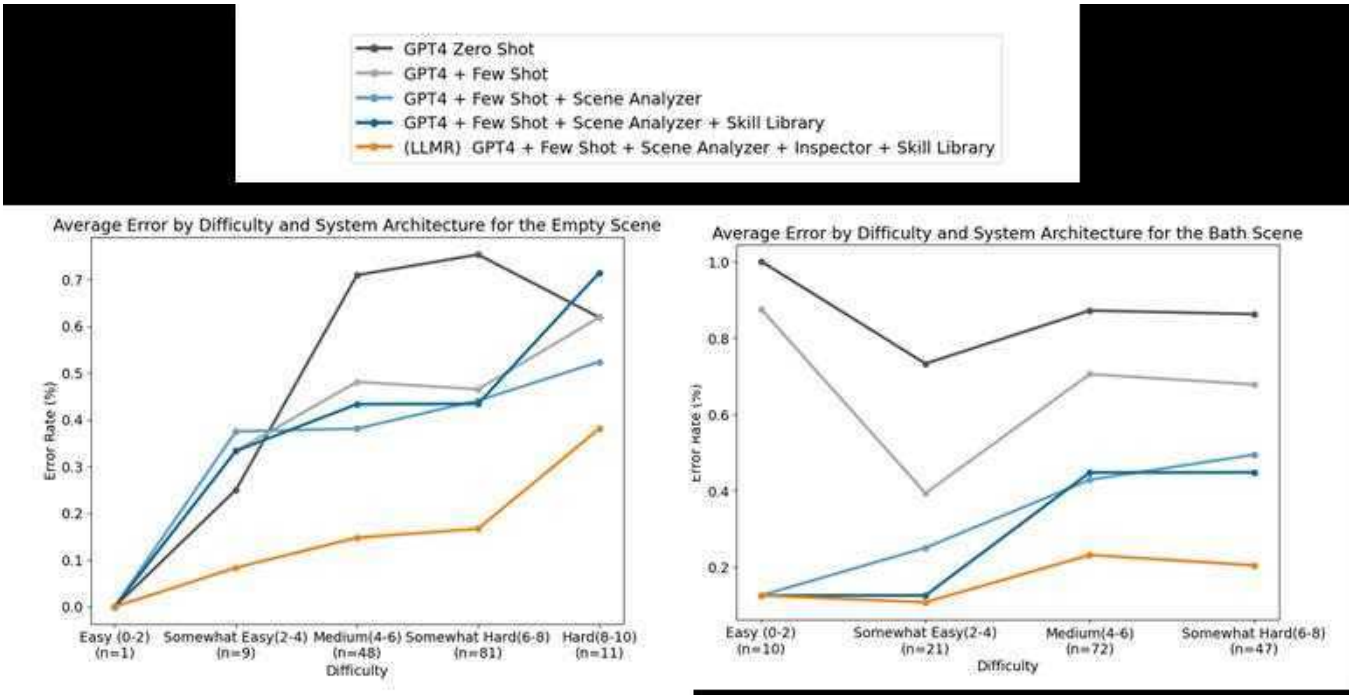

Figure 15: *Performance Improvement of LLMR Modules Organized by Difficulty Level of Prompts.* Comparing Error Rates of GPT-4 with Incremental LLMR Module Integration. The error rates for most methods increase with difficulty, and the LLMR method (in orange) still maintains a consistently lower error rate compared to others.

and rotations. An optimization could involve developing a simple way to represent the time-series information.

Memory Requirement – When a task requires prior knowledge of previous prompts (e.g., behaviors created by previous prompts), this requires previous prompts to have been successfully compiled and to be robust enough. We described approaches to managing the memory of the different modules in Section 5 to both conserve total memory consumed while preserving the necessary information for the framework to carry out complex tasks.

Quality Requirement – A user may request different levels of fidelity of the output. For example, the user could create a complex scene out of primitives only with the help of the *Planner* module (e.g., a full kitchen, Figure 1) instead of out of higher fidelity 3D models (see examples of participants' creations in the video figure). The flexibility to create visually simple yet functional and interactive scenes is akin to creating a lo-fi mockup that allows users to quickly prototype and iterate without waiting for the full generation of 3D scenes that are visually complex but cost a lot of compute and time and are not easy to modify.

8.4 Iterative, Incremental Design

8.4.1 Experimental Setup. In practice, creating content-rich virtual worlds requires incremental steps. Therefore, it is important to assess how LLMR performs in iterative scenarios, where requests are made and fulfilled one after another to gradually build and alter a virtual scene. We tested LLMR with 80 sequential prompts, each averaging 5 single prompts. These sequential prompts consist of a set of instructions aimed at completing a complex task. For instance, a sequential prompt for constructing a bedroom might include steps like *"create an empty room with walls; add a bed with a lamp next to it; add a window on the wall."*

We use three metrics to evaluate performance in an iterative setting. First, the error rate on all individual prompts is considered and is the same as in single tasks. Second, we calculate the average degree of completion, measured as the number of completed single prompts over the sequence length for each sequential prompt. As the sequential prompts have varying lengths, accessing the completion average prevents "long and simple" sequences from flooding the error rate. Lastly, we define *fulfilled prompts* to be sequential prompts that are completed from start to finish and compute their percentage over the total number of prompts. This is a demanding metric that validates whether the model can manage extended use sessions gracefully. In extreme cases, a model excelling only in short sequences can have a reasonable error rate yet zero perfectly fulfilled prompts.

