Procedural Scene Programs for Open-Universe Scene Generation: LLM-Free Error Correction via Program Search

MAXIM GUMIN, Brown University, USA
DO HEON HAN, Brown University, USA
SEUNG JEAN YOO, Brown University, USA
ADITYA GANESHAN, Brown University, USA
R. KENNY JONES, Brown University, USA
KAILIANG FU, Brown University, USA
RIO AGUINA-KANG, UC San Diego, USA
STEWART MORRIS, Brown University, USA
DANIEL RITCHIE, Brown University, USA

Fig. 1. Our method generates 3D indoor and outdoor scene layouts from open-ended text prompts. Generated layouts are not limited to a fixed set of room types or object categories. All layouts in the figure are generated as procedural programs, where the LLM's mistakes are corrected using our program search mechanism.

Synthesizing 3D scenes from open-vocabulary text descriptions is a challenging, important, and recently-popular application. One of its critical subproblems is *layout generation*: given a set of objects, lay them out to produce a scene matching the input description. Nearly all recent work adopts a *declarative* paradigm for this problem: using an LLM to generate a specification of constraints between objects, then solving those constraints to produce the final layout. In contrast, we explore an alternative *imperative* paradigm, in which an LLM iteratively places objects, with each object's position and orientation computed as a function of previously-placed objects. The imperative approach allows for a simpler scene specification language while also handling a wider variety and larger complexity of scenes. We further improve the robustness of our imperative scheme by developing an error correction mechanism that iteratively improves the scene's validity while staying as close as possible to the original layout generated by the LLM. In forced-choice perceptual studies, participants preferred layouts generated by our imperative approach 82% and 94% of the time when compared against two declarative layout generation methods. We also present a simple, automated evaluation metric for 3D scene layout generation that aligns well with human preferences.

CCS Concepts: • **Computing methodologies** → **Computer graphics**; **Neural networks**; **Natural language generation**.

Additional Key Words and Phrases: scene synthesis, program synthesis, layout generation, large language models

ACM Reference Format:
Maxim Gumin, Do Heon Han, Seung Jean Yoo, Aditya Ganeshan, R. Kenny Jones, Kailiang Fu, Rio Aguina-Kang, Stewart Morris, and Daniel Ritchie. 2025. Procedural Scene Programs for Open-Universe Scene Generation: LLM-Free Error Correction via Program Search. In *SIGGRAPH Asia 2025 Conference Papers (SA Conference Papers '25), December 15–18, 2025, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3757377.3763930

SA Conference Papers '25, December 15–18, 2025, Hong Kong, Hong Kong

ACM ISBN 979-8-4007-2137-3/2025/12
https://doi.org/10.1145/3757377.3763930

1 INTRODUCTION

3D scenes serve as representations of the environments surrounding us: homes, workplaces, social gathering spaces, etc. They can also represent virtual worlds for games, films, and architecture. In this paper, we study the problem of *open-universe scene generation*: synthesizing a 3D scene from a natural language prompt, where prompts are not limited to a fixed vocabulary, and objects are not restricted to a fixed set of object categories.

Large language models (LLMs) are a natural fit for this task given their vast knowledge bases. A 3D scene can be viewed as a collection of objects, where each object is specified by attributes such as size, mesh, position, and orientation. Synthesizing such a scene involves several steps: generating a set of objects, determining the positions and orientations of those objects (i.e. layout), and generating or retrieving 3D meshes for each object. In this paper, we focus on the layout subproblem.

The first works that studied LLM-based scene generation took an *imperative* approach, directly specifying the desired layout state. LayoutGPT [Feng et al. 2023] is a representative example, where an LLM designs a layout by iteratively predicting object parameters (sizes, positions, orientation). This paradigm offered a number of advantages, as LLMs showed an impressive ability to know what objects should populate a scene across a wide range of scenarios. While this allowed LLMs to quickly produce 3D scene layouts from text, LLMs struggled with accurate placement of objects and would produce layouts that violated physical properties (overlapping objects, floating objects) [Makatura et al. 2023].

To overcome these limitations, recent work has overwhelmingly adopted *declarative* approaches [Aguina-Kang et al. 2024; Fu et al. 2024; Hu et al. 2024; Kodnongbua et al. 2024; Littlefair et al. 2025; Yang et al. 2023; Çelen et al. 2024]. Unlike the imperative paradigm, these declarative specifications do not directly specify object attributes, but rather describe properties that the layout should have. These operations are often designed to associate with object-to-object relations such as `on(a,b)`, `adjacent(a,b)` or `aligned(a,b,c)`. An execution module is then used to convert these programs into a layout, e.g. a solver can search for a layout that best minimizes the error with respect to the LLM-predicted relations. A nice property of this paradigm is that instead of predicting object parameterizations directly and independently, the LLM's job is simplified, as its prediction can use higher-level operations.

While declarative approaches show advantages over simple imperative schemes like LayoutGPT [Aguina-Kang et al. 2024; Yang et al. 2023], this framing is not without its own limitations. Declarative approaches can be slow to execute, as they have to solve a complex optimization problem just to produce a single layout. This is especially true for scenes containing a large number of objects (e.g. museums, theaters), because the time required for a solver to find a satisfying configuration of object positions depends at least linearly on the number of relations, and the number of relations usually depends quadratically on the number of objects. Such slowness makes it more difficult to edit or modify a scene. In addition, not every object configuration can be expressed in a particular declarative domain-specific language (DSL). For example, it can be hard to arrange objects in a spiral if the declarative DSL has no explicit `spiral()` constraint.

In searching for ways to improve LLM-based scene generation, we can take inspiration from how declarative approaches made improvements upon early imperative methods. So, what lead to the dominance of the declarative paradigm? For one, declarative programming languages used higher-level operations that had more natural semantic analogs, which simplified the LLM prediction task. Perhaps more importantly, declarative paradigms have a built-in error-correction module—their solver. That is, all existing LLM models make mistakes when producing scene layout specifications, but solving for an error-minimizing layout can mitigate this issue.

Are the above differentiators unique to the declarative setting, or could they be realized in an imperative approach as well? The imperative programming paradigm has a number of benefits over the declarative formulation. Execution is fast, direct and exact (specifying a single scene). Declarative programs can only enforce relationships encoded as constraints in their domain-specific language, while imperative programs in principle have the capacity to realize any possible object configuration. How can we take the lessons from the declarative paradigm and use them to improve the imperative approach?

We propose a co-designed system for LLM scene synthesis that combines procedural scene programs with a symbolic error-correction module. To this end, we introduce a new relation-centric design pattern for procedural scene programs, which we implement in a Python-embedded DSL we call Procedural Scene Description Language (PSDL). Beyond simply specifying each object position as an independent decision, PSDL allows object attributes to be defined using *parametric relationships* with respect to the attributes of previously defined objects or constants. PSDL also includes a domain-specific operator for specifying *local coordinate frames*, simplifying the task for the LLM, as it can create sub-layouts in a canonical space that are then cohesively transformed into the global scene. Since PSDL programs are embedded in Python, they allow the LLM to make use of *control flow* operations such as loops and conditionals, supporting the definition of complex layout patterns.

In tandem with the design of PSDL, we co-design an error-correction scheme that makes use of these improved procedural programs. Our error correction module iteratively refines LLM-generated programs while preserving their original structure as much as possible. The goal of this module is to search for programs that minimize errors in the scene produced by executing the program while also minimizing deviation from the initial program definition. As our procedural scene representation avoids global solvers, program evaluation is cheap, so our search procedure is able to execute modified programs, observe a new layout, and use this information to decide how to adjust the symbolic scene representation. We find that a simple local search method performs well, and is particularly effective for the procedural scenes that use our new design patterns. For instance, due to the presence of shared variables, the error correction module can easily coordinate updates across multiple objects, such as adjusting the layout of an entire row of objects with a single update.

