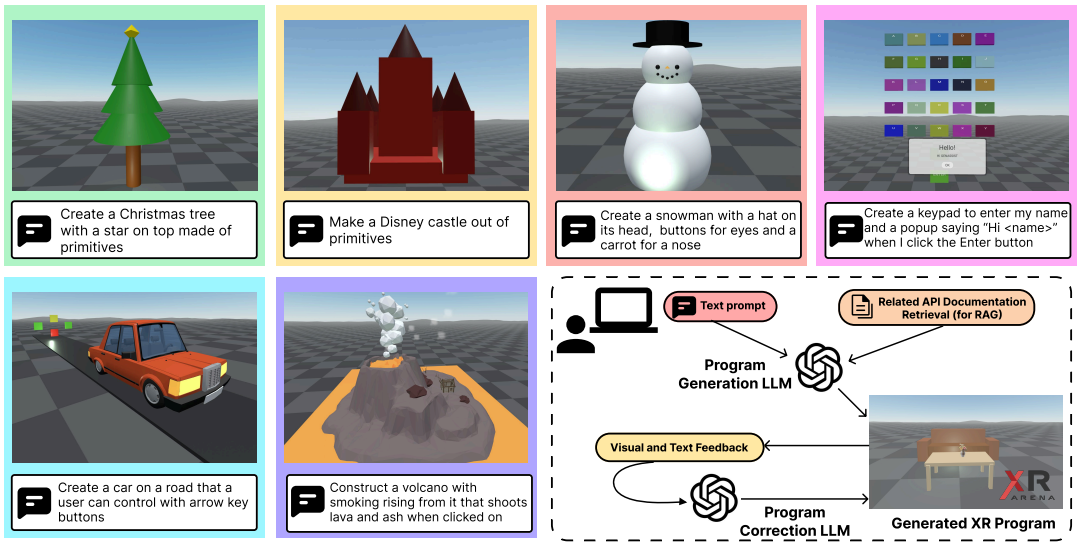
GenAssist: Interactive Prompt-Driven XR Program Generation

Sruti Srinidhi* Carnegie Mellon University
Akul Singh† Carnegie Mellon University
Edward Lu‡ Carnegie Mellon University
Anthony Rowe§ Carnegie Mellon University, Bosch Research



(a) Example Programs Generated using GenAssist **(b) High Level System Diagram and Workflow for GenAssist**

Figure 1: **GenAssist overview and examples. (a)** Example XR programs generated from natural-language prompts: procedural modeling with a set of base objects ("Christmas tree with a star"; "disney castle"), compositional object creation ("snowman with a hat, button eyes, and a carrot nose"), GUI and event handling ("keypad to enter a name and a pop-up saying 'Hi <name>' on Enter"), continuous control ("car that the user drives with arrow keys"), and event-triggered effects ("volcano that smokes and erupts on click"). *Images are static renderings; many examples involve interactions (e.g., clicking, and animations) that are present in the generated programs but not visible in the stills.* **(b)** System workflow: a user writes a text prompt; a Program-Generation LLM (aided by retrieval-augmented API documentation) synthesizes code and generate an XR program; the system then captures visual and textual snapshots; a Program-Correction LLM inspects these artifacts and edits the code; the loop repeats until the requested scene and behavior are achieved, yielding the final XR program.

Abstract

This paper introduces GenAssist, a system for generating interactive Extended Reality (XR) programs from natural language prompts. Given plain text descriptions of desired programs, our system uses Retrieval-Augmented Generation (RAG) to retrieve related documentation and example code, which is then used to prompt Large Language Models (LLMs) to generate and execute hot-pluggable XR programs in real time. To ensure that the programs are written correctly to the user's specifications, we add a closed-loop feedback mechanism using virtual cameras in the scene that iteratively refines the system's output, mimicking the development cycle of human developers that compile and then interactively test programs. GenAssist generates scripts that can not only place multiple primitives and 3D models in various locations in a virtual scene, but it can also animate and enable user interactions with those objects. We show that across a benchmark of 50 diverse XR program prompts, our system achieves high output accuracy and program generation quality. Furthermore, we conduct a user study with 18 participants that demonstrates GenAssist's effectiveness and usability (NASA TLX = 36.6) for XR program generation. We compare GenAssist to prior systems and show that it is significantly faster (<10 seconds per run) and requires fewer LLM calls.

Index Terms: Virtual Reality, Extended Reality, Large Language Models, Program Generation

1 Introduction

With the rapid advancement of Extended Reality (XR) technologies, 3D applications will one day move beyond niche use cases and become part of everyday experiences. Once limited to specialized fields, 3D content will soon be found in a wider range of applications, from immersive entertainment to interactive tools that blend digital content with the physical world. Despite this growing adoption, the development of XR programs remains complex. It often requires a steep learning curve and a combination of skills in design, programming, art, and many software tools. Even creating simple 3D interactions can be time-consuming, while advanced applications require significant effort and expertise. This high baseline effort creates a barrier for non-experts, limiting opportunities for rapid prototyping, experimentation, and broader participation in XR content creation.

Similarly, AI-powered chatbots and assistants are rapidly advancing to the point where they can be used as a unique tool to

*e-mail: ssrinidh@andrew.cmu.edu
†e-mail: akuls@andrew.cmu.edu
‡e-mail: elu2@andrew.cmu.edu
§e-mail: agr@andrew.cmu.edu

simplify XR authoring. While this vision is promising, text-only chatbots are not yet sufficient for highly visual workflows like XR development, which rely on the visual and interactive output of the program. Ideally, we need mechanisms that can both generate XR programs and give developers visual feedback into the scene so that they can iteratively evaluate their output.

This paper presents **GenAssist**, a system that enables users to create interactive XR programs in real time using natural language prompts. GenAssist leverages Large Language Models (LLMs) to generate code that integrates with interactive XR scripting environments. In our implementation, we target programs that run on the ARENA platform [30], which exposes a WebXR front-end to dynamically load and execute Python programs. Users can interact with these ARENA XR programs in standard browsers (in 3D) or in immersive mode on headsets. GenAssist creates programs that allow users to place 3D objects, import existing models, create animations, and define interactive behaviors, all by simply describing their goals in natural language. For example, a user can enter *"create a tree"* instead of manually searching for a model or assembling one from primitive objects, or *"make the cube rotate when clicked"* rather than writing event-driven code from scratch. Our system also supports iterative program development, allowing users to modify and expand XR programs as their goals evolve. While we build out GenAssist for ARENA, the core architecture is engine agnostic. We demonstrate this by adapting the GenAssist architecture for XR program generation in Unity.

GenAssist is designed to democratize XR content creation by enabling non-experts to generate small, interactive 3D programs purely using natural language. Rather than replacing professional XR development tools, our system targets rapid prototyping, educational content creation, and experimentation scenarios where ease of use and quick iteration are more important than scene complexity or performance.

To ensure the accuracy and reliability of generated XR code using GenAssist, we incorporate two key techniques:

First, GenAssist employs a **self-correction feedback mechanism** that helps to ensure that the generated XR program matches the intent of the user. The system continuously evaluates the generated program using visual feedback, spatial information, and the state of the program to identify discrepancies between what the user requested and what was produced. When misalignments or inaccuracies are detected, GenAssist automatically refines previously generated code to bring the output closer to the intended result. This feedback loop supports more accurate content generation over time.

Second, it pulls semantically relevant information from platform-specific documentation and relevant code examples using **Retrieval-Augmented Generation (RAG)** to ground LLM outputs. This not only ensures that generated programs are relevant to the user's query but also prevents syntactical and potential runtime errors by providing the LLM with context tailored to the target runtime environment and task. This improves the accuracy of the generated code, especially for ARENA-specific code, which is a rapidly evolving and improving platform.

To evaluate GenAssist, we assess its accuracy and performance in generating a diverse set of XR programs. Specifically, we propose a benchmark of 50 programs of varying complexity, covering object placement, animations, and user interactions, which we use to test the XR output of GenAssist. We compare GenAssist against several baselines, including ablated variants without the feedback loop or retrieval module, as well as previous work on prompt-based XR program generation. Since evaluating an XR program is highly subjective and there could be multiple correct programs, we have human evaluators rate the accuracy and quality of outputs. Additionally, to ground these evaluations, we also report objective correctness checks (e.g., whether expected objects are instantiated, whether the generated code executes without errors etc), which provide a baseline measure of functional validity. GenAssist achieves the highest average rating in all metrics, outperforming standard GPT-4o and other baselines, highlighting the value of our feedback and retrieval mechanisms.

We validate our system through a structured user study with 18 participants, measuring the user experience during XR program creation with our system. The system received a NASA TLX score of 36.6/100 indicating low perceived workload and a system usability score (SUS) of 69.6 indicating good usability.

Finally, GenAssist shows strong efficiency, with an average generation time of 9.93 seconds per query and 14.17 seconds per correction. It also requires significantly fewer LLM calls per program compared to prior systems while producing more accurate results.

In summary, our paper contributes the following:

1. **GenAssist**: an open-source system for generating XR programs from natural language prompts, enabling rapid 3D scene creation and interaction design.[1]
2. A visual feedback loop for LLM generated code refinement to improve the quality of XR programs generated.
3. A user study and system-level analysis that provides information on GenAssist's usability, generation accuracy, and runtime performance compared to existing approaches.
4. A comprehensive evaluation of our system's program generation quality compared to state-of-the-art baselines across 50 diverse prompts, with all prompts and outputs released as a public benchmark dataset.

2 Related Work

2.1 XR Prototyping and Generation

To create 3D scenes and interactive programs, designers have traditionally relied on commercial game engines such as Unity [37] and Unreal Engine [10], which include plugins like MRTK [26] which provides primitives for 3D user interfaces. These platforms often have a high barrier to entry, as developing advanced applications requires skilled developers or experienced game designers. This makes rapid prototyping of 3D scenes or interactive content difficult and time-consuming for non-experts who may only need a small application for a quick 3D visualization. As a result, there is growing interest in democratizing the process of 3D content creation, allowing users to build interactive environments without requiring extensive technical expertise.