8.4.2 Results and Discussion. The numerical results, presented in Table 2, show that GPT-4's performance in sequential tasks improves significantly with the addition of each LLMR module. When all modules are integrated, LLMR surpasses the standard GPT-4 by approximately 2.5 times across all metrics, aligning with results from single prompt tests. Furthermore, LLMR's memory-efficient design maintains a constant context usage for arbitrary prompt sequences and thus removes token size limitations during prolonged sessions. As such, LLMR demonstrates promising performance in

Model	Error rate (↓)	Avg. prompt completion (↑)	% of fulfilled prompts (↑)
GPT-4	0.745	0.339	0.288
GPT-4, few-shot	0.452	0.624	0.500
GPT-4, few-shot with SA	0.374	0.691	0.575
LLMR	**0.245**	**0.824**	**0.775**

Table 2: Numerical results for sequential prompts. The arrows next to the metrics point to its favored direction. For example, the down arrow next to error rate means a lower error rate should be preferred.

Model	Single prompts, empty [sec]	Single prompts, bathroom [sec]	Sequential prompts [sec]
GPT-4	35.24	20.60	77.10
GPT-4, few-shot	37.82	21.28	112.50
GPT-4, few-shot with SA	33.90	20.58	69.40
GPT-4, few-shot with SA, SL	34.46	21.64	74.60
LLMR	**90.98**	**49.16**	**170.90**

Table 3: Average time taken in seconds to generate and compile each prompt. SA stands for the *Scene Analyzer*, and SL stands for the *Skill Library*. LLMR is equivalent to GPT-4 augmented with the *Analyzer*, the *Skill Library*, and the *Inspector*.

the progressive creation and modification of virtual scenes, a scenario that resonates more closely with practical use cases. Lastly, we discuss in section 9 how the users subjectively rate the iteration process working with our framework.

In general, sequential prompts are much more challenging than single prompts because they require the model to maintain and manage long-range dependencies, a task known to be challenging in sequence modeling [61]. To use the provided example, adding a window on the wall requires knowledge of the wall that was created a few prompts prior. From this perspective, the *Scene Analyzer* serves as an effective summarization [54] that helps the model redirect its attention to the part of the scene most relevant to the request, thereby reducing potential errors. In addition, the *Inspector* receives scene parsing from the *Scene Analyzer* and can thus effectively shield the generated code against potential errors in a sequential setting.

8.5 Real-time

8.5.1 Methodology. Last but not least, an important strength and design goal of our framework is the *real-time* creation and modification of objects and scenes, which is crucial to ensure the practicality of use. To evaluate the framework's real-timeness, we measured the time taken for task completion (from generation to compilation) with different combinations of modules. Once again, the results were from 5 runs of 150 prompts with each model condition. Note that we ran the experiment from August 2023 until December 2023, where the performance and latency of OpenAI's GPT-4 model varied slightly but the difference was marginal. The experiments were run on a PC with 32GB of RAM and an Nvidia RTX 3080 GPU.

8.5.2 Results and Discussion. Table 3 shows the average completion time (i.e., including generation and compilation time) for each model and condition. The off-the-shelf GPT-4 takes around half a minute to complete a single prompt, and the full LLMR framework on average takes a little over a minute – a timeframe we consider acceptable given the task complexity (Figure 15) and improved task completion rate (Figure 14 and Table 2). To put things into context, completing these complex tasks manually takes much longer even for someone reasonably familiar with Unity if we account for the time spent on looking up documentation and debugging.

There are a couple of factors that contribute to the additional operation time. First, the complexity of certain tasks, such as retrieving 3D assets from Sketchfab, requires extra time to download assets from third-party sites. The time needed to finish retrieving a 3D model varies a lot and depends on the size of the model, and thus we did not include this in our evaluation. In this case, our framework anticipates this by caching the previously saved model for faster reloading. Second, to ensure the success rate of our framework, the Inspector nearly doubles the code generation time (see the last two rows of Table 3). This is an inherent tradeoff, and the *Inspector* module can be turned off for simple tasks. Finally, back-and-forth interactions between LLMR and the user as well as iteration over the generated results contribute to the overall development time. It is worth noting that during our user study (Section 9), none of the participants mentioned or complained about generation time. Participants who are novice Unity users appreciated that LLMR saved them time from the steep learning curve. In addition, our "saving and reloading" capability (Section 3.4) allows users to iterate faster by reusing prior creations, which takes less than 10 seconds to recompile.