We evaluate our approach against prior imperative and declarative scene generation schemes. We propose a taxonomy of scene prompts along a number of different axes: small indoor vs large

outdoor, common spaces to fantastical scenes, chaotic to highly structured. While we find that our new formulation is competitive in all of these settings, it especially excels for prompts describing large, highly-structured scenes. As these perceptual studies don't scale well, we also introduce a new vision-language model (VLM)-based evaluation scheme that we find is better aligned with human judgments than alternative automatic evaluation approaches that use VLMs.

In summary, our contributions are:

(1) The PSDL language for imperative specification of open-universe scene layouts.
(2) An error correction method for PSDL programs that does not involve additional calls to an LLM.
(3) A protocol for evaluating open-universe scene layout synthesis systems, including a benchmark set of input descriptions covering a wide variety of possible scenes.
(4) A human-aligned automated evaluation method for scene layout generation.

2 RELATED WORK

Scene Synthesis pre-LLMs. The problem of scene synthesis has a rich history in computer graphics. Early work focused on laying out objects based on manually-defined design principles [Merrell et al. 2011], simple statistical relationships between objects extracted from a small set of examples [Yu et al. 2011], or with programmatically-specified constraints [Yeh et al. 2012]. Later research focused on data-driven methods [Fisher et al. 2012; Kermani et al. 2016; Liang et al. 2017; Qi et al. 2018], with a surge in activity as deep neural networks gained popularity [Li et al. 2018; Lin and Mu 2024; Paschalidou et al. 2021; Ritchie et al. 2019; Tang et al. 2023; Wang et al. 2019, 2018, 2020; Zhang et al. 2018; Zhou et al. 2019]. These prior works develop closed-universe generative models (i.e. restricted to certain scene and object categories), and all of them require (in some cases quite large) datasets of 3D scenes for training. By contrast, LLMs offer the capability—in theory—to synthesize arbitrary types of scenes and to do so with no explicit 3D scene training data.

Scene Synthesis with LLMs. While research on text-based scene generation predates the rise of LLMs [Chang et al. 2017; Coyne and Sproat 2001], their development has led to a new generation of text-to-scene generative models which are both more flexible and more open-ended than earlier systems. LayoutGPT [Feng et al. 2023] was a pioneering system that operated in a very simple imperative paradigm by iteratively specifying object coordinates, dimensions, and orientations. This 'coordinate-centric' framing has seen adoption in other scene generation framings as well: [Bhat et al. 2025; Öcal et al. 2024]. The Scene Language [Zhang et al. 2024] is a concurrent work that also uses an imperative approach for producing scene layouts of objects, which can then be converted into detailed scene through a neural rendering scheme. This system produce scene programs that use control flow, but doesn't use parametric relationships, local coordinate frames, or any error-correction mechanism. Many other LLM scene generation works have adopted a declarative approach, i.e. using an LLM to produce a declarative program specifying the layout constraints [Aguina-Kang et al. 2024; Fu et al. 2024; Hu et al. 2024; Kodnongbua et al. 2024; Littlefair et al. 2025; Yang et al. 2023; Çelen et al. 2024]. We experimentally compare the benefits of our procedural formulation with an integrated error-correction module over these alternatives.

Automatically Correcting LLM Outputs. As LLMs can fail to produce correct output in one shot, many prior works deploy corrective mechanisms to refine LLM-generated outputs, including some work on LLM-based scene generation [Hu et al. 2024]. The most common, and general, approach is self-correction, where the output of an LLM is iteratively refined by the LLM itself [Pan et al. 2023]. Such self-correction mechanisms, while appealing in theory, are costly to run (requiring multiple LLM calls), and on code generation tasks, they typically offer modest or no real performance gain [Olausson et al. 2024]. Instead, we propose an efficient error correction scheme based on iterative local search, finding programs that are close to the LLM's original output while minimizing errors such as object overlaps. We compare our proposed error-correction module against LLM self repair, a gradient descent baseline, and alternative formulations, demonstrating its benefits experimentally.

3 OVERVIEW

Our goal is to investigate the quality of layouts generated by our method, in comparison to prior approaches. As much as possible, we would like this comparison to show the relative merits of each system for layout generation, rather than details of how a particular scene synthesis system was implemented. Thus, we design a scene synthesis pipeline which factors out computational stages not relevant to layout generation and shares those stages in common between the different layout generation methods that we consider.

In the first stage, an LLM takes a textual scene description as input and outputs a "scene template" consisting of the dimensions of the scene and a list of objects. Each object is defined by a name, a set of dimensions, and one of three types of physical support: `STANDING`, `WALL-MOUNTED`, or `FLOATING`. This template is then passed to the layout generation stage, which determines each object's position and orientation.

In the first sub-stage of layout generation, an LLM writes a scene program in our Python-embedded DSL (PSDL). The LLM is prompted with a system prompt that says, in effect: given a scene name and the list of objects, produce a high-quality layout described in PSDL. The prompt explains how to use the DSL and emphasizes the core principle: place new objects relative to already placed objects or to the scene cuboid. It also encourages variable sharing to minimize repeated numeric constants (e.g., prefer a single parameter reused across placements rather than duplicating the same literal). Four in-context examples (provided in the supplement) accompany this system prompt to illustrate the intended use of the DSL.

In the second sub-stage of layout generation, an LLM-free error-correction module rewrites parts of the generated scene program to remove layout violations while preserving the program's structure. The layout generation stage differs across the systems we evaluate, but all share the same upstream template and downstream visualization components.

Finally (and optionally), the pipeline can invoke an object retrieval stage to retrieve a 3D mesh for each object in the layout. Object retrieval is not the focus of our work, but we include it in

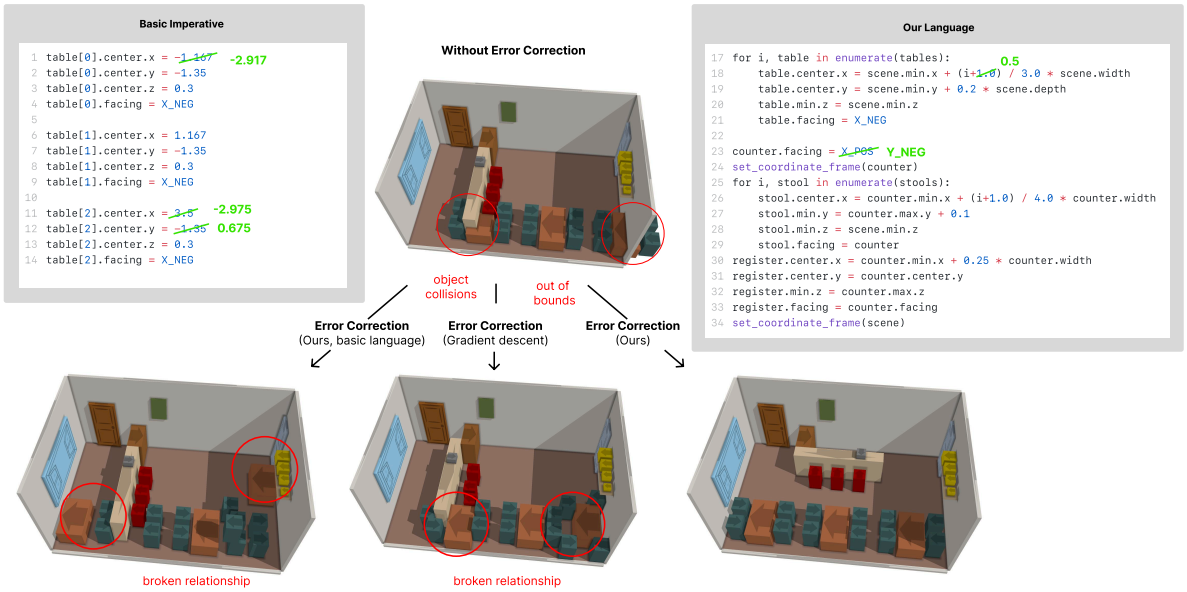

Fig. 2. Here we show how our error correction mechanism works for the same initial object configuration expressed in 2 languages: Layout-GPT style basic imperative language (left), and PSDL (right). The initial configuration has a positive loss because some objects go out-of-bounds, and some pairs of objects are overlapping. All corrected scenes have loss zero. However, the corrected version expressed in the basic imperative language, and the scene corrected with gradient descent (middle) break the relationship between tables and chairs. Our experiment shows that, given 2 corrections of the same initial object configuration, the automatic evaluator prefers the corrected version expressed in PSDL in 71.4% of cases against LayoutGPT-style, see Table 6.

our pipeline for visualizing some qualitative results. In our implementation, we use simple CLIP similarity [Radford et al. 2021] to retrieve a mesh whose rendered image matches the object's name. We retrieve meshes from the HypeHype Asset Library [HypeHype 2024], which currently contains about 6000 3D model assets. Other 3D shape datasets could also be used here [Chang et al. 2015; Deitke et al. 2023, 2022].