Early approaches to the generation of 3D and XR content focused on scene adaptation using predefined rules, semantic reasoning, and domain-specific heuristics. These systems typically modified existing scenes based on spatial or contextual cues rather than generating content from scratch [25, 24, 46]. For example, Cheng et al.'s *SemanticAdapt* [7] adjusts object layouts based on semantic relationships, while Chang et al.'s *SceneSeer* [6] allows natural language prompting to search through existing scenes and models and place them in the scene. Other systems like Xu et al.'s *Sketch2Scene* [42] allow users to sketch 3D scenes, relying on model training and visual templates to construct environments. Adobe's Aero [2] enables interaction with AR elements through a GUI, reducing the need for programming. Although these systems provide more intuitive authoring tools than scripting in a game engine, they typically only support static scenes and have limited ability to generalize across domains. In addition, they lack support for open-ended user interactions with the scene objects.

Today with large language models and generative AI, emerging systems have begun to support text-driven 3D scene creation, providing a more natural interaction modality compared to scripting or graphical user interfaces. Diffusion-based methods [32, 23, 19, 14] generate 3D assets from natural language, focusing primarily on

[1]The code can be found at https://www.srutisrinidhi.com/GenAssist/

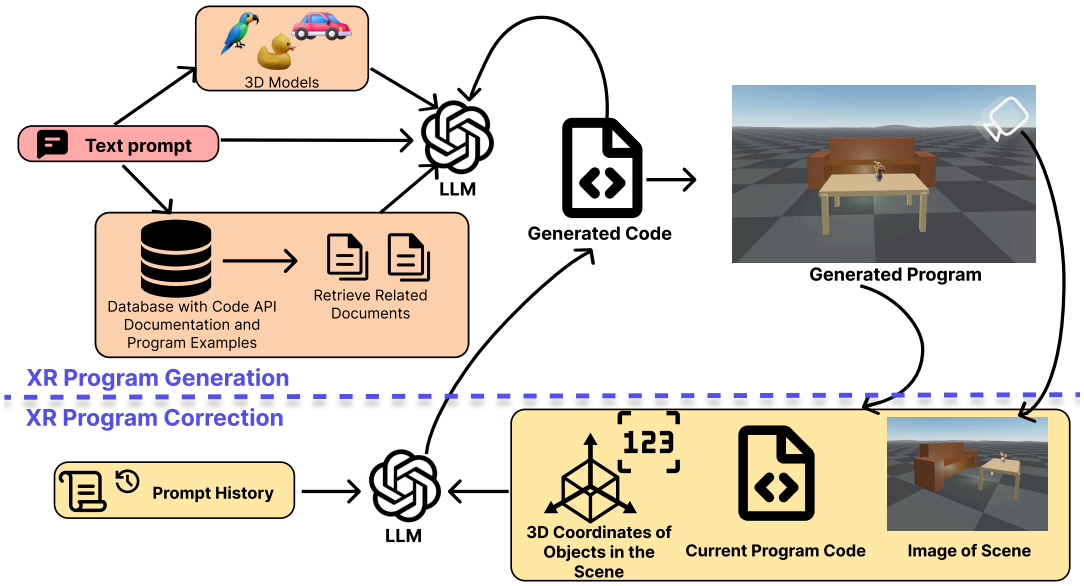


Figure 2: **GenAssist system architecture.** GenAssist comprises of two main stages: program generation and program correction. In the generation stage, a user-provided text prompt is used to retrieve relevant ARENA platform documentation, API usage examples, and 3D models using RAG techniques. These are passed to a large language model (LLM) to generate code, which is then executed in ARENA's Python runtime to produce the XR program. In the correction stage, the system gathers the current program code, 3D object coordinates, and an image of the scene from a virtual camera, along with the user's prompt history. This context is used by the LLM to make code corrections as needed.

mesh generation. Though powerful for object synthesis, these approaches are typically limited to rigid, static models and do not extend to interactive or programmable XR content. Similarly, Google has been releasing a series of world building models such as Genie [11] that can create high-fidelity 3D worlds and have agent interaction, but are still static and only support limited interactions.

Scene generation and editing systems using LLMs have also been rapidly advancing. For instance, Qian et al. introduced *SHAPE-IT* [33], which leverages LLMs to translate user prompts into code that generates shapes on pin-based shape displays. Closely related to GenAssist are systems such as *3D-GPT* [36] and *BlenderAlchemy* [16], both of which generate Blender-compatible Python code to synthesize and edit existing geometry and materials. *3D-GPT* emphasizes reasoning and task decomposition from textual instructions, whereas *BlenderAlchemy* integrates visual processing capabilities, enabling the LLM to interpret both text and image inputs for a more interactive and multimodal approach to 3D content generation. Building on these efforts, *SceneCraft* [15] introduces a self-correction mechanism that iteratively refines generated scenes to address errors. GenAssist builds upon the ideas presented in these works to create *interactable* scenes using existing 3D models and object primitives. Specifically, we make use of a scripting API and runtime that provides straightforward and accessible methods for scene manipulation that an LLM can leverage.

Although LLMs have been applied to XR content generation, many existing approaches are tailored to highly specific use cases rather than general-purpose scene creation. Applications such as *VR Copilot* [47] and *Holodeck* [43] leverage LLMs to specifically generate room layouts in Unity. Other applications explore this in the context of video games by dynamically generating in-game objects using text or player inputs [18, 34]. These systems show the potential of LLMs for 3D scene modification, but lack the adaptability needed for general-purpose, open-ended XR creation.

In the domain of interactive XR program generation, multiple systems have leveraged LLMs to create interactable content. Giunchi et al. with *DreamCodeVR* [12] present a simpler approach to generating objects, performing a LLM call to generate code for object creation. However, it lacks an iterative refinement process and exhibits limited complexity in object outputs. De La Torre et al.'s *LLMR* [9], a recent system for XR program generation, employs a pipeline of multiple LLM calls to plan, analyze, build, and refine scene generation outputs, which we discuss in Section 4.2 as a recent example of multistage XR program synthesis using LLMs. Conceptually, our approach aligns with broader efforts like PAIL [45], which reframe LLM-based programming as a design activity involving iterative exploration and decision tracking.

2.2 RAG for Code Generation

RAG enhances the capabilities of LLMs by supplementing their internal knowledge with relevant external context. Initially developed to improve accuracy and reduce hallucinations in natural language tasks [21], RAG retrieves relevant documents or code snippets from a corpus and incorporates them into the generation process. This is especially useful in domains where training data alone may be insufficient, enabling dynamic access to up-to-date or domain-specific information. RAG has also been adapted for multimodal tasks, such as image generation with retrieval-augmented diffusion models [3] and 3D content generation [35].

In code generation, RAG has shown promise but also presents challenges. Wang et al. introduce *CodeRAG-Bench* [39], a benchmark evaluating RAG-based code generation, noting that while retrieved context can improve results, models often struggle to integrate semantically relevant but lexically mismatched content. Large-scale systems such as *CodeRetriever* [22] demonstrate the effectiveness of unimodal and multimodal retrieval strategies for code synthesis and search. Systems such as *REDCODER* [29] retrieve and incorporate various code–text pairs, and others have explored the retrieval for specialized domains, including the generation of RTL codes for hardware design [20, 40], the enhancement of code security [28] and automated bug fixing [38]. However, studies show that irrelevant retrieval can negatively impact performance, highlighting the importance of high-quality retrieval strategies [44].

Program Generation Prompt=
You are a 3D XR program generation assistant. Based on the following context, generate code to create the specified program:

1. Relevant Documentation and Examples:
{retrieved_docs_and_examples}

2. Available 3D Models:
{model_urls}

3. Current Scripts that are Running:
{current_running_scripts}

4. History of prompts (first one is the most recent):
{prompt_history}

Generate the code considering the above context for the question that follows: **{question}**

(a) Program Generation Prompt

Program Correction Prompt=
Based on the following context, decide whether to modify the program or not. If you decide to modify it, provide the python script of the corrected program:

{image_of_scene.png}

Current Script Running in the scene:
{current_running_scripts}

Scene Object Bounding Boxes [format = objectid: min: minimum xyz coordinate, max: maximum xyz coordinate] (if the bounding box is inf that means the model is bad and should be replaced):
{bounding_boxes}

History of Prompts(first prompt is the most recent):
{prompt_history}

Use the image and the information to correct the program.

(b) Program Correction Prompt

Figure 3: Template prompts used to provide additional context for program generation and program correction.

Building on these foundations, GenAssist applies RAG to XR program synthesis, a novel extension of retrieval-augmented code generation to 3D and interactive environments.

3 System Design

In the following subsections, we outline how GenAssist generates XR programs from natural language for ARENA (a WebXR runtime platform), how it employs a feedback loop to iteratively refine its output, and how Retrieval-Augmented Generation (RAG) techniques are applied to enhance code generation. An overview of the system architecture is shown in Fig. 2, with descriptions of each component described below.

3.1 Code Generation for ARENA

ARENA provides a dedicated Python API, `arena-py` [41], and a Python runtime. All program I/O passes through a scene graph that is overlaid on a pub-sub backend. The choice of Python is particularly advantageous: It is one of the most widely used programming languages with a vast ecosystem of publicly available code. Thus, most commercial large language models (LLMs) such as GPT-4o [1] have been trained on large amounts of Python programming data, making these models well suited to generate syntactically correct and directly executable Python code.