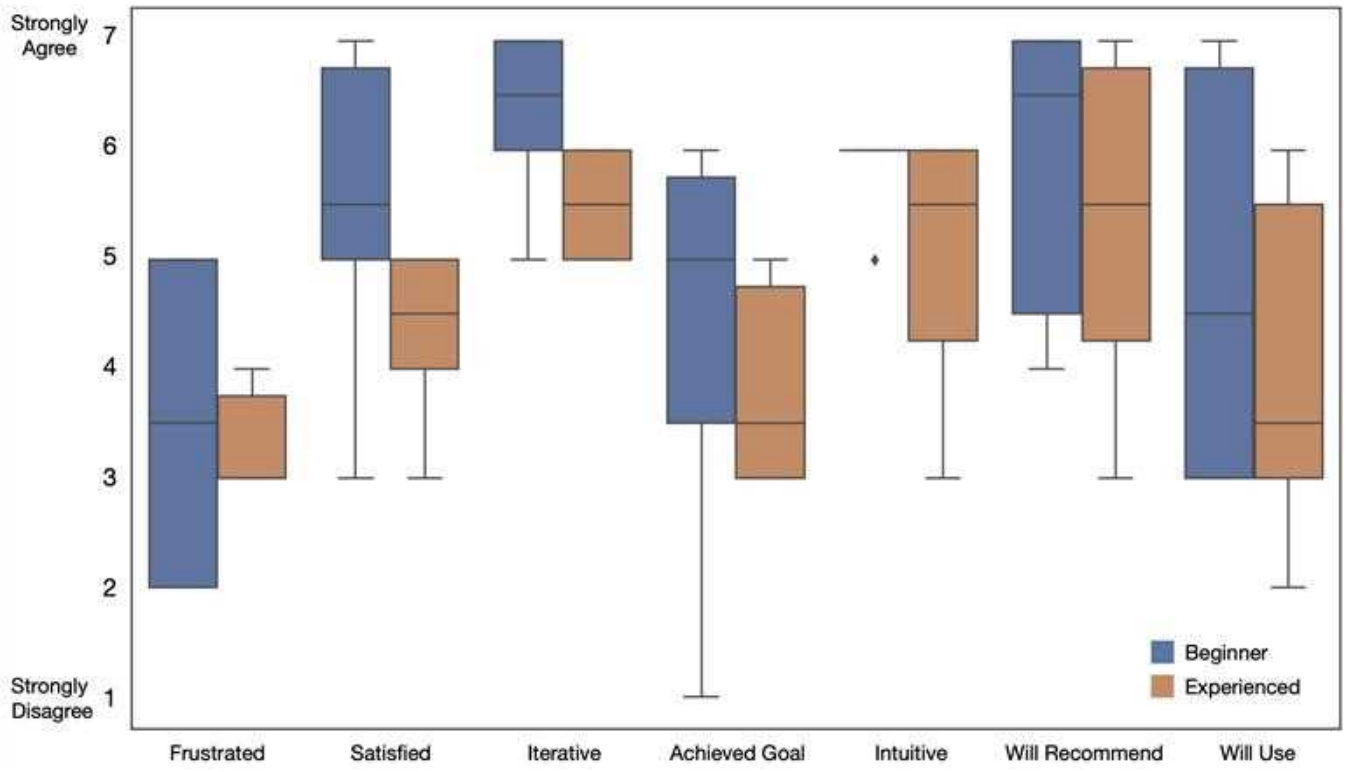

Figure 16: Results of the user study for both experienced and beginner users of Unity. Overall, the users found LLMR satisfactory and would recommend others to use it too.

9 USABILITY STUDY

The ablation test focused on only the compilation and run-time errors in the code generated by our framework. We also wanted to evaluate the quality of the generated output with human users. In addition, we also wanted to understand how users with different levels of familiarity with Unity would use our framework.

9.1 Procedure

We recruited twelve users (1 pilot, 11 participants) with different levels of experience using Unity (5 participants had more than one year of Unity experience). The participants' backgrounds were software engineers, product managers, or researchers. Each session took around 2 hours, and each participant had at least 1.5 hours to experiment with the framework. We provided a unity package that includes basic features (*Scene Analyzer* and *Skill Library, Builder*, and *Inspector*). Before the study, each participant downloaded the package to an empty or existing Unity scene and followed the instructions to set it up. Each participant went through a few rounds of interaction with the framework. A round of interaction could look like the following. The participant types: "Create a tool that changes the color of the car." The framework processed the prompt and generated scripts that were then automatically compiled at runtime. The participant looked at the generated output and decided on the next prompt. The investigator might suggest different things to try or remind the participant of the capabilities of the framework. They were asked to think out loud throughout the study. At the end, the investigators conducted a semi-structured interview with the participant (see **Appendix** for the full list of questions). After the study, each participant filled out a seven-question questionnaire on a seven-point Likert-scale about their experience using the framework.

9.2 Results and Design Recommendation

Participants were able to generate various outputs using our framework, such as cities and Asteroids-like games. Some even recreated their professional work, such as rigging camera angles and generating animations.

We used a mixed-methods approach to analyze the user study; We took into account the quantitative insights from the questionnaire response, and we thematically grouped participants' think-aloud and semi-structured interview responses to identify patterns. These findings were then utilized to generate a set of design suggestions, which we will discuss in detail.

Questionnaire results revealed that participants generally had positive experiences with our framework in terms of achieving their goals, intuitiveness, and iterative use. However, there is room for improvement in reducing frustration and further enhancing user satisfaction (Figure 16). We also compared the responses between beginners and experienced Unity users. Beginners rate their experience with our framework more positively across most categories, as we will detail in the following sections.

9.2.1 Approach to Prompting and Instruction Strategies. We asked participants to describe their approach to prompting when using our framework. Participants emphasized the importance of ensuring that their prompts were easy for GPT to understand (P1, P2). Some participants treated the interaction with the framework as an experimental playground, experimenting with different prompts and refining them over time through trial-and-error (P0, P6). Many participants stressed the need to be highly specific in their instructions. This involved specifying object names, exact changes, and detailed parameters to achieve desired results and avoid unpredictability (P3, P4, P5, P9, P11). Many took the approach of breaking down tasks into smaller, more manageable steps. This included starting with simple components and gradually adding complexity (P4, P5, P6, P7, P8, P10). When creating environments or settings, participants often prioritized static elements before motion-centric ones and ensured that interactive elements responded to the environment (P7).