By comparing different layout methods while keeping the template synthesis and object retrieval stages the same, we ensure that scenes being compared have the same size and the same set of objects, making the comparison fair. To be consistent with the prior methods to which we compare, we restrict object orientations to four cardinal directions.

4 PROCEDURAL SCENE DESCRIPTION LANGUAGE

In this section, we present our new scene description language and motivate why its various features make it more amenable to error correction.

On one hand, existing declarative scene layout methods like Holodeck [Yang et al. 2023] rely on expensive global solvers, making search-based error correction loop infeasible time wise. On the other hand, existing imperative methods like LayoutGPT [Feng et al. 2023] express scenes as a flat list of object placements instructions. As a result, the search to repair such programs can only be conducted over the space of every object's position and orientation without regard for the higher-level relations that are critical for the scene. Consequently, symbolic repair of such programs often eliminates spatial violations at the cost of breaking critical scene relations.

As a solution to this predicament, we propose a new language, Procedural Scene Description Language (PSDL), which has two benefits. First, it is an imperative language which helps to avoid expensive constraint solvers. Second, PSDL provides language constructs which help express a variety of scene relations procedurally, which makes performing error correction on PSDL programs stronger. Specifically, PSDL distinguishes itself from prior imperative languages through three key features:

Explicit Object Relationships. Object positions and orientations in PSDL are not simple numerical values. Instead, they are functions defined relative to previously placed objects or scene bounds (see Table 1 left). Because these expressions are preserved during repair, object–to-object relations survive most parameter perturbations (see Fig. 2).

Expression sharing through variables. Variables allow a single symbolic constant to control many objects (see Table 1 right). Changing one parameter — say, the spacing between chairs — coherently updates all dependents, drastically shrinking the space the search must explore.

Control-flow for structured repetition. Loops and conditionals let programs capture patterns (rows of desks, symmetric place settings, etc.) concisely. A loop introduces implicit sharing: the constant stride inside the loop is automatically reused across all iterations, again tightening the search domain (see Table 1 right).

Together these features enable compact expression of scenes while benefiting from the solver-free execution of imperative programs. Representing the scene procedurally allows us to conduct the repair search at the level of parametrizations of the procedural program, rather than at the object level. Consequently, every candidate program variant preserves the scene relations by construction. Hence, our search space is compact and more semantically meaningful, which makes fixing violations less likely to disrupt essential scene structure. We evaluate the effect of different PSDL features on the quality of generated scenes in Section 6.

4.1 Language Details

Semantically, a PSDL program takes a scene template (scene dimensions and a pre-instantiated list of objects) as input and returns positions and orientations for those objects. Implementation-wise, the template supplies a short prelude that is concatenated with the PSDL code, binding object identifiers and setting the scene dimensions; the program then executes in this environment. The result is conveyed by side effects, namely updates to object attributes.

To make PSDL practical and expressive, we implement it as an embedded domain-specific language in Python. PSDL programs are allowed to be fully arbitrary Python code that additionally leverages the provided domain-specific operators; accordingly, generated programs may freely use standard library functions such as `cos`, `enumerate`, `filter`, `isinstance`, `len`, `map`, `random`, `range`, `sum` and `zip`. In the following paragraphs, we highlight the key domain-specific operators and explain their meaning.

All scene objects `o` are represented as cuboids, with dimensions `o.width`, `o.depth`, and `o.height`. Width is defined as the dimension perpendicular to the object's facing direction, while depth is defined along the facing direction. Each object exposes its geometric center `o.center`, its orientation relative to the current coordinate frame `o.facing`, which takes values among `X_NEG`, `X_POS`, `Y_NEG` or `Y_POS`, and its axis-aligned bounding box (AABB). Object's AABB is defined by two 3d vectors: `o.min` contains the minimum `x`, `y`, and `z` coordinates (in the current coordinate frame) of the box, while `o.max` contains the maximum. The special `scene` object represents the bounding cuboid of the entire scene.

By default, the program's coordinate frame is centered on the scene cuboid, but it can be re-centered on a particular object `o` using `set_coordinate_frame(o)`, which aligns the y-axis with the object's facing direction, the x-axis 90° clockwise from y, and the z-axis upward. For expressions like `chair.max.x = table.min.x - 0.1` (see Table 1) to change not only the `chair.max` vector, but the position of the `chair` itself, we overload Python's assignment operator.

An exhaustive PSDL API is provided in Supplementary Tables 8, 9.

5 LLM-FREE ERROR CORRECTION

We now describe our approach for correcting layout errors in PSDL programs. Although LLMs are capable of generating these programs, LLMs often make errors. We classify LLM errors as either *exceptions* or *layout errors*.

Exceptions include hallucinated function calls, incorrect argument usage, or "index out of range" errors that trigger exceptions at runtime. When such an error occurs, we discard the current program and request a new layout from the LLM. Layout errors arise when objects overlap, exceed scene boundaries, or violate support constraints. A common cause of layout errors is an LLM "forgetting" previously placed objects; for example, a door might be placed first, then a chair to its left, and, 20 lines of code later, a table to the chair's right, which blocks the door from opening.

Table 1. Key features of PSDL. The left column demonstrates explicit geometric relationships for positioning objects relative to others. The right column shows the use of variables to define reusable patterns, enabling concise scene descriptions. Together, these features allow PSDL to describe scenes precisely and efficiently.

Explicit Geometric Relationships	Use of Loops and Variables
`chair.max.x = table.min.x - 0.1`	`d = 2.0`
`chair.center.y = table.center.y`	`for i, c in enum(cols):`
`chair.min.z = scene.min.z`	`  c.center.x = \`
`chair.facing = table`	`scene.center.x + i * d`

In this paper, we focus on layout errors and seek to correct them without additional calls to the LLM (so-called "self-repair"), which can be time-consuming, as well as monetarily costly if LLM APIs are used. We first present our formalization of the error correction problem, then we discuss the algorithm we use to solve it.

5.1 Formulation

To quantify how problematic a layout is, we define a *loss function* $\text{loss}(L)$ that is composed of the following terms:

(1) **Out-of-Bounds Loss**: For each object, this loss is the maximum linear distance by which its bounding cuboid protrudes outside the scene boundary. If the object is fully within bounds, this term is zero.
(2) **Overlap Loss**: For each pair of objects, this loss is the cube root of the volume of the intersection of objects' bounding cuboids. To ensure usability, doors and windows are assigned expanded collision boxes to account for opening space and prevent obstruction.
(3) **Standing Loss**: For each `STANDING` object, this term measures the distance from the bottom face of the object's cuboid to the nearest horizontal surface that can support it.
(4) **Mounted Loss**: For each `MOUNTED` object, this term is the distance between the object's mountable face (usually the back or a side) and the nearest vertical surface that can support it.

These components ensure that the loss function captures both structural integrity and functional correctness in the layout. The same classes of errors are usually represented as default constraints in declarative approaches.