Although ARENA is primarily web-based, its programs are inherently cross-platform and distributed, meaning they can run on any device with a network connection and are not limited to a specific viewing environment. In fact, all our experiments were conducted using a browser-based viewing app, though ARENA also supports a Unity-based viewer. It is important to note that while our implementation targets ARENA, the core methodology generalizes to any XR platform. The visual feedback loop requires only the ability to capture screenshots and extract object spatial information, capabilities available in Unity, Unreal Engine, and other platforms through their respective APIs. Similarly, our RAG approach can incorporate documentation and examples from any platform's ecosystem. ARENA was selected primarily for its hot-pluggable execution model, which facilitates rapid iteration during development and evaluation. Additionally, we do show that our system techniques can be used for other platforms and demonstrate this for Unity in Section 6.

GenAssist *only* synthesizes the `arena-py` code, the ARENA runtime handles event dispatch, scene synchronization, and execution. As of this submission, the interactions exposed through ARENA to `arena-py`, and hence the interactions that GenAssist can use in the programs it generates, consists of (a) pointer/cursor events (mouse or controller interactions), (b) proximity-based interaction, and (c) a limited set of keyboard events. This is a platform boundary rather than a limitation of our method, and so on runtimes with richer inputs, GenAssist can target those events as long as documentation and examples are available for retrieval.

3.2 Iterative Scene Correction Feedback Loop

Although LLMs have demonstrated strong capabilities in generating code across a wide range of domains, they are not error-free. LLMs can still produce syntactically invalid code, incorrectly use APIs, or introduce logical or physical inconsistencies, which affects the quality of generated XR programs. To mitigate these issues and improve reliability, GenAssist incorporates a visual and textual feedback loop inspired by how developers typically refine XR applications: writing code, observing the resulting scene, and iteratively adjusting based on visual output as seen in Fig. 2.

To replicate this workflow, GenAssist periodically captures 2D screenshots of the scene, taken from strategic camera positions that provide a comprehensive view of all objects in the environment. To capture these images, we render the ARENA scene in a browser window using Playwright [27], a browser automation framework, and take screenshots of the generated program. To guarantee that the entire scene is captured in the image, the system automatically computes the 3D bounding boxes of all objects and determines an appropriate camera position that ensures that all objects are within view. This visual feedback allows the model to "see" what has been generated, enabling it to detect issues that may not be obvious from the code alone. This process can be executed on a separate machine, ensuring that the user's performance remains unaffected.

Additionally, GenAssist feeds the 3D bounding box coordinates of all the objects to the program correction module. This helps the LLM place objects in physically plausible locations by providing an understanding of spatial relationships within the program, which are not always evident from 2D screenshots, such as relative object sizes and placements. This is especially important when incorporating external objects and models in ARENA, as the generated code only references a file server URL (e.g., `https://arenaxr.org/fakeuser/mymodels/model.glb`) without specifying the underlying geometry. As a result, the system cannot determine the model's relative size or shape until it has been rendered in the scene. This spatial information helps the LLM understand a given 3D model's positioning, scale, and potential overlaps with other objects, providing essential context for reasoning about spatial relationships within the 3D environment. Alongside visual and spatial feedback, the loop also includes the current generated code and a full history of previous prompts to ensure that the model remains aware of both the intended goals and the program's evolution over time.

Lastly, to improve robustness, the system captures and feeds any detected errors, including syntax issues, runtime exceptions, or execution logs, into the correction loop. This loop iteratively refines the scene by prompting the LLM to fix identified problems, correct object placements, and adjust the program to better align with the intended design. Through this continuous cycle of feedback and correction, GenAssist ensures that the generated XR programs become increasingly accurate and functional over time. The template for the data provided as the context for the scene corrector is

also shown in Fig. 3 and the full prompt is in the supplementary material. This is one of the key features that allows for iterative development, where a person's prompt can reference previous actions, the current scene, and program state, to build and modify a program using multiple sequential prompts.

3.3 XR Program Generation with Retrieval-Augmented Generation

To enable GenAssist to generate code that follows ARENA's syntax, structure, and feature set, we supplement the LLM with additional context in the form of curated examples and API documentation. Although LLMs are powerful, it is well known that they can hallucinate and produce incorrect responses, especially when they lack up-to-date or domain-specific knowledge [17]. This is especially true for rapidly evolving platforms like ARENA, where APIs and conventions frequently change, or for complex ecosystems like Unity and Unreal Engine, where the breadth of features and multiple engine versions can make it difficult for the LLM to determine which details are most relevant to a given question. Previous work like LLMR [9] handled this using another LLM call to identify what information needs to be provided as a context, which can be expensive and time consuming. To address this, we used a lightweight RAG-based approach to improve the model's ability to produce accurate and relevant code. At a high level, GenAssist generates code by prompting the LLM with a selected blend of documentation, usage examples, and the current state of the program.

A key component of this process is retrieving relevant content from ARENA's API documentation and example code. By scraping documentation webpages and code repositories, we create a vector database of semantic embeddings mapped to text. When a user queries GenAssist, it will tokenize the prompt and retrieve the closest matches from the database. The text of those matches is then injected into the prompt before being sent to the LLM for RAG. By including documentation and examples directly in the prompt, the system leverages in-context learning [5], which is a known strength of LLMs, allowing the model to better follow API usage patterns, coding style, and program structure. We found that this improves the accuracy of the generated code and reduces hallucinations.

Furthermore, ARENA includes a public file server of 3D models (in `.gltf/.glb` and `.obj` formats). Using a similar technique as described above, GenAssist creates a vector database using embeddings created from file metadata that can be queried using RAG. For each user query, we search the vector database to find any 3D models that might be useful for the LLM to use when generating the program, allowing for easy model integration into the system.

To maintain program continuity and context, the system also feeds the current running XR program directly into the prompt. This allows the LLM to understand what is already present in the scene and allows it to directly edit the current program. Furthermore, GenAssist appends a history of prior prompts and responses to preserve the flow of user intent and interactions over time. This running context ensures that the model's generation aligns with both the current scene and the user's iterative design process. We provide the template for the context provided to the LLM in Fig. 3.

Our system uses GPT-4o [1] as the LLM and Chroma-db [8] as the RAG vector database. We use OpenAI's *text-embedding-ada-002* embedding models to embed ARENA documentation and `arena-py` examples, with each page or example being an entry into the database. We also use the same embedding model for the 3D models, which we save in a separate chroma-db database.

4 Evaluation

To be effective, GenAssist must be easy to use, generate plausible and intent-aligned XR programs, and operate with low overhead. We evaluate the system across these three key dimensions: (1) User Experience – Does GenAssist lower technical barriers and enable successful task completion in XR program creation? (2) Generation Quality – Does it produce XR programs that are both accurate and consistent with user intent? (3) System Overhead – What is the runtime cost of the system, measured in terms of latency and the number of LLM queries required? [2]

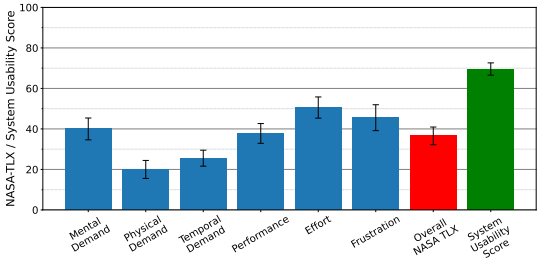


Figure 4: Average Usability Scores across all participants. A lower NASA TLX score indicates lower perceived workload (lower is better for blue and red), while a higher System Usability Score indicates better usability (higher is better for green).

4.1 User Experience

To evaluate the user experience of GenAssist, we conducted a study with 18 participants (11 male and 7 female between the ages of 18 and 55). Only 5 of the participants had experience with building XR programs before and 13 were new to XR program generation, which is the target audience for our system. Each participant received a brief introduction to the system and its functionalities and was given five minutes to freely explore GenAssist by generating their own programs and exploring the system's capabilities. After the familiarization phase, participants were presented with a pre-generated *target* program and given time to interact with it and explore its functionality. They were then asked to use GenAssist to reproduce the program. They were allowed to make as many prompts as they wished until they were satisfied with the result. Participants performed this task twice, each time with a different target program representing a distinct level of difficulty. Both programs were designed to evaluate how effectively users could achieve specific goals with the system. To ensure consistency, all participants attempted the same two programs (included in the supplementary material).

After completion of the tasks, the participants were given a short survey comprising of the NASA Task Load Index (NASA TLX) [13] and the System Usability Scale (SUS) [4]. The specific XR programs used for testing and the survey questionnaires are included in the supplementary material for reference.

Fig. 4 presents the usability results, including the six NASA TLX metrics, the overall average NASA TLX workload score, and the SUS score. Lower NASA TLX score indicate reduced workload and effort. Our results show a NASA TLX overall score of **36.6 out of 100**, suggesting that using GenAssist imposes a relatively low cognitive load. Among NASA TLX metrics, *performance* had the highest score (indicating a higher perceived difficulty in being successful in the task), which aligns with participant feedback. Users reported that generating correct output sometimes required multiple iterations of prompting, particularly for complex programs. This iterative process increased the perceived effort to successfully complete the task, although the participants acknowledged that this would be much easier than writing code to generate these programs.

In contrast, the System Usability Scale (SUS), where a higher score indicates more usability, yielded a score of 69.6 for GenAssist. With 68 generally considered average, this suggests that the system is reasonably usable. Although SUS provides a useful overall measure of perceived usability, we place greater emphasis on the NASA TLX results for this evaluation. NASA TLX captures a

[2]The user studies in section 4.1 and 4.2 were determined as not human research by the Carnegie Mellon University IRB (STUDY2025_00000065).