9.2.2 Comparison with Prior Approach to 3D World Creation. When asked to compare our framework to their prior experience of creating 3D worlds, several participants appreciated the ease of describing their ideas directly to the model, eliminating the need for extensive manual scripting or documentation reference (P1, P3, P5, P6). Participants appreciated our framework's integration capabilities and its ability to automate certain aspects of 3D world creation, such as selecting and loading objects from 3D model repositories like Sketchfab and determining model placement (P4, P7). Some participants mentioned that our framework reduced the learning curve for newcomers, making it easier to get started with 3D world creation (P3). Participants appreciated the ability to directly intervene and manually adjust the 3D world generated by our framework, which they considered a powerful feature compared to other uses of generative models, which do not allow the user to adjust the final output (P9, P11).

9.2.3 Considerations and Challenges. Participants noted that our framework's output was often unpredictable compared to traditional methods, and there was uncertainty about whether the model would understand the desired structure (P0). Some participants pointed out that the choice between traditional methods and our generative method of creating 3D worlds might depend on the artistic nature of the project and the need for creative input (P5, P7, P11). Other participants recognized that for projects requiring precise, structured, or rigid control, traditional tools might be preferred over our framework (P4, P8). Lastly, they also mentioned that for more complex tasks or as projects grew in size, manual code editing might still be necessary, as it could be faster than creating detailed descriptions of edits for the model (P1, P4, P8).

9.2.4 User Expectations and Surprises. We also probed participants' expectations and what they were surprised by during the user study. Many were surprised by our framework's ability to generate code effectively, helping them automate complex scripting tasks (P1, P4, P6, P8, P10). Specifically, P4 was impressed by the model's ability to handle complex structures like trees, despite token limits. P10 also highlighted the framework's ability to understand the hierarchy of game objects and utilize this information as input, especially in large and complex projects. Participants were impressed by the integration capabilities of our framework with Dall-E 2 and Sketchfab, which allowed for the creation of complex structures and the addition of 3D objects (P4, P6, P8, P10). It was surprising for participants to discover that our framework allowed for subjective queries and descriptions, accommodating plain language and euphemisms rather than strictly technical terms (P5). P1 was also amazed that the framework exhibited flexibility in understanding their Unity scripts and could even help resolve errors. Similarly, some were pleasantly surprised that, when prompted correctly, our framework could produce unconventional or unexpected results, such as unique player movement (P0, P3).

10 ETHICAL CONSIDERATIONS

While LLMR and other LLM tools can be transformative to many industries and applications, there exist risks with any AI-enabled systems. Firstly, the concern of developers and creators being replaced has been on the surface of discussion. However, these tools have not been proven to achieve end-to-end development. Participants of our user study commented that our framework is better at integrating human intervention and involvement, and thus our framework helps improve productivity and facilitate brainstorming, rather than completely automating the creation process. A more serious concern is the potential for individuals to generate harmful and inappropriate content with our framework. Despite the safeguards put in by Sketchfab and OpenAI through content moderation and model alignment, it is still possible to creatively circumvent these safeguards [29]. While the Roslyn compiler can automatically check for unsafe code, the need for research on how to moderate 3D content is merited.

11 LIMITATIONS AND FUTURE WORK

Limitation in scene understanding – Currently, our framework requires access to a "scene graph" with descriptions and the hierarchy of the game objects. The scene graph provides the spatial relationship of game objects and it assumes that the names of the game objects (and their children game object components) are correct and unique. However, 3D models from repositories like Sketchfab often have random, non-descriptive component names. At the moment, the framework manipulates objects by finding the game objects with the exact name, which is not always reliable.

In addition, the scene hierarchy does not contain meta information about the objects, such as affordance and functions. Furthermore, a scene graph would not be readily available when the scene is a physical environment with augmented virtual objects. The natural next step is the incorporation of Large Vision Models (LVM) [18] to achieve tasks that require visual knowledge and semantic understanding of objects and environments. Our framework can benefit from the enhanced feedback and semantic information from these models, and our framework can enable more interactive editing of and interactions with a given 3D environment.

Incorporating world feedback and direct user feedback – Similar to how the *Builder-Inspector* loop reduces code compilation error, the framework's understanding of the world could be further improved by incorporating feedback from the virtual world and from the user. For example, if a 3D model is loaded from Sketchfab, the framework is ignorant of the model's (and their subcomponents') orientation

and center of pivot, and thus does not consistently produce the desired output when asked to rotate the 3D model.

Limitation in Memory and Traceability – Another limitation of the framework is the token size. As mentioned in Section 5, we have optimized access to historical conversation and generation to reduce token usage. There is an inherent tradeoff, where the user instruction might refer to something in a previous prompt exchange that is not exposed to the next exchange. For example, the *Scene Analyzer* has access to the name of the script, the summary of the script, and the public fields of the script, but if the user just wants to change a specific part of a previous script generated two prompts prior, the framework would not know what to do.