Our goal is to modify the program written by the LLM such that errors are removed (as much as possible), or, equivalently, the loss function is minimized. However, in fixing errors, modifications to the scene program might destroy the semantic integrity of the scene (e.g. one could fix an overlap between a dining table and its chairs by moving the table to a completely different empty region of the room). Thus, we seek to minimize the errors while maintaining semantic consistency/integrity. Evaluating whether (and to what

extent) semantic integrity is preserved is challenging; one could attempt to use an LLM to do so, but one of our explicit goals is to avoid further LLM calls. Instead, we take a conservative approach, leveraging a sufficient-but-not-necessary property: the more similar a modified program is to the original program, the more likely it is to be semantically consistent with that original program. Our goal then becomes: minimize errors while maximizing similarity between the original program and the modified one. We formalize this goal as an optimization problem:

$$\text{argmin}_P[\text{loss}(L) + d(P, P_0)]$$

where P_0 is the original program, P is the modified program, L and L_0 are layouts of P and P_0 respectively, and

$$d(P, P_0) = d_{\text{edit}}(P, P_0) + d_{\text{OT}}(L, L_0),$$

where $d_{\text{edit}}(P, P_0)$ is the edit distance between programs P and P_0, and

$$d_{\text{OT}}(L, L_0) = \min_{f \in F} \sum_{o \in O(L)} \text{Vol}(o) \cdot \|\text{center}(o) - \text{center}(f(o))\|_2$$

is the optimal transport distance [Peyré and Cuturi 2019] between layouts. Here $O(L)$ is the sets of objects in L, and F is a set of bijections between $O(L)$ and $O(L_0)$ that preserve object categories. Adding the mass transport term ensures that larger objects are penalized more for movement, encouraging the preservation of the positions of major scene elements while allowing smaller objects to be adjusted.

To define the edit distance $d_{\text{edit}}(P, P_0)$ between two PSDL programs, we need to define the set of elementary edits $E(P)$. Then, edit distance $d_{\text{edit}}(P, P_0)$ is the length of the shortest sequence of elementary edits that transforms P into P_0. In our experiments, we noticed that, when tasked with generating a layout, LLMs make most mistakes in (1) using correct numeric parameters and (2) setting correct object facing directions. Therefore, to make error correction as efficient as possible, we define the set of elementary edits $E(P)$ to consist of rewrites of constant expressions and rewrites of direction expressions.

5.2 Algorithm

Minimizing the above objective is a complex search problem in a vast space of programs. Complete enumeration of this space (popular in other program synthesis work) is infeasible. Therefore, we must instead use approximations. Since one of our objectives is to maximize similarity to original scene, we consider search algorithms which start from the original program. Then we iteratively improve this program using local search in the space of PSDL programs.

Precisely, given an LLM-generated program P_{LLM}, we iteratively refine it by constructing a sequence of programs $P_0 = P_{LLM}, P_1, P_2, \ldots$:

$$P_{i+1} = \underset{P \in N(P_i)}{\text{argmin}}\, f(L), \quad f(L) = \text{loss}(L) + d_{\text{OT}}(L, L_0),$$

where L and L_0 are layouts of P and P_0 respectively. Here $N(P)$ is the *neighborhood* of program P relative to the edits $E(P)$:

$$N(P) = \{e(P) \mid e \in E_{\text{fin}}(P)\},$$

Table 2. Preference rates for scenes generated using our procedural approach with the error correction module vs. two declarative approaches and a variant without the error correction module in a forced-choice perceptual study.

Scene Type	DeclBase	Holodeck	Ours w/o EC
Ours w/ EC vs.	82.9%	94.3%	74.3%
Ours w/o EC vs.	61.8%		-

Table 3. How frequently different automated evaluation metrics agreed with the majority human judgment from our perceptual study. Our new method (LLMCompare) is simple and outperforms prior approaches designed for evaluating text-to-image generative models.

LLMCompare	LLMCompare (no +/-)	VQAScore	DSG
77.1%	70.0%	58.6%	50.7%

Table 4. Preference rates using the automated evaluation metric. 'Ours' column includes scenes generated using our 70 prompts. 'Holodeck' column includes 52 scene types from MIT Scenes [Quattoni and Torralba 2009] used by Holodeck, and 14 longer prompts from Holodeck's qualitative examples. The 'Complex' column includes all prompts with four or more words. Our method consistently outperforms other approaches across all scene categories and performs well even with complex prompts.

		Scene Source		*Prompt Type*	
	All	**Ours**	**Holodeck**	**Simple**	**Complex**
vs. DeclBase	76.5%	77.1%	75.8%	77.8%	68.4%
vs. Holodeck	82.4%	90.0%	74.2%	82.9%	78.9%
vs. FlairGPT	88.2%	85.7%	90.9%	87.2%	94.7%

where $E_{\text{fin}}(P)$ is finite subset of edits randomly sampled from $E(P)$. Specifically, $E_{\text{fin}}(P)$ consists of 10 edits per each constant expression and 4 edits per each direction expression. For constants, we multiply a constant by a random variable $\pm 4^Y$, where the sign is chosen uniformly from $\{-1, 1\}$, and Y is sampled uniformly and independently in the interval $[-1, 1]$; for direction expressions, we include all 4 cardinal directions in $E_{\text{fin}}(P)$. The iteration terminates when no available edit decreases the objective f more than some small threshold.

Unlike the declarative approach, which adjusts each object individually, our error-correction strategy modifies expressions in the scene description program. This is especially advantageous for scenes with many objects, where a single floating-point parameter often governs multiple placements (e.g., chair spacing in a theater). By working in this lower-dimensional parameter space, the method both preserves the overall structure of the scene — ensuring aesthetic consistency — and reduces computational overhead compared to per-object adjustments. On average, only a few adjustments — 7.13 per scene — are sufficient to resolve errors. See the supplemental material for videos illustrating the error correction process.

In the next section, we evaluate this simple LLM-free iterated local search against the error correction baselines of gradient descent and LLM self-repair.

Table 5. Preference rates over Holodeck by number of objects using auto metric. '10-20' bin had 9 scenes total, '20-30' bin had 45, '30-40' bin had 37, and '40-beyond' had 42. Our method performs well in all bins, especially in scenes with 40 or more objects.

# Objs per Scene	10–20	20–30	30–40	40->
vs. Holodeck	66.7%	80.0%	72.9%	**95.2%**

Table 6. Ablation study comparing error correction modules types across preference rate, error count, and run time. The amount of errors and run time are average values per scene.

Error Correction	*Pref rate*	*# Errors*	*Run time*
No solver	60.0%	12.3	0s
Gradient descent	61.4%	0.5	2.6s
LLM self-repair	68.1%	5.7	106.7s
Local search, basic imperative	71.4%	0.8	25.2s
Local search, PSDL (ours)	-	1.1	9.3s

6 RESULTS AND EVALUATION

In this section, we evaluate different layout generation approaches on their ability to synthesize open-universe 3D scenes. We compare our method with two declarative baselines and assess the impact of error correction through forced-choice perceptual studies. We also compare various ablation conditions using a new automated evaluation method, which we show is better aligned with the perceptual study results than previous methods for automatic evaluation of text-based visual generation systems.

Implementation Details. Unless otherwise specified, we use Anthropic's `claude-3-5-sonnet-20241022` for language generation components of our proposed system. We use OpenAI's `gpt-4o` as the multimodal LLM backbone for automated evaluation methods.

Benchmark. To evaluate layout generation methods, we created a benchmark of 70 scene prompts spanning diverse environments and complexity levels. Each prompt is labeled by four attributes: *size* (small, medium, large), *location* (indoor, outdoor), *realism* (realistic, fantastical), and *structure* (chaotic, structured). The benchmark includes 14 small, 35 medium, and 21 large scenes; 48 indoor and 22 outdoor; 50 realistic and 20 fantastical; and 31 chaotic and 39 structured.

We also incorporate 52 prompts from the MIT indoor scenes dataset [Quattoni and Torralba 2009] used by Holodeck in their evaluations. To include longer prompts, we also included 14 prompts that Holodeck uses for their qualitative results. From our 70 prompts and 66 prompts from Holodeck, we extracted 19 long prompts we deemed as 'complex' to test more detailed prompts. See the supplemental for the complete list of scene prompts.