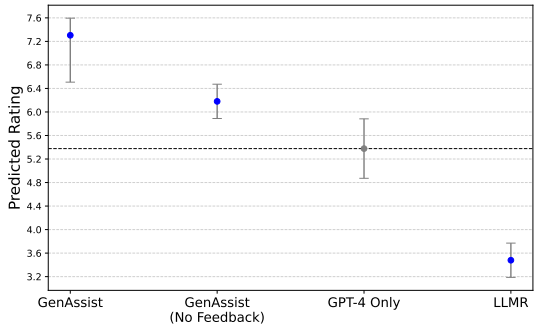


Figure 5: Estimated system effects from the linear mixed effects model with 95% confidence intervals. Coefficients represent the actual rating values predicted by the model. Higher the value, higher the rating which is out of 10.

broader picture of user experience by measuring perceived workload across dimensions such as mental demand, effort, and frustration. This makes it particularly well-suited for evaluating systems like GenAssist, where user effort can be affected not just by the interface but also by the behavior of the underlying model. For example, instances where users needed to re-prompt the system to refine outputs contributed directly to higher perceived workload. As such, the NASA TLX offers a more comprehensive understanding of how challenging or demanding the system is to use in practice.

4.2 Generation Accuracy

Our goal is to evaluate the accuracy of GenAssist in generating XR programs. However, defining what constitutes an "accurate" or a "good" output in this context is inherently subjective. For instance, a prompt such as "make a car" could be satisfied by a simple 3D car model, by a basic construction using a rectangular cube with four cylinders as wheels, or by a more detailed, higher-fidelity car composed of primitives with windows, realistic proportions etc.

While prior work has reported accuracy and error metrics, the underlying evaluation methods can be unclear or insufficiently detailed. To address this gap, we conducted a study where participants (referred to as raters) evaluated the correctness of XR programs generated from their prompts. Each program was assessed by raters along four distinct metrics, described below, using a 10-point scale.

Program Correctness Metrics:

1. **Prompt Match**: How closely does the XR program align with the input prompt?
2. **Object Placement**: How well are the objects positioned within the scene?
3. **Functionality**: Does the program behave as expected?
4. **Overall Quality**: What is the overall perceived quality of the experience?

For comparison, we compare four XR generation systems: (1) **GenAssist**, (2) **GenAssist (No Feedback, RAG Only)**, (3) **GPT-4o only (No Feedback, No RAG)**, and (4) **LLMR** [9], a recent state-of-the-art system for Unity program generation.

We curate a set of 50 prompts across five categories of increasing complexities. We create a taxonomy of prompts based on the skills and techniques we expect a non-expert user to use when generating simple XR programs. Each category consists of 10 prompts. Some are drawn directly from the LLMR evaluation set, while others were newly created to reflect practical tasks users might attempt or to highlight the capabilities of the ARENA platform. The full set of prompts is provided in the supplementary material. This taxonomy consists of: (1) **Object Placement** (2) **Animations** (3) **Interactivity**, (4) **Complex Programs Combining Multiple Features**, and (5) **Iterative Program Generation**.

For the study, we recruited 15 raters with varying levels of familiarity with XR. Each rater evaluated the output of 40 prompts (distributed across the 4 systems and 5 program categories), with each prompt's output assessed by three different raters. Prompt assignments were randomized: some raters evaluated the same prompt across all systems, while others reviewed different prompts and systems. This design was intended to mitigate potential bias and ensure a diversity of scoring perspectives.

Given multiple sources of variability (system, prompt, metric, rater), we use a linear mixed-effects model [31] to estimate system effects while controlling for prompt difficulty and rater stringency. Systems and the program correctness metric are modeled as fixed effects while prompts and raters receive random intercepts, controlling for confounding factors like prompt difficulty or rater bias. This lets us attribute differences in ratings to the systems and metrics rather than variations in specific prompts or raters. We define our model as follows:

$$Rating_{i,j,k} = \beta_0 + \beta_{\text{System}} \cdot System_{i,j,k} + \beta_{\text{Metric}} \cdot Metric_{i,j,k} + u_i + v_k + \varepsilon_{i,j,k} \quad (1)$$

Where:

i, j, k : Individual rater index, Rater ID, prompt index.
$\beta_0 = 5.377$: Intercept term (baseline for *GPT-4o Only* system).
β_{System} : Fixed effect for system type (GenAssist: 7.304 ; GenAssist (No Feedback): 6.180; LLMR: 3.479; all $p < 0.001$).
β_{Metric} : Fixed effect for metric values.
u_i, v_k : Rater-level and Prompt-level random intercept.
$\varepsilon_{i,j,k}$: Residual error.

The baseline condition in our mixed-effects model corresponds to the GPT-4o Only system. As shown in Fig. 5, the full GenAssist system outperforms its ablated variants and baselines, achieving a predicted rating of 7.304, compared to 6.180 for GenAssist without feedback, 5.377 for GPT-4o, and 3.479 for LLMR. These results provide strong evidence that the closed-loop GenAssist system produces more reliable and higher-quality outputs. In particular, they underscore the effectiveness of incorporating a feedback mechanism to identify and correct hallucinations, and the use of RAG to enhance robustness across a wide range of task complexities.

We acknowledge that comparisons between GenAssist and LLMR are not direct for several reasons. First, LLMR was developed for Unity C#, whereas GenAssist generates Python code for ARENA. While Unity is more prevalent in XR development, Python is more widely represented in LLM training data overall, creating different baseline capabilities. Second, LLMR uses GPT-4 while our system leverages GPT-4o. Updating LLMR to use GPT-4o would require non-trivial modifications to its multi-stage prompt pipeline and Unity-specific examples, potentially compromising the integrity of the original system design. Therefore, we include LLMR as a contextual comparison to generally understand where our system stands compared to prior work, rather than a statistically significant performance claim. Our core contributions—specifically, the effectiveness of RAG and visual feedback—are validated through ablation studies on GenAssist variants, which enable controlled comparisons under identical infrastructure.

Objective Correctness and Functional Validity

In addition, to ground our evaluation, we report objective correctness checks by evaluating the following 5 boolean questions for each of the 50 prompts generated by the 3 primary systems:

1. Do all expected objects exist in the scene?
2. Are they at expected locations?
3. Do they have expected animations?
4. Do the interactions trigger as expected?
5. Does the code execute without errors?

For each one, we check if the objects exist, whether they are in the correct relative position to each other and the ground given the prompt, whether the right objects are animated that can broadly match the description, whether animations get triggered with the right interaction, and whether the code can execute without crashing

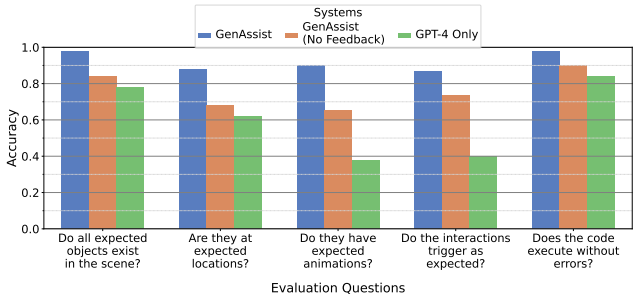


Figure 6: Objective correctness results. Bars show the proportion of prompts for which each system satisfies a set of boolean correctness checks (e.g., expected objects instantiated, code executes without errors etc.). These metrics complement the subjective human ratings by providing a baseline measure of functional validity.

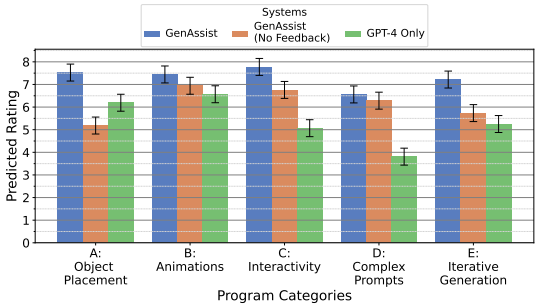


Figure 7: Accuracy of outputs across prompt categories for each system. Reported values are z-scores, obtained by normalizing ratings within each rater.

System Components	***Average Time Taken (s)***
Program Generation	**9.928**
Retrieval of Documentation and Examples	4.414
Retreival of 3D Models	0.478
Queries to LLM (GPT-4)	5.036
Program Correction	**14.173**
Get bounding boxes of objects in the scene	0.152
Get screenshot of the scene	3.792
Queries to LLM (GPT-4)	10.229

Table 1: Average time taken for the Program Generation and Program Correction stages.

Program Category	***Average Number of LLM Queries***
Object Placement	1.90 ± 2.07
Animations	2.10 ± 1.58
Interactivity	2.10 ± 1.22
Complex Programs Combining Multiple Features	3.10 ± 2.98

Table 2: Average number of calls to LLMs needed to generate the program for the first 4 categories of programs. The first query corresponds to the prompt the user sends to the program generator, and any subsequent ones refer to the queries made to the program corrector. The fifth category (*Iterative Scene Generation*) is not evaluated as the number of prompts used as input is not constant.

or throwing errors. The idea is to get a baseline sense of how well the different systems generate XR programs while leaving the fine-grained detailed evaluation to the subjective ratings above. Figure 6 shows that across all categories, GenAssist consistently satisfies a higher proportion of objective criteria than both the RAG-only variant and GPT-4o. While these metrics do not capture experiential quality, they provide a baseline measure of functional validity and mirror the trends observed in the subjective evaluations.

Category-wise Performance Analysis

To further investigate how the performance of the different systems vary across prompt categories, we model the data with another mixed-effects model specified as follows:

$$Rating \sim System \times Category + Metric \quad (2)$$

This model allows us to gain insight on how the ratings vary for different systems for the different prompt categories A through E, shown in Fig. 7. We exclude LLMR from this category-wise analysis as it serves as a contextual comparison rather than a direct baseline. Across all categories, GenAssist consistently outperforms its ablated variants.

Interestingly, for simpler prompts (Category A), the RAG-only variant, *GenAssist (No Feedback)*, underperforms. Upon closer analysis, we found that this was often due to irrelevant retrievals. For simpler tasks, the language model and the few-shot examples in the prompt are usually sufficient to generate correct code. However, when extraneous documents are retrieved, the model tends to over-prioritize them, leading to hallucinated or incorrect outputs. This issue stems from the model's strong bias toward its immediate context [44], causing it to pay attention to less relevant information even when simpler logic would suffice.