Correspondingly, showing the generated code gives traceability and transparency to the results of our framework. At the moment, code written by our framework is stored locally in a cached folder and can be viewed within the Unity editor window. In addition to providing feedback via a follow-up prompt, the option to directly edit the code generated by our framework would give users more agency and achieve more complex, precise tasks (as mentioned in Section 9.2.3).

Automatic Skill Generation – At the moment, skills in the *Skill Library* are created by human users. For example, we incorporated the skill of loading assets from Sketchfab as well as the skill of making objects "grabbable" using MRTK's [31] namespaces. The ability to automatically generate new skills [53] based on a couple of examples would allow our framework to achieve more complex tasks (such as generating animations) and to be compatible with different platforms (such as Quest and ARKit).

Interoptability – We built upon the Unity engine for its robustness and the large amount of existing examples of C# code that our LLMs have likely seen during training. Our work is independent of Unity and its closed-beta AI tools [1], although our tool can be an add-on to Unity. We want to highlight that our framework can be adapted to any environment that supports run-time compilation. Unity is the baseline requirement for using our framework, and a web-based approach would further make prompt-based interactive 3D worlds easy to share and collaborate within. In fact, some of our user study participants work on web-based mixed reality development, and they commented that our framework can be easily adapted to their coding environment.

12 CONCLUSION

In this paper, we have introduced a novel framework that addresses certain difficulties in applying LLMs to generate interactive 3D experiences. This framework leverages the abilities of multiple distinct and specialized LLM modules, orchestrated in a way that enhances their individual and collective performance on both coding and reasoning. Additionally, we have presented certain engineering aids, such as a skill that utilizes other AI models to add content into scenes, further expanding the capabilities of our framework.

Our research has demonstrated the benefits of each LLM-based module, providing a clear rationale for the inclusion of each module in our framework. By combining somewhat specialized components, our overall system became more robust and is significantly better than off-the-shelf LLMs. Through a user study, we have tested the quality and usability of our framework, allowing participants to challenge our framework with unprecedented prompts, thereby pushing the boundaries of the examples provided to LLMs.

The significance of this work lies in its potential to improve the generation of virtual world content with internal degrees of freedom and interactivity, and to improve the likelihood that such content will make sense intuitively to humans in a human-scale world. In turn, this shows a path to making LLMs more reliable in the domain of human-scale activity. The LLM is not merely incorporating what has been said about the world, but tests results in a simulation of the world. The described framework operates across various devices and platforms; the present implementation does assume Unity.

We propose that this framework offers an opportunity for the HCI community studying LLMs. By providing virtual- or real-world data and the ability to act via code in such a world, our framework can serve as a platform to test and improve the limits of LLM reasoning capabilities when placed in 3D environments.

In conclusion, our work presents a significant step forward in the integration of LLMs with virtual world content and experience generation, offering a powerful tool for both developers and researchers. We look forward to seeing how this framework will be utilized and expanded upon by the wider community.

[1] Link to Unity's closed-beta AI tools: https://unity.com/ai

ACKNOWLEDGMENTS

We would like to thank various collaborators who gave very useful comments and suggestions, in particular Jennifer Marsman, Jason Carter, Sebastien Vandenberghe, Haiyan Zhang, Nebojsa Jojic, Andy Wilson, Gavin Jancke, Grace Huang, Lily Cheng, Octavio Martinez, Pavan Davuluri, Robin Seiler, Ruben Caballero, Anuj Gosalia, David Wolf, Marc Pollefeys, Miguel Susffalich, Allan Naim, Darren Bennett, Doug Berrett, Gilles Zunino, Jay Watts, Tucker Burns, Preet Mangat. Additionally, we would like to thank all the study participants.

REFERENCES

[1] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program Synthesis With Large Language Models. *ArXiv Preprint ArXiv:2108.07732* (2021). https://arxiv.org/pdf/2108.07732

[2] Sanghwan Bae, Donghyun Kwak, Soyoung Kang, Min Young Lee, Sungdong Kim, Yuin Jeong, Hyeri Kim, Sang-Woo Lee, Woomyoung Park, and Nako Sung. 2022. Keep Me Updated! Memory Management in Long-Term Conversations. *ArXiv Preprint ArXiv:2210.08750* (2022). https://arxiv.org/pdf/2402.11975

[3] Erik Bethke. 2003. *Game Development and Production.* Wordware Publishing, Inc.

[4] Ang Cao and Justin Johnson. 2023. Hexplane: A Fast Representation for Dynamic Scenes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition.* 130–141. https://doi.org/10.1109/CVPR52729.2023.00021

[5] Yifei Cheng, Yukang Yan, Xin Yi, Yuanchun Shi, and David Lindlbauer. 2021. Semanticadapt: Optimization-Based Adaptation of Mixed Reality Layouts Leveraging Virtual-Physical Semantic Connections. In *The 34th Annual ACM Symposium on User Interface Software and Technology.* 282–297. https://doi.org/10.1145/3472749.3474750

[6] Fenia Christopoulou, Gerasimos Lampouras, Milan Gritta, Guchun Zhang, Yinpeng Guo, Zhongqi Li, Qi Zhang, Meng Xiao, Bo Shen, Lin Li, et al. 2022. PanguCoder: Program Synthesis With Function-Level Language Modeling. *ArXiv Preprint ArXiv:2207.11280* (2022). https://arxiv.org/pdf/2207.11280

[7] Stephanie Claudino Daffara, Federico Saldarini, Balasaravanan Thoravi Kumaravel, and Bjørn Hartmann. 2020. AuthorIVE: Authoring Interactions for Virtual Environments Through Disambiguating Demonstrations. (2020).