Comparison Conditions. In the subsequent experiments, we compare the following LLM-based scene layout generation methods:

- **Ours**: generating layouts as procedural programs and then improving them using our iterative error correction scheme.
- **DeclBase**: declarative layout generation approach described in [Aguina-Kang et al. 2024].
- **Holodeck**: the "Constraint-based Layout Design Module" of the Holodeck system [Yang et al. 2023].
- **FlairGPT**: declarative layout generation approach described in [Littlefair et al. 2025].

We do not compare to LayoutGPT, as it is strictly dominated by Holodeck and DeclBase, making this comparison unlikely to offer new insights.

6.1 Human Perceptual Evaluation

To compare how well different layout generation methods can produce layouts satisfying the scene prompts in our benchmark, we conducted two-alternative, forced-choice perceptual studies pitting our method against two of the declarative methods.

We recruited 10 participants from a population of university students. The participants were divided into two groups, one for each study. Each participant was shown a series of 70 comparisons, where each comparison contains a scene prompt, images of two layouts in randomized order, and a question asking them to choose which scene they thought was better (taking into account overall scene plausibility and appropriateness for the prompt).

For each comparison, we take the majority vote across all participants as the final answer. Since we seek to evaluate only the quality of object layouts, to eliminate any impact that 3D model choice might have on participant response, objects in images were rendered as colored boxes over which participants could hover their mouse cursor to reveal the object's name.

Table 2 shows the results of this experiment, and Figure 4 shows some qualitative comparisons between generated layouts. Overall, participants preferred layouts generated using our method to those generated by either of the declarative versions. Examining individual decisions rather than majority vote, the per-task histograms (Supplementary Fig. 3, top) are strongly skewed toward our method: against DeclBase most tasks fall in the 4/5–5/5 bins, and against Holodeck fully 51/70 tasks are unanimous (5/5) for ours. The per-participant totals (Supplementary Fig. 3, bottom) show the same trend: each rater preferred our method on 48–61 of 70 scenes vs. DeclBase and on 61–68 of 70 vs. Holodeck.

To evaluate the impact of error correction, we conducted two other perceptual studies using the same protocol. Our method with error correction was preferred over our method without correction 74.3% of the time, which in turn was preferred over DeclBase 61.8% of the time. These results confirm that while describing scenes as procedural programs alone is effective, error correction plays a crucial role in further improving scene quality.

6.2 Automated Evaluation

As perceptual studies are costly to run, we also investigated using automated evaluation metrics to approximate the results of a perceptual study. We experimented with two metrics designed for automated evaluation of text-to-image generative models:

- **VQAScore** [Lin et al. 2024]: Scores how well a generated image matches a text prompt using the probability a visual question answering model assigns to the output token 'yes' when asked whether the image depicts that text prompt.

- **Davidsonian Scene Graphs (DSG)** [Cho et al. 2024]: Computes a score by generating a dependency graph of simpler yes/no questions to ask about the image and then aggregating the percentage of those questions for which a VQA system returns 'yes.'

We can use these methods to compare two scene layouts by running them on rendered images of both and returning whichever has the higher score. Unfortunately, we found that neither of these methods performed much better than chance (50%) at agreeing with the majority-vote judgments from our perceptual study. As long as a scene contains the right objects, VQA models can recognize the type of scene even with poor layouts, making VQAScore insufficiently discriminative for layout evaluation. Similarly, DSG struggled to generate yes/no questions which could differentiate the two scenes, leading to ties for most judgments.

These results motivated us to develop a simple new method for automated evaluation of scene layouts. Specifically, we prompt a multimodal LLM with images of two scene layouts (along with a general task prompt) and ask it to list the pros and cons of each layout with respect to the scene prompt. At the end of its output, the LLM returns which scene is better. As shown in Table 3, this simple method is much more aligned with human judgments. Table 3 also includes results for an ablated version of our method which does not ask the LLM to first generate a pros & cons list, illustrating that this additional step does improve the method's agreement with people.

As shown in Table 4, running our automated evaluation metric on our benchmark results in our scenes being chosen 77.1% of the time over DeclBase scenes (vs. 82.9% in the perceptual study) and 90.0% of the time over Holodeck scenes (vs. 94.3% from the perceptual study). While there is some discrepancy from 'gold standard' human judgments, the trends are still clear.

Table 5 shows that our automated evaluation metric prefers our scenes over Holodeck scenes across different object count scenarios. It is notable that our scenes are preferred the most when there are 40 or more objects, showing that our method excels in complicated scenes with high object counts.

6.3 Error Correction

Table 6 reports automatic evaluation preference rates among various error correction strategies, and the average amounts of remaining errors. We can see that our local search for PSDL programs is very effective, correcting 91% of all errors. Gradient descent and local search for basic imperative programs corrects even a bigger share of errors (96% and 93.5% respectively), because these two methods are able to move individual objects. However, our method wins over all baselines in terms of preference rates: as we explain in Section 4, resolving errors in the space of individual objects often breaks intended relationships between objects.

We describe the details of our implementation of LLM self-repair in the supplementary.

6.4 Timing

The scene template generation stage of our pipeline takes 9.5s, generating a PSDL layout program takes 19.2s, and error correction via program search takes 9.3s, on average. Overall, it takes 38s on average to synthesize a scene using our method. In comparison, synthesizing a DeclBase scene takes 40.8s on average, which consists of 10s for synthesizing a declarative layout program, 21.3s for the solver, and shares the template generation stage. Therefore, the average run time of our system is comparable to declarative systems.

Table 6 reports timings for various error correction schemes: gradient descent is the fastest, while LLM self-repair is more than an order of magnitude slower than our method. Program search for PSDL programs is nearly 3 times faster than for the basic imperative programs because PSDL's loops and variables lower the dimensionality of the search space.

7 CONCLUSION AND FUTURE WORK

We introduced Procedural Scene Description Language (PSDL) and a complementary, LLM-free error-correction strategy that together re-establish the strengths of imperative scene layout programs while overcoming their fragility to LLM errors. By embedding explicit geometric relationships, shared variables, and structured control flow directly into a lightweight Python DSL, our approach enables fast, solver-free execution and yields a compact, semantically meaningful search space for symbolic repair. Across a diverse benchmark of open-universe prompts, perceptual studies and a new VLM-based automatic metric show that PSDL with program search delivers layouts that people prefer to those produced by state-of-the-art declarative and imperative baselines — especially in large, highly structured scenes — without incurring additional LLM calls.

Limitations. First, our local-search repair currently edits only numeric constants and facing directions; more complex semantic flaws (e.g., swapping object identities or swapping x and y coordinates) remain out of scope. Second, the loss we minimize does not explicitly encode higher-order aesthetics such as line-of-sight, walkability or affordances beyond support and collision. Third, while the VLM-based evaluator aligns better with human judgments than prior automatic scores, it is itself an empirical proxy whose biases may steer future systems toward its own blind spots. Finally, our experiments rely on a fixed object retrieval module and constrain orientations to four cardinals; relaxing either assumption will likely expose new challenges for both generation and correction.

Future Work. While scene generation has received significant attention in research, much of it has focused on static scenes. A natural next step is to extend scene synthesis to dynamic scenes, where objects interact or evolve over time. As the generation tasks grow in complexity, simplifying the coding process for LLMs, as done with our procedural approach, will become increasingly important.