Thus, in isolation, RAG may not be as helpful and can even hinder performance in basic tasks like category A. However, when combined with closed-loop feedback, the system can identify and recover from these errors, preserving strong performance even on simple prompts. For more complex prompts (Categories B–E), the additional retrieved context becomes significantly more useful, as it provides the model with implementation patterns not covered by the initial prompt itself. This demonstrates that retrieval is particularly effective when prompt complexity increases, but must be paired with mechanisms like feedback to ensure robustness across the full range of task difficulty.

4.3 System Overhead

To evaluate the run-time cost of GenAssist, we measured both system latency and average number of LLM queries required for program generation and correction. These factors directly impact the system's responsiveness and practicality for interactive use.

Each generation cycle introduces several sources of latency. For every user prompt, the system performs a retrieval of relevant documentation and example code, as well as 3D models to construct the LLM prompt context. Although this step adds some delay, the most significant overhead comes from the LLM response time during code generation. Once generated, running the program has negligible overhead due to ARENA's hot-pluggable execution model. The scene correction loop also introduces additional latency. Capturing scene screenshots and extracting 3D bounding boxes of all the objects in the scene adds runtime costs. However, as with the generation cycle, the LLM response time is the dominant factor.

Since the number of generation and correction cycles needed vary based on prompt complexity and the number of times the corrector needs to be called, we report these costs separately. Specifically, we measure (1) the average time taken for initial program generation, (2) the average time per correction cycle, and (3) the average number of LLM queries required per program.

Our analysis focuses on four of the five prompt categories used in the generation quality evaluation in Sec. 4.2. We exclude the *Iterative Scene Generation* category because it involves multiple user inputs by design, making it difficult to compare directly with the other categories.

Tab. 1 summarizes the system latency for various components of GenAssist. Because LLM calls are the main driver of latency and cost, we report the average number of LLM calls needed to generate XR programs for the different prompt categories in Tab. 2. Simple prompts generally require only a single LLM query, with no corrections needed to produce a correct response. In contrast, more complex prompts, particularly those involving animations or interactivity, often need multiple correction cycles before a satisfactory program is generated. In addition, the number of lines in the program, which corresponds to the length of the LLM output, directly impacts generation time. Therefore, our reported values represent averages across varying levels of complexity and program lengths. The reported number of LLM queries reflects the total across both generation and correction stages. Since we exclude the *Iterative Scene Generation* category, there is only one query for program generation and any additional queries are from the scene corrector. In general, our system takes **less than 10 seconds** to generate the program code with each iteration of the corrector taking **14.2 seconds**. Overall, even under the assumption that GenAssist requires on average one LLM call for program generation and an additional call for program correction, the **total wait time of our system is 24.10 seconds, which is approximately 3.7× lower than 90.98 seconds taken by LLMR, the current state of the art.**

It is important to note that determining when a program is "complete" is inherently subjective. Our reported averages reflect the observed number of LLM queries during evaluation and serve as ballpark estimates of system overhead rather than strict upper bounds.

5 Applications

GenAssist can have a wide range of use cases, spanning both practical scenarios and open-ended creative tasks. In this section, we highlight several example applications that demonstrate how the system can be used to build interactive 3D experiences in domains such as education, remote assistance, and entertainment.

5.1 Interactive Educational Content

GenAssist can be used to create immersive and interactive educational experiences in 3D, ranging from intractable visualizations of educational concepts to more structured tools like labeled 3D models, flashcards, and interactive quizzes. For example, a biology instructor could generate a scene which includes a model of a plant with clickable parts that reveal descriptions, or a history teacher could create a virtual exhibit that students can explore. Because most educators are not experts in XR programming, tools that allow them to easily author and modify content can be extremely useful in the classroom to quickly drive up excitement. Interactivity is especially important for engagement and comprehension in learning environments, and GenAssist allows educational content to be created and adapted dynamically based on the needs of the learner. This opens the door to personalized and responsive educational tools without requiring specialized XR knowledge. An example scene can be seen in Fig. 8.

5.2 Creative Exploration and Game Design

GenAssist enables novice users to create simple games and experiment with interactive 3D content without needing to write code. It functions as a creative playground, allowing users to explore the possibilities of XR programming through natural language prompts. One such example can be seen in Fig. 9.

5.3 Remote Assistance

In current remote assistance scenarios, such as helping someone troubleshoot equipment through VR streaming, guidance is typically limited to voice communication, which can be ambiguous or hard to follow during complex or multi-step tasks. While some systems now support remote annotations, these features are often

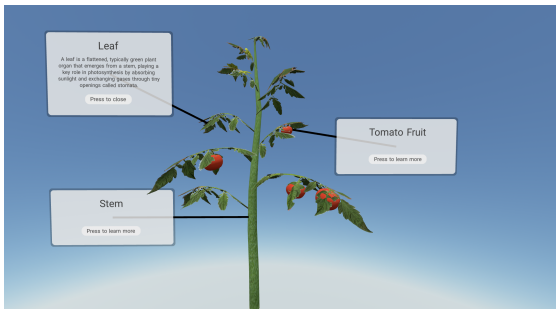


Figure 8: Example usage of GenAssist to create educational 3D programs. Here the plant is annotated with clickable cards to learn about various parts of a plant in an interactive and engaging way.

Figure 9: Example usage of GenAssist to create a animated dragon that breathes fire next to a castle.

constrained to basic markup and lack the flexibility to add richer or more interactive content within the scene. With GenAssist, a remote assistant can instead create 3D annotations, overlays, or interactive elements directly within the user's XR environment. Because GenAssist operates in natural language, these can be made on the fly, making remote support sessions more effective, interactive, and intuitive. For example, during a filter-replacement procedure, the expert says: "Create a floating checklist titled 'Filter Replacement', highlight the intake valve, draw an arrow to the release latch, load `filter.glb` and place it over the target socket, add a 'Next' button that advances the checklist when clicked." GenAssist generates the ARENA code to create these programs, reducing ambiguity compared to voice-only guidance.

5.4 Guided Task Assistance

GenAssist can be used to create XR programs that assist with physical-world tasks by providing contextual visual guidance. For example, it can generate annotated 3D scenes that illustrate how to operate a device or perform a step-by-step procedure. These virtual guides can include arrows, labels, and interactive elements to help users follow along more effectively. By lowering the barrier to authoring such assistants, GenAssist makes it easier to create customized instruction manuals and AR overlays without requiring technical expertise. It also enables the assistance to adapt to the user's actions for more responsive and personalized guidance.

6 Cross-Platform Generalizability

While our primary implementation and evaluation target ARENA, the core generation–observation–correction loop is designed to be engine-agnostic. We selected ARENA as our primary testbed due to its lightweight, hot-pluggable Python execution model, which is particularly well suited for rapid iteration, LLM-based code generation, and controlled evaluation. To validate this

architecture beyond ARENA, we implemented a preliminary Unity backend that executes generated C# scripts, captures scene screenshots, and extracts object-level spatial metadata such as positions and bounding boxes. Figure 10 illustrates a prompt ("Make a snowman that is made of three snowballs of different sizes and has a hat on its head") being executed in both ARENA and Unity, producing structurally similar outputs despite differences in programming language and runtime environment. The Unity backend exposes the same observation signals used by GenAssist in ARENA, i.e. rendered images, spatial metadata and code, which are sufficient to support iterative correction through a scene correction loop. Details of the process are below:

Initial Generation Errors: In both engines, the initial output contained structural errors: the snowballs were vertically compressed, and the hat was incorrectly positioned relative to the top.

Correction Mechanism: GenAssist captured the scene state, specifically the 3D bounding boxes and rendered images, from both environments. Despite ARENA running Python and Unity running C#, the feedback loop detected the same spatial inconsistencies, and edited the positions of the objects in the scene. After one self-correction cycle, the system adjusted the vertical offsets in both programs, resulting in properly stacked spheres and a centered hat.

Additional Prompt Iteration: A follow-up prompt ("Add eyes and a smile made of buttons") was successfully processed by both stubs, demonstrating that the iterative state management is not platform-dependent.

Currently, this is not a complete implementation, but just a stub to suggest this architecture can be extended to other engines as well. This experiment demonstrates that GenAssist depends only on general XR engine capabilities rather than ARENA-specific APIs, providing an existence proof for extending GenAssist to other high-fidelity engines like Unity or Unreal.

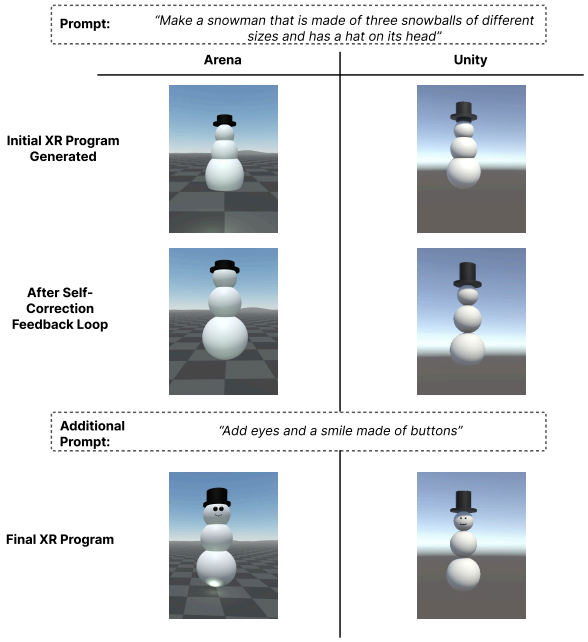


Figure 10: Preliminary evidence of cross-platform generalizability. A single prompt is executed in ARENA (Python) and Unity (C#). Despite different runtimes, both engines provide very similar observation signals (rendered images and spatial metadata) to the feedback loop, enabling successful iterative refinement across different runtimes. The bottom row demonstrates successful iterative refinement (adding features) within the Unity backend.