[8] Danny Driess, Fei Xia, Mehdi SM Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, et al. 2023. Palm-E: An Embodied Multimodal Language Model. *ArXiv Preprint ArXiv:2303.03378* (2023). https://arxiv.org/pdf/2303.03378.

[9] Catarina G Fidalgo, Yukang Yan, Hyunsung Cho, Mauricio Sousa, David Lindlbauer, and Joaquim Jorge. 2023. A Survey on Remote Assistance and Training in Mixed Reality Environments. *IEEE Transactions on Visualization and Computer Graphics* 29, 5 (2023), 2291–2303. https://doi.org/10.1145/3533376

[10] Jonas Freiknecht and Wolfgang Effelsberg. 2017. A Survey on the Procedural Generation of Virtual Worlds. *Multimodal Technologies and Interaction* 1, 4 (2017), 27. https://doi.org/10.1111/cgf.12276

[11] Kyle Gao, Yina Gao, Hongjie He, Denning Lu, Linlin Xu, and Jonathan Li. 2022. Nerf: Neural Radiance Field in 3d Vision, a Comprehensive Review. *ArXiv Preprint ArXiv:2210.00379* (2022). https://arxiv.org/pdf/2210.00379

[12] Google. 2016. Tilt Brush. https://www.tiltbrush.com/

[13] Eric Guerin, Julie Digne, Eric Galin, Adrien Peytavie, Christian Wolf, Bedrich Benes, and Benoit Martinez. 2017. Interactive Example-Based Terrain Authoring With Conditional Generative Adversarial Networks. *ACM Trans. Graph* 36, 6 (2017), 228–1. https://doi.org/10.1145/3130800.3130804

[14] Izzeddin Gur, Hiroki Furuta, Austin Huang, Mustafa Safdari, Yutaka Matsuo, Douglas Eck, and Aleksandra Faust. 2023. A Real-World Webagent With Planning, Long Context Understanding, and Program Synthesis. *ArXiv Preprint ArXiv:2307.12856* (2023).

[15] Ayaan Haque, Matthew Tancik, Alexei A Efros, Aleksander Holynski, and Angjoo Kanazawa. 2023. Instruct-Nerf2nerf: Editing 3d Scenes With Instructions. *ArXiv Preprint ArXiv:2303.12789* (2023).

[16] Teresa Hirzle, Florian Muller, Fiona Draxler, Martin Schmitz, Pascal Knierim, and Kasper Hornbaek. 2023. When XR and AI Meet-A Scoping Review on Extended Reality and Artificial Intelligence. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–45. https://doi.org/10.1145/3544548.3581072

[17] Lukas Höllein, Ang Cao, Andrew Owens, Justin Johnson, and Matthias Nießner. 2023. Text2room: Extracting Textured 3d Meshes From 2d Text-to-Image Models. *ArXiv Preprint ArXiv:2303.11989* (2023).

[18] Yining Hong, Haoyu Zhen, Peihao Chen, Shuhong Zheng, Yilun Du, Zhenfang Chen, and Chuang Gan. 2023. 3D-LLM: Injecting the 3D World Into Large Language Models. *ArXiv Preprint ArXiv:2307.12981* (2023).

[19] Han Huang, Fernanda De La Torre, Cathy Mengying Fang, Andrzej Banburski-Fahey, Judith Amores, and Jaron Lanier. 2023. Real-time Animation Generation and Control on Rigged Models via Large Language Models. *arXiv preprint arXiv:2310.17838* (2023).

[20] Ziheng Huang, Sebastian Gutierrez, Hemanth Kamana, and Stephen MacNeil. 2023. Memory Sandbox: Transparent and Interactive Memory Management for Conversational Agents. *ArXiv Preprint ArXiv:2308.01542* (2023). https://arxiv.org/pdf/2308.01542

[21] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large Language Models Meet Program Synthesis. In *Proceedings of the 44th International Conference on Software Engineering*. 1219–1231. https://doi.org/10.1145/3510003.3510203

[22] Heewoo Jun and Alex Nichol. 2023. Shap-E: Generating Conditional 3d Implicit Functions. *ArXiv Preprint ArXiv:2305.02463* (2023). https://arxiv.org/pdf/2305.02463

[23] Animesh Karnewar, Andrea Vedaldi, David Novotny, and Niloy J Mitra. 2023. Holodiffusion: Training a 3D Diffusion Model Using 2D Images. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 18423–18433.

[24] Eric Kolve, Roozbeh Mottaghi, Winson Han, Eli VanderBilt, Luca Weihs, Alvaro Herrasti, Matt Deitke, Kiana Ehsani, Daniel Gordon, Yuke Zhu, et al. 2017. Ai2-Thor: An Interactive 3d Environment for Visual Ai. *ArXiv Preprint ArXiv:1712.05474* (2017). https://arxiv.org/pdf/1712.05474

[25] Gang Li, Heliang Zheng, Chaoyue Wang, Chang Li, Changwen Zheng, and Dacheng Tao. 2022. 3ddesigner: Towards Photorealistic 3d Object Generation and Editing With Text-Guided Diffusion Models. *ArXiv Preprint ArXiv:2211.14108* (2022).