REFERENCES

Rio Aguina-Kang, Maxim Gumin, Do Heon Han, Stewart Morris, Seung Jean Yoo, Aditya Ganeshan, R. Kenny Jones, Qiuhong Anna Wei, Kailiang Fu, and Daniel Ritchie. 2024. Open-Universe Indoor Scene Generation using LLM Program Synthesis and Uncurated Object Databases. arXiv:2403.09675 [cs.CV] https://arxiv.org/abs/2403.09675

Kiran Bhat, Nishchaie Khanna, Karun Channa, Tinghui Zhou, Yiheng Zhu, Xiaoxia Sun, Charles Shang, Anirudh Sudarshan, Maurice Chu, Daiqing Li, and et al. 2025. Cube: A Roblox View of 3D Intelligence. *arXiv preprint arXiv:2503.15475* (March 2025). https://doi.org/10.48550/arXiv.2503.15475

Angel X. Chang, Mihail Eric, Manolis Savva, and Christopher D. Manning. 2017. SceneSeer: 3D Scene Design with Natural Language. arXiv:1703.00050 [cs.GR] https://arxiv.org/abs/1703.00050

Angel X. Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, Jianxiong Xiao, Li Yi, and Fisher Yu. 2015. ShapeNet: An Information-Rich 3D Model Repository. *arXiv:1512.03012* (2015).

Jaemin Cho, Yushi Hu, Roopal Garg, Peter Anderson, Ranjay Krishna, Jason Baldridge, Mohit Bansal, Jordi Pont-Tuset, and Su Wang. 2024. Davidsonian Scene Graph: Improving Reliability in Fine-Grained Evaluation for Text-to-Image Generation. In *ICLR*.

Bob Coyne and Richard Sproat. 2001. WordsEye: an automatic text-to-scene conversion system. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '01)*. Association for Computing Machinery, New York, NY, USA, 487–496. https://doi.org/10.1145/383259.383316

Matt Deitke, Ruoshi Liu, Matthew Wallingford, Huong Ngo, Oscar Michel, Aditya Kusupati, Alan Fan, Christian Laforte, Vikram Voleti, Samir Yitzhak Gadre, Eli VanderBilt, Aniruddha Kembhavi, Carl Vondrick, Georgia Gkioxari, Kiana Ehsani, Ludwig Schmidt, and Ali Farhadi. 2023. Objaverse-XL: A Universe of 10M+ 3D Objects. *arXiv preprint arXiv:2307.05663* (2023).

Matt Deitke, Dustin Schwenk, Jordi Salvador, Luca Weihs, Oscar Michel, Eli VanderBilt, Ludwig Schmidt, Kiana Ehsani, Aniruddha Kembhavi, and Ali Farhadi. 2022. Objaverse: A Universe of Annotated 3D Objects. *arXiv preprint arXiv:2212.08051* (2022).

Weixi Feng, Wanrong Zhu, Tsu-Jui Fu, Varun Jampani, Arjun Reddy Akula, Xuehai He, S Basu, Xin Eric Wang, and William Yang Wang. 2023. LayoutGPT: Compositional Visual Planning and Generation with Large Language Models. In *Thirty-seventh Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=Xu8aG5Q8M3

Matthew Fisher, Daniel Ritchie, Manolis Savva, Thomas Funkhouser, and Pat Hanrahan. 2012. Example-based synthesis of 3D object arrangements. *ACM Transactions on Graphics (TOG)* 31, 6 (2012), 135:1–11.

Rao Fu, Zehao Wen, Zichen Liu, and Srinath Sridhar. 2024. AnyHome: Open-Vocabulary Generation of Structured and Textured 3D Homes. In *Proceedings of the European Conference on Computer Vision (ECCV)*.

Ziniu Hu, Ahmet Iscen, Aashi Jain, Thomas Kipf, Yisong Yue, David A Ross, Cordelia Schmid, and Alireza Fathi. 2024. SceneCraft: an LLM agent for synthesizing 3D scenes as blender code. In *Proceedings of the 41st International Conference on Machine Learning* (Vienna, Austria) *(ICML'24)*. JMLR.org, Article 776, 31 pages.

HypeHype. 2024. Asset Library | HypeHype Learning Hub. https://learn.hypehype.com/en/editor-overview/ui-basics/asset-library. Accessed: 2025-01-22.

Z Sadeghipour Kermani, Zicheng Liao, Ping Tan, and H Zhang. 2016. Learning 3D Scene Synthesis from Annotated RGB-D Images. In *Computer Graphics Forum*, Vol. 35. 197–206.

Milin Kodnongbua, Lawrence H. Curtis, and Adriana Schulz. 2024. Zero-shot Sequential Neuro-symbolic Reasoning for Automatically Generating Architecture Schematic Designs. arXiv:2402.00052 [cs.AI] https://arxiv.org/abs/2402.00052

Manyi Li, Akshay Gadi Patil, Kai Xu, Siddhartha Chaudhuri, Owais Khan, Ariel Shamir, Changhe Tu, Baoquan Chen, Daniel Cohen-Or, and Hao Zhang. 2018. GRAINS: Generative Recursive Autoencoders for INdoor Scenes. *CoRR* arXiv:1807.09193 (2018).

Yuan Liang, Song-Hai Zhang, and Ralph Robert Martin. 2017. Automatic Data-Driven Room Design Generation. In *Next Generation Computer Animation Techniques*, Jian Chang, Jian Jun Zhang, Nadia Magnenat Thalmann, Shi-Min Hu, Ruofeng Tong, and Wencheng Wang (Eds.). Springer International Publishing, Cham, 133–148.

Chenguo Lin and Yadong Mu. 2024. InstructScene: Instruction-Driven 3D Indoor Scene Synthesis with Semantic Graph Prior. In *Proceedings of the 12th International Conference on Learning Representations (ICLR)*. arXiv:2402.04717 [cs.CV] https://arxiv.org/abs/2402.04717 Spotlight presentation.

Zhiqiu Lin, Deepak Pathak, Baiqi Li, Jiayao Li, Xide Xia, Graham Neubig, Pengchuan Zhang, and Deva Ramanan. 2024. Evaluating Text-to-Visual Generation with Image-to-Text Generation. *arXiv preprint arXiv:2404.01291* (2024).

Gabrielle Littlefair, Niladri Shekhar Dutt, and Niloy J. Mitra. 2025. FlairGPT: Repurposing Large Language Models for Interior Designs. *Comput. Graph. Forum* 44, 2, Article e70036 (2025), 14 pages. https://doi.org/10.1111/cgf.70036

Liane Makatura, Michael Foshey, Bohan Wang, Felix HähnLein, Pingchuan Ma, Bolei Deng, Megan Tjandrasuwita, Andrew Spielberg, Crystal Elaine Owens, Peter Yichen Chen, Allan Zhao, Amy Zhu, Wil J Norton, Edward Gu, Joshua Jacob, Yifei Li, Adriana Schulz, and Wojciech Matusik. 2023. How Can Large Language Models Help Humans in Design and Manufacturing? arXiv:2307.14377 [cs.CL]

Paul Merrell, Eric Schkufza, Zeyang Li, Maneesh Agrawala, and Vladlen Koltun. 2011. Interactive furniture layout using interior design guidelines. In *ACM SIGGRAPH 2011 Papers* (Vancouver, British Columbia, Canada) *(SIGGRAPH '11)*. Association for Computing Machinery, New York, NY, USA, Article 87, 10 pages. https://doi.org/10.1145/1964921.1964982

Başak Melis Öcal, Maxim Tatarchenko, Sezer Karaoğlu, and Theo Gevers. 2024. SceneTeller: Language-to-3D Scene Generation. In *Computer Vision – ECCV 2024 (Lecture Notes in Computer Science, Vol. 15143)*. Springer, 362–378. https://doi.org/10.1007/978-3-031-73013-9_21

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2024. Is Self-Repair a Silver Bullet for Code Generation?. In *International Conference on Learning Representations (ICLR)*.

Liangming Pan, Michael Saxon, Wenda Xu, Deepak Nathani, Xinyi Wang, and William Yang Wang. 2023. Automatically Correcting Large Language Models: Surveying the landscape of diverse self-correction strategies. arXiv:2308.03188 [cs.CL] https://arxiv.org/abs/2308.03188

Despoina Paschalidou, Amlan Kar, Maria Shugrina, Karsten Kreis, Andreas Geiger, and Sanja Fidler. 2021. ATISS: Autoregressive Transformers for Indoor Scene Synthesis. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Gabriel Peyré and Marco Cuturi. 2019. *Computational Optimal Transport*. Now Publishers / MIT Press.