7 Discussion

Although GenAssist enables the generation of natural language-driven XR programs with iterative refinement, it does have limitations. The system's feedback loop relies on visual observations of the current scene, specifically a rendered image, the 3D bounding boxes of all objects, and the current program code. In addition, it incorporates any run-time errors or stack traces that appear in the execution logs. This setup allows the model to detect and fix issues related to object positioning, sizing, visibility, or syntax and runtime errors. However, our objective checks revealed that behavioral issues often persist: while visual-related errors occurred in 5/50 prompts, errors in animations (4/37) and interaction triggers (4/30) were more frequent. Such behavioral issues are only detected when they produce explicit errors or visible mismatches, for example, when an animation or interaction fails silently unless it results in a visually incorrect scene or raises an exception. As a result, some behavioral errors may go undetected. Future extensions could incorporate interaction traces or lightweight behavioral checks to move beyond purely visual verification.

Another challenge is reliance on embedding-based retrieval for API documentation and examples. While RAG improves LLM responses by retrieving semantically similar examples, it often retrieves documents that share keywords with the prompt rather than reflecting its compositional or structural intent. For example, a prompt like *"create a car and make it drive"* can retrieve results related to cars or driving animations without providing the primitives or logic needed to combine them into a coherent scene. Improving retrieval may require more task-specific representations or retrievers trained to support such tasks.

GenAssist targets accessibility and rapid prototyping rather than production-level XR applications. While the generated programs handle object placement, animations, and basic interactions effectively, more complex scenarios (e.g., advanced physics simulations, custom materials etc.) would still require traditional development approaches or further advancements in the system.

Finally, the current 3D model search approach involves a trade-off between ease of use and retrieval quality. We initially explored using the Sketchfab API for dynamic model search, but its keyword-based matching often produced results that were inconsistent or poorly suited to the prompt. To improve reliability, we instead pre-downloaded a curated set of models and indexed their metadata using embeddings. While this manual curation ensures reliability, we acknowledge the scalability tradeoffs and implications for broader adoption compared to live API search. Although this process is more manual, it yielded significantly better results in practice. To support scalability, we built our Sketchfab integration to be extensible and will share it in our open-source code releases to allow the community to develop alternative retrieval strategies.

8 Conclusion

We present GenAssist, a system for generating XR programs from natural language prompts by combining retrieval-augmented generation, a visual feedback loop, and hot-pluggable code execution. It allows users to iteratively build scenes that include object placement, simple animations, and basic interactivity, without writing code manually. Compared to existing systems, GenAssist achieves higher output accuracy and significantly lower latency, making it well suited for rapid XR prototyping. By reducing the technical barriers to creating and modifying XR content, it offers a practical tool for both novices and experienced developers.

Acknowledgments

This work was supported by the NSF under award CNS-1956095, the NSF Graduate Research Fellowship under award DGE-2140739 and Bosch Research. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

References

[1] Openai gpt-4o. https://openai.com/index/hello-gpt-4o, 2024. Online. Accessed: July 2024. 4, 5

[2] Adobe. Adobe aero. Accessed: 2025-03-12. 2

[3] A. Blattmann, R. Rombach, K. Oktay, J. Müller, and B. Ommer. Retrieval-augmented diffusion models. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, eds., *Advances in Neural Information Processing Systems*, vol. 35, pp. 15309–15324. Curran Associates, Inc., 2022. 3

[4] J. Brooke. *SUS – a quick and dirty usability scale*, pp. 189–194. 01 1996. 5

[5] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners, 2020. 5

[6] A. X. Chang, M. Eric, M. Savva, and C. D. Manning. Sceneseer: 3d scene design with natural language. *arXiv preprint arXiv:1703.00050*, 2017. 2

[7] Y. Cheng, Y. Yan, X. Yi, Y. Shi, and D. Lindlbauer. Semanticadapt: Optimization-based adaptation of mixed reality layouts leveraging virtual-physical semantic connections. UIST '21, p. 282–297. Association for Computing Machinery, New York, NY, USA, 2021. doi: 10.1145/3472749.3474750 2

[8] Chroma. Chroma - the open-source embedding database, 2023. Accessed: 2025-03-30. 5

[9] F. De La Torre, C. M. Fang, H. Huang, A. Banburski-Fahey, J. Amores Fernandez, and J. Lanier. Llmr: Real-time prompting of interactive worlds using large language models. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, pp. 1–22, 2024. 3, 5, 6

[10] Epic Games. Unreal engine. 2

[11] P. J. B. et al. Genie 3: A new frontier for world models. 2025. 3

[12] D. Giunchi, N. Numan, E. Gatti, and A. Steed. Dreamcodevr: Towards democratizing behavior design in virtual reality with speech-driven programming. In *2024 IEEE Conference Virtual Reality and 3D User Interfaces (VR)*, pp. 579–589, 2024. doi: 10.1109/VR58804.2024.00078 3

[13] S. G. Hart and L. E. Staveland. Development of nasa-tlx (task load index): Results of empirical and theoretical research. *Human mental workload*, 1(3):139–183, 1988. 5

[14] L. Höllein, A. Cao, A. Owens, J. Johnson, and M. Nießner. Text2room: Extracting textured 3d meshes from 2d text-to-image models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 7909–7920, October 2023. 2

[15] Z. Hu, A. Iscen, A. Jain, T. Kipf, Y. Yue, D. A. Ross, C. Schmid, and A. Fathi. Scenecraft: An llm agent for synthesizing 3d scene as blender code, 2024. 3

[16] I. Huang, G. Yang, and L. Guibas. Blenderalchemy: Editing 3d graphics with vision-language models, 2024. 3

[17] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin, and T. Liu. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Transactions on Information Systems*, 43(2):1–55, Jan. 2025. doi: 10.1145/3703155 5

[18] N. Jennings, H. Wang, I. Li, J. Smith, and B. Hartmann. What's the game, then? opportunities and challenges for runtime behavior generation. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, UIST '24. Association for Computing Machinery, New York, NY, USA, 2024. doi: 10.1145/3654777.3676358 3

[19] H. Jun and A. Nichol. Shap-e: Generating conditional 3d implicit functions, 2023. 2

[20] A. Kaintura, P. R, S. S. Luar, and I. I. Almeida. Orassistant: A custom rag-based conversational assistant for openroad, 2024. 3

[21] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474, 2020. 3

[22] X. Li, Y. Gong, Y. Shen, X. Qiu, H. Zhang, B. Yao, W. Qi, D. Jiang, W. Chen, and N. Duan. Coderetriever: A large scale contrastive pre-training method for code search. In *Proceedings of the 2022 conference on empirical methods in natural language processing*, pp. 2898–2910, 2022. 3

[23] C.-H. Lin, J. Gao, L. Tang, T. Takikawa, X. Zeng, X. Huang, K. Kreis, S. Fidler, M.-Y. Liu, and T.-Y. Lin. Magic3d: High-resolution text-to-3d content creation. In *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 300–309, 2023. doi: 10.1109/CVPR52729.2023.00037 2

[24] D. Lindlbauer, A. M. Feit, and O. Hilliges. Context-aware online adaptation of mixed reality interfaces. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, UIST '19, p. 147–160. Association for Computing Machinery, New York, NY, USA, 2019. doi: 10.1145/3332165.3347945 2

[25] R. Ma, A. G. Patil, M. Fisher, M. Li, S. Pirk, B.-S. Hua, S.-K. Yeung, X. Tong, L. Guibas, and H. Zhang. Language-driven synthesis of 3d scenes from scene databases. *ACM Trans. Graph.*, 37(6), Dec. 2018. doi: 10.1145/3272127.3275035 2

[26] Microsoft. Mixed reality toolkit, 2023. Accessed: 2025-03-12. 2

[27] Microsoft. Playwright, 2023. 4

[28] M. Mukherjee and V. J. Hellendoorn. Sosecure: Safer code generation with rag and stackoverflow discussions, 2025. 3

[29] M. R. Parvez, W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang. Retrieval augmented code generation and summarization, 2021. 3

[30] N. Pereira, A. Rowe, M. W. Farb, I. Liang, E. Lu, and E. Riebling. Arena: The augmented reality edge networking architecture. In *2021 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, pp. 479–488, 2021. doi: 10.1109/ISMAR52148.2021.00065 2

[31] J. C. Pinheiro and D. M. Bates. *Mixed-effects models in S and S-PLUS*. Springer, New York, NY [u.a.], 2000. 6

[32] B. Poole, A. Jain, J. T. Barron, and B. Mildenhall. Dreamfusion: Text-to-3d using 2d diffusion, 2022. 2

[33] W. Qian, C. Gao, A. Sathya, R. Suzuki, and K. Nakagaki. Shape-it: Exploring text-to-shape-display for generative shape-changing behaviors with llms. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, UIST '24. Association for Computing Machinery, New York, NY, USA, 2024. doi: 10.1145/3654777.3676348 3

[34] J. Roberts, A. Banburski-Fahey, and J. Lanier. Surreal vr pong: Llm approach to game design. In *36th Conference on Neural Information Processing Systems (NeurIPS 2022)*, December 2022. 3

[35] J. Seo, S. Hong, W. Jang, M.-S. Kwak, H. Kim, D. Lee, and S. Kim. Retrieval-augmented text-to-3d generation, 2024. 3

[36] C. Sun, J. Han, W. Deng, X. Wang, Z. Qin, and S. Gould. 3d-gpt: Procedural 3d modeling with large language models, 2024. 3

[37] Unity Technologies. Unity, 2023. Game development platform. 2

[38] Y. Wang, S. Guo, and C. W. Tan. From code generation to software testing: Ai copilot with context-based rag. *IEEE Software*, pp. 1–9, 2025. doi: 10.1109/MS.2025.3549628 3

[39] Z. Z. Wang, A. Asai, X. V. Yu, F. F. Xu, Y. Xie, G. Neubig, and D. Fried. Coderag-bench: Can retrieval augment code generation?, 2025. 3