[26] Chen-Hsuan Lin, Jun Gao, Luming Tang, Towaki Takikawa, Xiaohui Zeng, Xun Huang, Karsten Kreis, Sanja Fidler, Ming-Yu Liu, and Tsung-Yi Lin. 2023. Magic3d: High-Resolution Text-to-3d Content Creation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 300–309. https://doi.org/10.1109/CVPR52729.2023.00037

[27] David Lindlbauer, Anna Maria Feit, and Otmar Hilliges. 2019. Context-Aware Online Adaptation of Mixed Reality Interfaces. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. 147–160. https://doi.org/10.1145/3332165.3347945

[28] Xin Liu, Daniel McDuff, Geza Kovacs, Isaac Galatzer-Levy, Jacob Sunshine, Jiening Zhan, Ming-Zher Poh, Shun Liao, Paolo Di Achille, and Shwetak Patel. 2023. Large Language Models are Few-Shot Health Learners. *arXiv preprint arXiv:2305.15525* (2023).

[29] Yi Liu, Gelei Deng, Zhengzi Xu, Yuekang Li, Yaowen Zheng, Ying Zhang, Lida Zhao, Tianwei Zhang, and Yang Liu. 2023. Jailbreaking Chatgpt via Prompt Engineering: An Empirical Study. *ArXiv Preprint ArXiv:2305.13860* (2023). https://arxiv.org/pdf/2305.13860

[30] Zhao Mandi, Shreeya Jain, and Shuran Song. 2023. RoCo: Dialectic Multi-Robot Collaboration With Large Language Models. *ArXiv Preprint ArXiv:2307.04738* (2023). https://arxiv.org/pdf/2307.04738

[31] Microsoft. 2017. Mixed Reality Toolkit. https://github.com/microsoft/MixedRealityToolkit-Unity

[32] Microsoft. 2023. Mixed Reality Mobile Remoting. https://github.com/microsoft/Mixed-Reality-Remoting-Unity

[33] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An Open Large Language Model for Code With Multi-Turn Program Synthesis. *ArXiv Preprint ArXiv:2203.13474* (2022). https://arxiv.org/pdf/2203.13474

[34] OpenAI. 2023. GPT-4 Technical Report. *ArXiv* abs/2303.08774 (2023). https://arxiv.org/pdf/2303.08774.

[35] Ben Poole, Ajay Jain, Jonathan T Barron, and Ben Mildenhall. 2022. Dreamfusion: Text-to-3d Using 2d Diffusion. *ArXiv Preprint ArXiv:2209.14988* (2022). https://arxiv.org/pdf/2209.14988

[36] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. Learning Transferable Visual Models From Natural Language Supervision. https://doi.org/10.1109/CVPR52688.2022.00101 arXiv:2103.00020 [cs.CV]

[37] Krishan Rana, Jesse Haviland, Sourav Garg, Jad Abou-Chakra, Ian Reid, and Niko Suenderhauf. 2023. SayPlan: Grounding Large Language Models Using 3D Scene Graphs for Scalable Task Planning. *ArXiv Preprint ArXiv:2307.06135* (2023). https://arxiv.org/pdf/2307.06135

[38] Elad Richardson, Gal Metzer, Yuval Alaluf, Raja Giryes, and Daniel Cohen-Or. 2023. Texture: Text-Guided Texturing of 3d Shapes. *ArXiv Preprint ArXiv:2302.01721* (2023). https://arxiv.org/pdf/2302.01721

[39] Jasmine Roberts, Andrzej Banburski-Fahey, and Jaron Lanier. 2022. Steps Towards Prompt-Based Creation of Virtual Worlds. *ArXiv Preprint ArXiv:2211.05875* (2022). https://arxiv.org/pdf/2211.05875

[40] Jasmine Roberts, Andrzej Banburski-Fahey, and Jaron Lanier. 2022. Surreal VR Pong: LLM Approach to Game Design. In *36th Conference on Neural Information Processing Systems (NeurIPS 2022)*. https://www.microsoft.com/en-us/research/publication/surreal-vr-pong-llm-approach-to-game-design/

[41] Uriel Singer, Shelly Sheynin, Adam Polyak, Oron Ashual, Iurii Makarov, Filippos Kokkinos, Naman Goyal, Andrea Vedaldi, Devi Parikh, Justin Johnson, et al. 2023. Text-to-4d Dynamic Scene Generation. *ArXiv Preprint ArXiv:2301.11280* (2023).

[42] Yizhi Song, Zhifei Zhang, Zhe Lin, Scott Cohen, Brian Price, Jianming Zhang, Soo Ye Kim, and Daniel Aliaga. 2023. ObjectStitch: Object Compositing With Diffusion Model. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 18310–18319. https://doi.org/10.1109/CVPR52729.2023.00582

[43] Misha Sra, Sergio Garrido-Jurado, and Pattie Maes. 2017. Oasis: Procedurally Generated Social Virtual Spaces From 3d Scanned Real Spaces. *IEEE Transactions on Visualization and Computer Graphics* 24, 12 (2017), 3174–3187. https://doi.org/10.1007/978

[44] Misha Sra, Sergio Garrido-Jurado, Chris Schmandt, and Pattie Maes. 2016. Procedurally Generated Virtual Reality From 3D Reconstructed Physical Space. In *Proceedings of the 22nd ACM Conference on Virtual Reality Software and Technology*. 191–200. https://doi.org/10.1145/2993369.2993372

[45] Misha Sra, Prashanth Vijayaraghavan, Pattie Maes, Deb Roy, et al. 2017. Deepspace: Mood-Based Image Texture Generation for Virtual Reality From Music. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*. 41–50.