Siyuan Qi, Yixin Zhu, Siyuan Huang, Chenfanfu Jiang, and Song-Chun Zhu. 2018. Human-centric Indoor Scene Synthesis Using Stochastic Grammar. In *Conference on Computer Vision and Pattern Recognition (CVPR)*.

Ariadna Quattoni and Antonio Torralba. 2009. Recognizing indoor scenes. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*.

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. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 8748–8763.

Daniel Ritchie, Kai Wang, and Yu an Lin. 2019. Fast and Flexible Indoor Scene Synthesis via Deep Convolutional Generative Models. In *CVPR 2019*.

Jiapeng Tang, Nie Yinyu, Markhasin Lev, Dai Angela, Thies Justus, and Matthias Nießner. 2023. DiffuScene: Scene Graph Denoising Diffusion Probabilistic Model for Generative Indoor Scene Synthesis. In *arxiv*.

Kai Wang, Yu-An Lin, Ben Weissmann, Manolis Savva, Angel X Chang, and Daniel Ritchie. 2019. Planit: Planning and instantiating indoor scenes with relation graph and spatial prior networks. *ACM Transactions on Graphics (TOG)* 38, 4 (2019), 132.

Kai Wang, Manolis Savva, Angel X. Chang, and Daniel Ritchie. 2018. Deep Convolutional Priors for Indoor Scene Synthesis. In *Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*.

Xinpeng Wang, Chandan Yeshwanth, and Matthias Nießner. 2020. SceneFormer: Indoor Scene Generation with Transformers. *arXiv preprint arXiv:2012.09793* (2020).

Yue Yang, Fan-Yun Sun, Luca Weihs, Eli VanderBilt, Alvaro Herrasti, Winson Han, Jiajun Wu, Nick Haber, Ranjay Krishna, Lingjie Liu, Chris Callison-Burch, Mark Yatskar, Aniruddha Kembhavi, and Christopher Clark. 2023. Holodeck: Language Guided Generation of 3D Embodied AI Environments. *arXiv preprint arXiv:2312.09067* (2023).

Yi-Ting Yeh, Lingfeng Yang, Matthew Watson, Noah D. Goodman, and Pat Hanrahan. 2012. Synthesizing open worlds with constraints using locally annealed reversible jump MCMC. 31, 4, Article 56 (jul 2012), 11 pages. https://doi.org/10.1145/2185520.2185552

Lap-Fai Yu, Sai Kit Yeung, Chi-Keung Tang, Demetri Terzopoulos, Tony F. Chan, and Stanley Osher. 2011. Make it home: automatic optimization of furniture arrangement. *ACM Transactions on Graphics (TOG)* 30, 4 (2011), 86:1–12.

Yunzhi Zhang, Zizhang Li, Matt Zhou, Shangzhe Wu, and Jiajun Wu. 2024. The Scene Language: Representing Scenes with Programs, Words, and Embeddings. arXiv:2410.16770 [cs.CV] https://arxiv.org/abs/2410.16770

Zaiwei Zhang, Zhenpei Yang, Chongyang Ma, Linjie Luo, Alexander Huth, Etienne Vouga, and Qixing Huang. 2018. Deep Generative Modeling for Scene Synthesis via Hybrid Representations. *CoRR* abs/1808.02084 (2018). arXiv:1808.02084 http://arxiv.org/abs/1808.02084

Yang Zhou, Zachary While, and Evangelos Kalogerakis. 2019. SceneGraphNet: Neural Message Passing for 3D Indoor Scene Augmentation. In *IEEE Conference on Computer Vision (ICCV)*.

Ata Çelen, Guo Han, Konrad Schindler, Luc Van Gool, Iro Armeni, Anton Obukhov, and Xi Wang. 2024. I-Design: Personalized LLM Interior Designer. arXiv:2404.02838 [cs.AI]

Table 7. Preference rates for each scene category in the forced-choice perceptual study, comparing our method against DeclBase and Holodeck.

Scene Type	DeclBase	Holodeck
Ours w/ EC vs.	82.9%	94.3%
Small	71.4%	100.0%
Medium	82.9%	91.4%
Large	90.5%	95.2%
Indoor	81.3%	95.8%
Outdoor	86.4%	90.9%
Realistic	84.0%	92.0%
Fantastical	80.0%	100.0%
Chaotic	74.2%	96.8%
Structured	89.7%	92.3%

A HOLODECK MODIFICATIONS

To focus on evaluating the quality of object layouts and eliminate any influence object selection might have on participant responses in the perceptual study, Holodeck's object selection module was modified to use only the same set of objects and sizes present in the corresponding scenes from our system. To avoid prompting the LLM for the object selection plan json, which resulted in hallucinations of objects and object sizes outside of the given constraints, a 'mock' json following the LLM output format was manually created and inserted into the system pipeline for the layout module to use. Because Holodeck's original pipeline handled both object selection and secondary object relations (e.g., placing monitors on desks) simultaneously, bypassing the LLM for object selection caused secondary relations to be omitted. As a result, Holodeck layouts for the 70 prompts used in the perceptual study lack secondary relations (e.g., computers appearing beside rather than on top of desks). We later use an updated configuration for the other set of 66 prompts, where secondary relations are included by prompting the LLM only for the object-to-object relations, without altering the fixed object list. Despite the two different configurations, the trend in preference rates of 'Ours' vs. 'Holodeck' in Table 4 are evident, and we do not have reason to believe that they would drastically change if the 70 scenes were re-run with secondary relations.

B LLM SELF-REPAIR

To implement LLM-based self-repair, we provide the language model with the original layout program and a description of the layout errors produced during execution. At each iteration, the LLM is prompted with a structured conversation: the system prompt describes the DSL and expected behaviors, the user provides the scene name, the assistant echoes the original layout code, and the user follows up with the list of detected errors (e.g., object overlaps, out-of-bounds issues, or support violations). The LLM then returns a revised layout program attempting to correct those issues.

This revised program is executed, and if errors remain, the process is repeated using the new scene as the next reference. We allow up to 6 correction iterations, as we observed that most successful repairs occur within the first few rounds and diminishing returns follow.

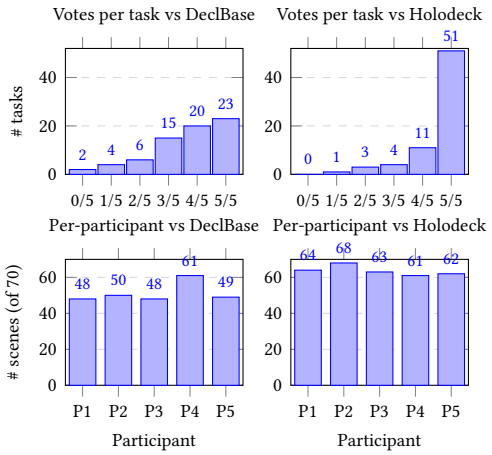

Fig. 3. Human preference study. Top row: per-task vote counts for our method (0/5–5/5) vs DeclBase (left) and Holodeck (right); each sums to 70 tasks. Bottom row: per-participant counts (out of 70) preferred for our method against each baseline.

Table 8. PSDL API (I): Types, attributes, and enumerations. Vector fields support component selection `.x`, `.y`, `.z`. Attributes marked R/W are writable via overloaded assignment; others are read-only inputs from the template.