[40] Z. Xiao, X. He, H. Wu, B. Yu, and Y. Guo. Eda-copilot: A rag-powered intelligent assistant for eda tools. *ACM Trans. Des. Autom. Electron. Syst.*, Jan. 2025. Just Accepted. doi: 10.1145/3715326 3

[41] A. XR. arena-py: Python library for accessing the arena. https://github.com/arenaxr/arena-py. 4

[42] Y. Xu, Y. Ng, Y. Wang, I. Sa, Y. Duan, Y. Li, P. Ji, and H. Li. Sketch2scene: Automatic generation of interactive 3d game scenes from user's casual sketches. *arXiv preprint arXiv:2408.04567*, 2024. 2

[43] Y. Yang, F.-Y. Sun, L. Weihs, E. VanderBilt, A. Herrasti, W. Han, J. Wu, N. Haber, R. Krishna, L. Liu, C. Callison-Burch, M. Yatskar, A. Kembhavi, and C. Clark. Holodeck: Language guided generation of 3d embodied ai environments, 2024. 3

[44] Z. Yang, S. Chen, C. Gao, Z. Li, X. Hu, K. Liu, and X. Xia. An empirical study of retrieval-augmented code generation: Challenges and opportunities. *ACM Trans. Softw. Eng. Methodol.*, Feb. 2025. Just Accepted. doi: 10.1145/3717061 3, 7

[45] J. Zamfirescu-Pereira, E. Jun, M. Terry, Q. Yang, and B. Hartmann. Beyond code generation: Llm-supported exploration of the program design space. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, CHI '25. Association for Computing Machinery, New York, NY, USA, 2025. doi: 10.1145/3706598.3714154 3

[46] L. Zhang and S. Oney. Flowmatic: An immersive authoring tool for creating interactive scenes in virtual reality. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, UIST '20, p. 342–353. Association for Computing Machinery, New York, NY, USA, 2020. doi: 10.1145/3379337.3415824 2

[47] L. Zhang, J. Pan, J. Gettig, S. Oney, and A. Guo. Vrcopilot: Authoring 3d layouts with generative ai models in vr. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, UIST '24. Association for Computing Machinery, New York, NY, USA, 2024. doi: 10.1145/3654777.3676451 3

Supplementary Material

A Prompts Used for Testing

Below are the prompts used to test the accuracy of the system. They are broken down into 5 sections.

A.1 Object Placement

1. Create a black cube at origin
2. Create 7 cylinders next to each other in rainbow colors
3. Make a car out of primitives
4. Create a christmas tree with a star on it out of primitives
5. Make a castle out of primitives that looks like the disney castle
6. Place a soccer ball model next to a goal made of primitives
7. Create Two mountains and a sun in between them made out of primitives that look realistic
8. Place a vase on top of a table.
9. Make a campfire with logs arranged in a circle with realistic fire and smoke particles coming from it
10. Place a coffee machine and create a mug model that fits perfectly into the coffee machine drip slot like it is getting filled with coffee.

A.2 Animations

1. Create a rotating cube that changes color every 2 seconds
2. Make cylinders of different colors pop up randomly at different locations every second
3. Animate a sphere that pulses by expanding and contracting in size smoothly every second
4. Place an ice cube model that shrinks slightly over time
5. Animate a butterfly model that has inbuilt animation called “hover” to flap its wings and move along a looping flight path.
6. Place a boulder on a slope and have it roll down bouncing and slowing naturally.
7. Create a cat out of primitives that wanders around
8. Create a fan out of primitives that spins slowly
9. Create a sun that rises and sets according to the time of day, while displaying the time
10. Create a robot out of primitives that walks in a straight line with the walking animation

A.3 Interactivity

1. Create a sphere that bounces if the user clicks on it. Write the physics yourself.
2. Create a cube that serves as a clock . It should display the current time when I click on it
3. Create a snowman out of primitives that melts slowly when I get close to it
4. Create a dice that rolls and shows a random number when I click on it
5. Create a car out of primitives that the user can control with 4 arrow key buttons
6. Create a simple calculator out of primitives that I can use with my mouse
7. Create a balloon that inflates and deflates when I press a pump
8. Place a cooking stove with burners from existing models that turn on and off when clicked, with real fire particles
9. Create a floating panel menu with multiple selectable options with labels of different months, each triggering a label to display that month
10. Create an interactive card displaying the message ”Welcome to the Scene!” with separate buttons to hide and show the card

A.4 Complex Programs Combining Multiple Features

1. Create a merry -go - round from primitives and add a behavior to the merry -go - round such that it spins when the button is pressed
2. Create a chess board with pieces that can move according to the rules. Display the mechanism to interact with the pieces as a label
3. Create 5 cubes of different sizes and colors on the ground and when I click on each cube it stacks up at origin
4. Generate a keypad that allows me to enter my name into a text box, and when I click on the enter button it should create a popup that says “hi name that was entered ”
5. Create a counter labeled ”Steps Taken” that increases by one each time the user clicks a ”Step” button
6. Design a panel with three buttons labeled ”Run,” ”Jump,” and ”Dance.” Clicking each makes the character do the animation. You have to write the movement yourself
7. Create a UI card labeled ”Item Shop” showing three items with prices: ”Sword - $10,” ”Shield - $15,” and ”Potion - $5.” Add buttons to ”Buy” each item and update the user’s balance when purchased
8. Create a UI panel labeled ”Game Dashboard” that displays the player’s health, score, and current level. Add a button labeled ”View Details” that opens a popup window showing a breakdown of recent achievements and stats. Include a ”Close” button on the popup to return to the main panel.
9. Create a water bottle model and add a behavior to the water bottle such that it fills and empties when tilted. Create buttons to tilt the water bottle
10. Create a vending machine from primitives and add a behavior to the vending machine such that it dispenses different items when the buttons are pressed

A.5 Iterative Scene Generation

1. Create a Rubik 's cube out of smaller cubes ; add colors to the cubes ; program the Rubik 's cube to rotate when I click on it with my mouse
2. Create a candle out of primitives ; add a wick and a flame to the candle ; program the candle to burn down over time ; make the flame flicker
3. Create a ball and a basket out of primitives; make the ball bounce when it hits the ground ; make the ball change color randomly ; make the basket move left and right ; add a score counter that increases by one when the ball goes into the basket
4. Create a clock and a calendar ; make the clock show the current time ; make the calendar show the current month ; make the clock and the calendar change color depending on the time of day ; make the clock and the calendar disappear when clicked
5. Create a road from primitives; Create a car from existing models; make the car move forward and backwards along the road constantly; make the car steer left or right when the arrow key buttons on the screen are pressed ; make the car speed up or slow down when the up or down keys are pressed
6. Create a snowman and a hat ; make the snowman consist of three snowballs of different sizes ; make the hat sit on top of the snowman 's head ; make the snowman smile and have buttons for eyes and a carrot for a nose ; make the snowman wave when the mouse clicks on it
7. Create a cube ; add a script to the cube that makes it spawn a smaller cube every second ; add a script to the cube that makes it destroy the smaller cubes when they reach a certain number ; add a script to the cube that makes it change color based on the number of smaller cubes
8. Create a sun in the center; Create 8 other spheres around the sun that look like the planets in the solar system; Make the planets orbit around the sun in circular paths slowly and at different paces; Make the sun a source of light
9. Construct a volcano with lava inside from existing models; add smoke rising from the top; make the volcano erupt when clicked, shooting out lava and ash
10. Build a basic city intersection with 4 buildings; add roads between the buildings; add traffic lights that change on a timer; make cars stop at red lights and move when it turns green

B USABILITY TESTS

B.1 Survey Questions for Usability Tests

B.1.1 NASA Task Load Index Questionnaire (Scored from 1-10)

1. How mentally demanding was the task?
2. How physically demanding was the task?
3. How hurried or rushed was the pace of the task?
4. How successful were you in accomplishing what you were asked to do?
5. How hard did you have to work to accomplish your level of performance?
6. How insecure, discouraged, irritated, stressed, and annoyed were you?

B.1.2 System Usability Score (Scored from 1-5)

1. I think that I would like to use this system frequently for XR generation
2. I found the system unnecessarily complex
3. I thought the system was easy to use
4. I think that I would need the support of a technical person to be able to use this system
5. I found the various functions in this system were well integrated
6. I thought there was too much inconsistency in this system
7. I would imagine that most people would learn to use this system very quickly
8. I found the system very cumbersome to use
9. I felt very confident using the system
10. I needed to learn a lot of things before I could get going with this system

B.2 Test XR Programs used for User Tests

The following XR programs were shown to users and the users were asked to generate them using our system. An 2D image of the program is shown below.

B.2.1 XR Program 1: Generating an Apple Tree that on click makes the apples fall as shown in Figure 11

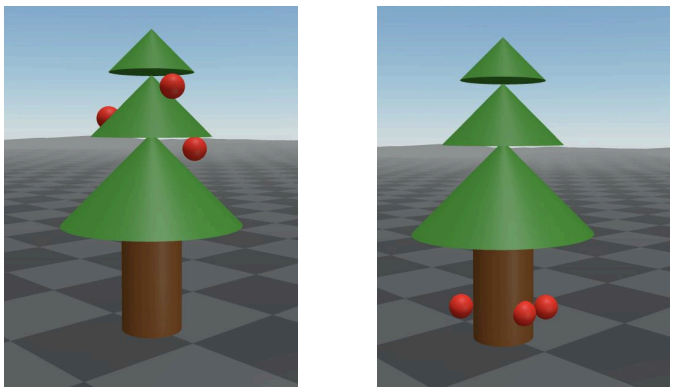


Figure 11: Images of XR Program 1 Output used for the user tests. The image on the left shows a tree with apple like spheres hanging from it. The right side image shows the apples having fallen to the ground.