[46] Unity Technologies. 2019. Unity Render Streaming. https://docs.unity3d.com/Packages/com.unity.renderstreaming@2.0/manual/index.html

[47] OpenAI. 2022. DALL·E 2. https://openai.com/dall-e-2

[48] Sketchfab, Inc. 2023. Sketchfab. https://sketchfab.com/

[49] Trivial Interactive. 2019. Roslyn C# - Runtime C# Compiler. https://forum.unity.com/threads/released-roslyn-c-runtime-c-compiler.651505/

[50] Unity Technologies. 2005. Unity Game Engine. https://unity.com/

[51] Balasaravanan Thoravi Kumaravel, Fraser Anderson, George Fitzmaurice, Bjoern Hartmann, and Tovi Grossman. 2019. Loki: Facilitating Remote Instruction of Physical Tasks Using Bi-Directional Mixed-Reality Telepresence. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. 161–174. https://doi.org/10.1145/3332165.3347872

[52] Ryan Volum, Sudha Rao, Michael Xu, Gabriel A 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 *The Third Wordplay: When Language Meets Games Workshop*. https://openreview.net/forum?id=I9glM3N6iAa

[53] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An Open-Ended Embodied Agent With Large Language Models. *ArXiv Preprint ArXiv:2305.16291* (2023). https://arxiv.org/pdf/2305.16291

[54] Qingyue Wang, Liang Ding, Yanan Cao, Zhiliang Tian, Shi Wang, Dacheng Tao, and Li Guo. 2023. Recursively Summarizing Enables Long-Term Dialogue Memory in Large Language Models. *ArXiv Preprint ArXiv:2308.15022* (2023). https://arxiv.org/pdf/2308.15022

[55] Weizhi Wang, Li Dong, Hao Cheng, Xiaodong Liu, Xifeng Yan, Jianfeng Gao, and Furu Wei. 2023. Augmenting Language Models With Long-Term Memory. *ArXiv Preprint ArXiv:2306.07174* (2023).

[56] Zehan Wang, Haifeng Huang, Yang Zhao, Ziang Zhang, and Zhou Zhao. 2023. Chat-3D: Data-Efficiently Tuning Large Language Model for Universal Dialogue of 3D Scenes. *ArXiv Preprint ArXiv:2308.08769* (2023). https://arxiv.org/pdf/2308.08769

[57] Zhenyu Wu, Ziwei Wang, Xiuwei Xu, Jiwen Lu, and Haibin Yan. 2023. Embodied Task Planning With Large Language Models. *ArXiv Preprint ArXiv:2307.01848* (2023). https://arxiv.org/pdf/2307.01848

[58] Xuhai Xu, Anna Yu, Tanya R Jonker, Kashyap Todi, Feiyu Lu, Xun Qian, João Marcelo Evangelista Belo, Tianyi Wang, Michelle Li, Aran Mun, et al. 2023. XAIR: A Framework of Explainable AI in Augmented Reality. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–30. https://doi.org/10.1145/3544548.3581500

[59] Lvmin Zhang, Anyi Rao, and Maneesh Agrawala. 2023. Adding Conditional Control to Text-to-Image Diffusion Models. arXiv:2302.05543 [cs.CV]

[60] Yaqi Zhang, Di Huang, Bin Liu, Shixiang Tang, Yan Lu, Lu Chen, Lei Bai, Qi Chu, Nenghai Yu, and Wanli Ouyang. 2023. MotionGPT: Finetuned LLMs Are General-Purpose Motion Generators. *ArXiv Preprint ArXiv:2306.10900* (2023).

[61] Jingyu Zhao, Feiqing Huang, Jia Lv, Yanjie Duan, Zhen Qin, Guodong Li, and Guangjian Tian. 2020. Do RNN and LSTM Have Long Memory?. In *International Conference on Machine Learning*. PMLR, PMLR, 11365–11375. https://doi.org/10.5555/3524938.3525992

[62] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A Survey of Large Language Models. *ArXiv Preprint ArXiv:2303.18223* (2023). https://arxiv.org/pdf/2303.18223.

[63] Wanjun Zhong, Lianghong Guo, Qiqi Gao, and Yanlin Wang. 2023. MemoryBank: Enhancing Large Language Models With Long-Term Memory. *ArXiv Preprint ArXiv:2305.10250* (2023). https://arxiv.org/pdf/2305.10250

[64] Zhongyi Zhou, Jing Jin, Vrushank Phadnis, Xiuxiu Yuan, Jun Jiang, Xun Qian, Jingtao Zhou, Yiyi Huang, Zheng Xu, Yinda Zhang, et al. 2023. InstructPipe: Building Visual Programming Pipelines with Human Instructions. *arXiv preprint arXiv:2312.09672* (2023).

[65] Xiangyang Zhu, Renrui Zhang, Bowei He, Ziyu Guo, Ziyao Zeng, Zipeng Qin, Shanghang Zhang, and Peng Gao. 2023. Pointclip V2: Prompting Clip and Gpt for Powerful 3d Open-World Learning. In *ICCV*, Vol. 2. 5. https://doi.org/10.1109/ICCV51070.2023.00249