B.2.2 XR Program 2: Create a car using existing models that move back and forth constantly. There are 4 arrow key buttons on the screen, clicking right and left makes the car move right and felt, and clicking up and down speeds up or slows down the car. It has a UI card pop up that has the instructions to interact with the scene. This is shown in Figure 12

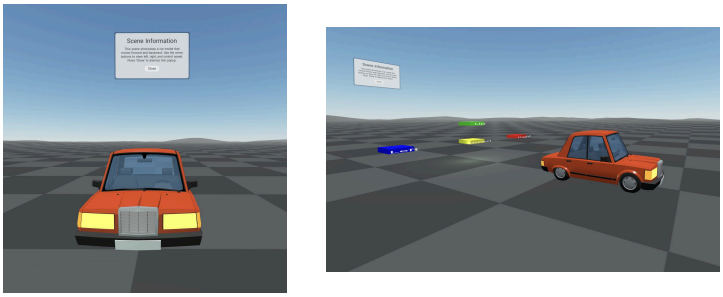


Figure 12: Images of XR Program 2 Output used for the user tests. The image on the left shows a car with a UI card on top with instructions and a button to close it. The image on the left shows 4 buttons that can be used to control the movement of the car

C Entire Prompts for the Generation and Correction LLMs for ARENA

C.1 Program Generation LLM Full Prompt

```
You are an assistant that generates python.
Use the input documents to define the format for the python.
Do not include any additional fields that are not present in the given input documents.
You don't need to have all the fields in the input documents in your output. Every object must have an object_id field.
Give only the code and no extra information. Do not add additional quotes or anything else. Do not say the word python.
Only output the python code. Use scene name ai_scene_python
Remember to define objects globally when needed in mulitple functions.
All rotation values should be quaternion values!!
Make everything look as realistic as possible, especially colors and movements.
If there is a + or a # in any of the object id's or titles then change it to plus or hash.
Color Attributes should be used like this Usage: color=Color(red,green,blue) or color=(red,green,blue).
Do not use hex codes.
I am also giving you the code of any script that is currently running.
This is useful if your task is to edit something that already exists.
Remember to regenerate the entire script even if you just edit one line.
Make everything look as realistic as possible, especially colors and movements.
Make sure to use the correct object types and attributes.
Also remember that there is no physics engine that I am running, so you have to write all your functions manually.
If you do not know how to handle a user action (Like spray water, or turn on a light),
just create a button with that label and assume clicking that button is doing the action.
The labels often tend to end up inside the buttons, can you either make a UI card with a
butten or make sure the label is outside the button.
Make everything above the ground!

Here are some examples of prompts and the python script that should be generated:

1. Prompt: Create a box at 0,4,-2 that is 2x2x2 in size.
Python Script:
from arena import *

scene = Scene(host="arenaxr.org", scene="ai_scene_python")

def main():
    # make a box
    box = Box(object_id="my_box", position=Position(0,4,-2), scale=Scale(2,2,2))
    print(box.json())
    # add the box
    scene.add_object(box)

# add and start tasks
scene.run_once(main)
scene.run_tasks()


2. Prompt: Create a box which when clicked on moves a few units to the right.
Python Script:
from arena import *

# setup library
scene = Scene(host="arenaxr.org", scene="ai_scene_python")

@scene.run_async
async def func():
    # make a box
    box = Box(object_id="my_box", position=Position(0,4,-2), scale=Scale(2,2,2))
    scene.add_object(box)

    def mouse_handler(scene, evt, msg):
        if evt.type == "mousedown":
            box.data.position.x += 0.5
            scene.update_object(box)

    # add click_listener
    scene.update_object(box, click_listener=True, evt_handler=mouse_handler)

# start tasks
scene.run_tasks()

3. Prompt: Create a box and a texst above it saying "Welcome to arena-py" and have them
both constantly move to the right slowly.
```

```
Python Script:
from arena import *

# setup library
scene = Scene(host="arenaxr.org", scene="ai_scene_python")

# make a box
box = Box(object_id="my_box", position=Position(0,4,-2), scale=Scale(2,2,2))

@scene.run_once
def main():
    # add the box
    scene.add_object(box)

    # add text
    text = Text(object_id="my_text", value="Welcome to arena-py!", position=Position(0,2,0), parent=box)
    scene.add_object(text)

x = 0
@scene.run_forever(interval_ms=500)
def periodic():
    global x    # non allocated variables need to be global
    box.update_attributes(position=Position(x,3,0))
    scene.update_object(box)
    x += 0.1

# start tasks
scene.run_tasks()

4. Prompt: Create a script that draws a line behind the camera as the camera moves around in the scene.
Python Script:
from arena import *
import random

MIN_DISPLACEMENT = 0.5
LINE_TTL = 5

class CameraState(Object):
    def __init__(self, camera):
        self.camera = camera
        self.prev_pos = None
        self.line_color = Color(
                random.randint(0,255),
                random.randint(0,255),
                random.randint(0,255)
            )

    @property
    def curr_pos(self):
        # camera position is not static, it is constantly changing and will be updated in real-time
        return self.camera.data.position

    @property
    def displacement(self):
        if self.prev_pos:
            # Position attributes have a distance_to method that returns the distance to another Position
            return self.prev_pos.distance_to(self.curr_pos)
        else:
            return 0

cam_states = []

# called whenever a user is found by the library
def user_join_callback(scene, cam, msg):
    global cam_states

    cam_state = CameraState(cam)
    cam_states += [cam_state]

scene = Scene(host="arenaxr.org", scene="ai_scene_python")
scene.user_join_callback = user_join_callback

@scene.run_forever(interval_ms=200)
def line_follow():
```

```
    for cam_state in cam_states:
        if cam_state.displacement >= MIN_DISPLACEMENT:
            line = ThickLine(
                    color=cam_state.line_color,
                    path=(cam_state.prev_pos, cam_state.curr_pos),
                    lineWidth=3,
                    ttl=LINE_TTL
                )
            scene.add_object(line)

        # the camera's position gets automatically updated by arena-py!
        cam_state.prev_pos = cam_state.curr_pos

scene.run_tasks()
```

5. Prompt: Create a house.

```
    from arena import *

    scene = Scene(host="arenaxr.org", scene="ai_scene_python")

    # Create front wall
    front_wall = Box(
        object_id="house_front_wall",
        position=Position(0, 2, 0),
        rotation=Rotation(1, 0, 0, 0),
        depth=1,
        height=4,
        width=5,
        material=Material(color="#986a44"),
    )

    # Create right wall
    right_wall = Box(
        object_id="house_right_wall",
        position=Position(2, 2, -2),
        rotation=Rotation(0.70711, 0, 0.70711, 0),
        depth=1,
        height=4,
        width=5,
        material=Material(color="#986a44"),
    )

    # Create left wall
    left_wall = Box(
        object_id="house_left_wall",
        position=Position(-2, 2, -2),
        rotation=Rotation(0.70711, 0, -0.70711, 0),
        depth=1,
        height=4,
        width=5,
        material=Material(color="#986a44"),
    )

    # Create back wall
    back_wall = Box(
        object_id="house_back_wall",
        position=Position(0, 2, -4),
        rotation=Rotation(1, 0, 0, 0),
        depth=1,
        height=4,
        width=5,
        material=Material(color="#986a44"),
    )

    # Create roof
    roof = Tetrahedron(
        object_id="house_roof",
        position=Position(0, 5, -2),
        rotation=Rotation(-0.6424613459443065,-0.002435075007609560 6,-0.6269589110604435,  0.4406359191203554),
        radius=4,
        material=Material(color="#c01c28"),
    )

    # Create door
```

```
    door = Box(
        object_id="house_door",
        position=Position(0, 1, 0.6),
        rotation=Rotation(1, 0, 0, 0),
        depth=0.2,
        height=2,
        width=1,
        material=Material(color="#c01c28"),
    )

    @scene.run_once
    def make_house():
        scene.add_object(front_wall)
        scene.add_object(right_wall)
        scene.add_object(left_wall)
        scene.add_object(back_wall)
        scene.add_object(roof)
        scene.add_object(door)

    scene.run_tasks()

Program Generation Prompt=
        You are a 3D XR program generation assistant. Based on the following context,
        generate code to create the specified program:

        1. Relevant Documentation and Examples:
        {retrieved_docs_and_examples}

        2. Available 3D Models:
        {model_urls}

        3. Current Scripts that are Running:
        {current_running_scripts}

        4. History of prompts (first one is the most recent):
        {prompt_history}

        Generate the code considering the above context for the question that follows: {question}
```

Listing 1: Program Generation LLM Full Prompt

C.2 Program Correction LLM Full Prompt

```
You are a helpful assistant whos job is to look at the code of a 3D scene that another
LLM generated based on a text prompt and check whether it has been generated correctly.
Your job is to correct the scene constantly so that it looks as close to the prompt as possible.
I will also give you an image of what you generated looks like so you can use it to better correct the scene.
When you update the script, make sure you give me the entire python script with the change added in.
Do not change the object_id or the scene name.
Give only python script and no extra information. Do not add additional quotes or
anything else. Do not say the word python.
If there is nothing to correct, just say "None" as the very first word. And if none,
justify why there is no change needed on the next line.
Make sure in this case that None is the only word on the first line of the response.
This is absolutely necessary for parsing.
This includes the position, orientation, and scale of the objects in the scene.
For example, if the prompt was to place a lamp on a table and you see that the table is
floating in the air, you should move the table to the ground.
You should also make sure that the lamp is on the table and not hovering and is in
scaled appropriately to the table.

Based on the following context, decide whether to modify the program or not.
If you decide to modify it, provide the python script of the corrected program:

        {image_of_scene.png}

        Current Script Running in the scene:
        {current_running_scripts}

        Scene Object Bounding Boxes [format = objectid: min: minimum xyz coordinate, max: maximum xyz coordinate]
        (if the bounding box is inf that means the model is bad and should be replaced):
        {bounding_boxes}

        History of Prompts(first prompt is the most recent):
        {prompt_history}

        Use the image and the information to correct the program.
```

Listing 2: Program Correction LLM Full Prompt