Symbol	Kind	Type	R/W	Meaning
`scene`	special object	Object	R	Bounding cuboid of the whole scene; exposes `center`, `min`, `max`, `width`, `depth`, `height`.
`o`	scene object	Object	R	Any pre-instantiated object bound by the template (e.g., `chair`, `table`).
`o.center`	attribute	`vec3`	R/W	Geometric center of `o` in the current coordinate frame. Writing `o.center.{x|y|z}` moves `o` along that axis.
`o.min`	attribute	`vec3`	R/W	AABB minimum in the current frame; writing components (e.g., `o.min.x = ...`) translates `o` to satisfy the constraint.
`o.max`	attribute	`vec3`	R/W	AABB maximum in the current frame.
`o.width`	attribute	`float`	R	Object width (perpendicular to facing).
`o.depth`	attribute	`float`	R	Object depth (along facing).
`o.height`	attribute	`float`	R	Object height (upwards).
`o.facing`	attribute	`Facing`	R/W	Orientation of `o` relative to the current frame. Allowed assignments: `o.facing = dir` (set cardinal), `o.facing = p` (face *toward* object `p`; forward axis points from `o.center` to `p.center` and is quantized to the nearest cardinal), or `o.facing = p.facing` (copy `p`'s orientation). The stored value is always a cardinal.
`o.name`	attribute	`str`	R	Object identifier from the template.
`o.support`	attribute	`Support`	R	Physical support type from the template (`STANDING`, `WALL_MOUNTED`, `FLOATING`); not modifiable in PSDL.
`Facing`	enum	—	—	Cardinal orientation: `X_NEG`, `X_POS`, `Y_NEG`, `Y_POS`. (Assignment to `o.facing` is overloaded as described above; the stored value remains a cardinal.)
`Support`	enum	—	—	`STANDING`, `WALL_MOUNTED`, `FLOATING`.

Table 9. PSDL API (II): Statements/operators and assignment forms. Right-hand sides (RHS) are standard Python expressions composed of numbers, Python variables, arithmetic, and attribute reads (e.g., `p.center.x`, `q.min.z`, `scene.max.y`).

Construct	Form / Signature	Semantics and Examples
Local frame	`set_coordinate_frame(o)`	Set the current coordinate frame to object `o`: y-axis aligns with `o.facing`, x-axis is 90° clockwise from y, z-axis is up. Useful for building sub-layouts in canonical space. Example: `set_coordinate_frame(counter)`.
Center component assignment	`o.center.{x|y|z} = expr`	Moves `o` so the specified center component equals `expr` (in the current frame). Example: `chair.center.y = table.center.y`.
AABB component assignment	`o.min.{x|y|z} = expr` / `o.max.{x|y|z} = expr`	Translates `o` so the chosen AABB face matches `expr`. Example: `chair.max.x = table.min.x - 0.1`.
Facing assignment (cardinal)	`o.facing = dir`	Sets `o`'s orientation to a cardinal `dir`. Example: `register.facing = X_NEG`.
Facing assignment (face toward object)	`o.facing = p`	Rotates `o` to face object `p`; the forward axis points from `o.center` to `p.center`, and the resulting `o.facing` is quantized to the nearest cardinal. Example: `chair.facing = table`.
Facing assignment (copy orientation)	`o.facing = p.facing`	Copies the orientation of `p`. Example: `chair.facing = table.facing`.
RHS expressions	(Python)	Any Python arithmetic expression over scalars/variables and attribute reads from `scene` or previously placed objects. Examples: `scene.center.x + i * d`, `counter.min.x + (i+0.5) * counter.width`.
Control flow	(Python)	Ordinary Python loops and conditionals (`for`, `if`, `enumerate`, `range`, ...) to express repetition and symmetry. Example: `for i, stool in enumerate(stools): stool.center.x = counter.min.x + (i+1.0) / 4.0 * counter.width`.

Table 10. Comparison of coordination errors, out-of-bounds placements, and overlaps for different LLM configurations across three setups: explicit coordinate prediction, pre-correction, and post-correction.

Model	claude-3-5-sonnet-20241022			gpt-4o-2024-11-20			o1-2024-12-17			gemini-exp-1206		
	All	Bound	Ovl	All	Bound	Ovl	All	Bound	Ovl	All	Bound	Ovl
Ours w/o correction	17.57	1.56	10.76	17.80	1.14	11.19	17.29	2.36	10.70	14.67	1.19	10.41
Ours	**2.10**	**0.54**	**0.56**	**3.34**	**0.61**	**0.64**	**3.24**	**0.63**	**1.17**	**3.14**	**0.61**	**1.14**

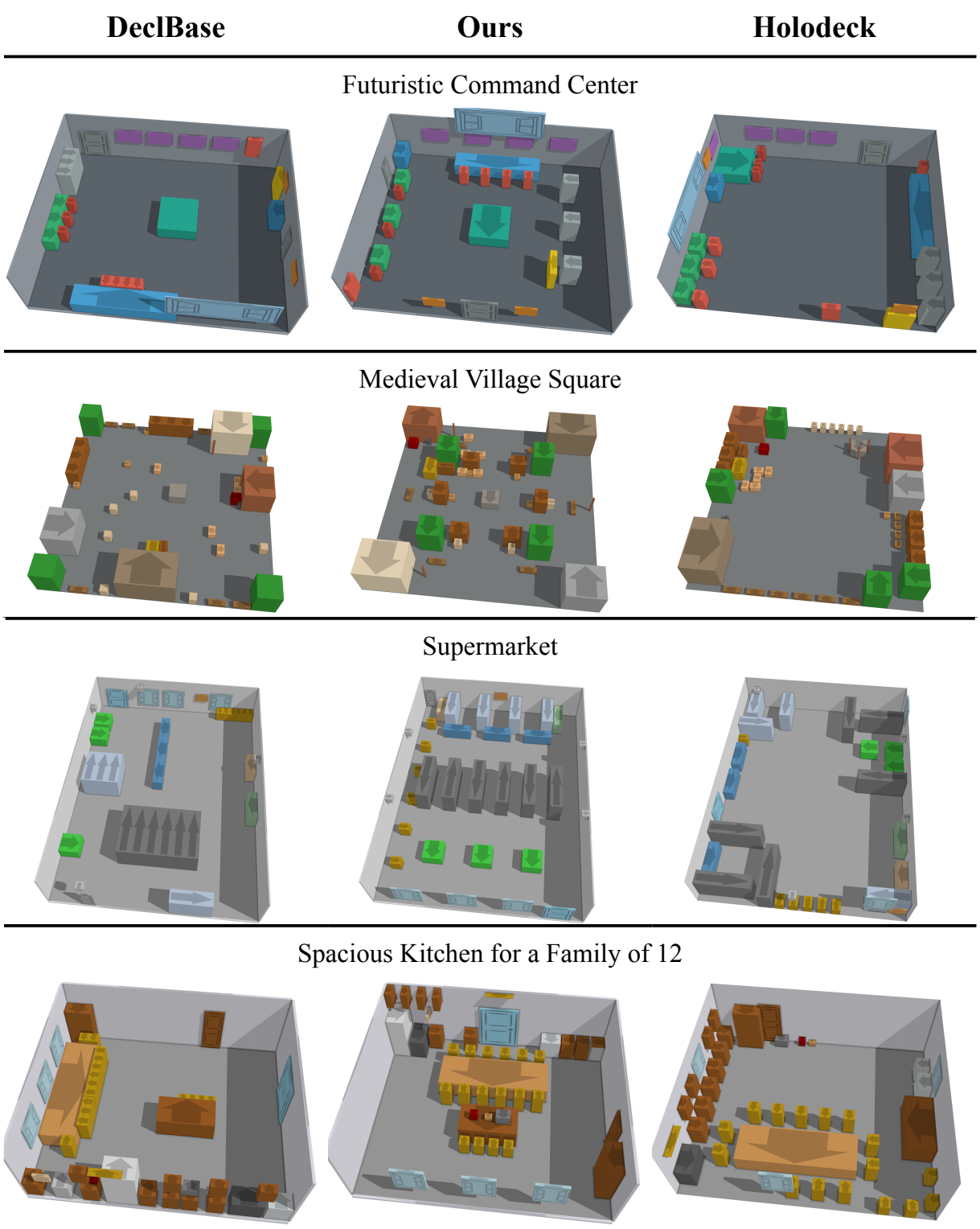

Fig. 4. Qualitative comparisons between our method, DeclBase, and Holodeck. Our method and Holodeck uses `gpt-4o`, while DeclBase uses `claude-3-5-sonnet-20241022`. See the supplemental for a comparison between our method and DeclBase only using `claude-3-5-sonnet-20241022`.

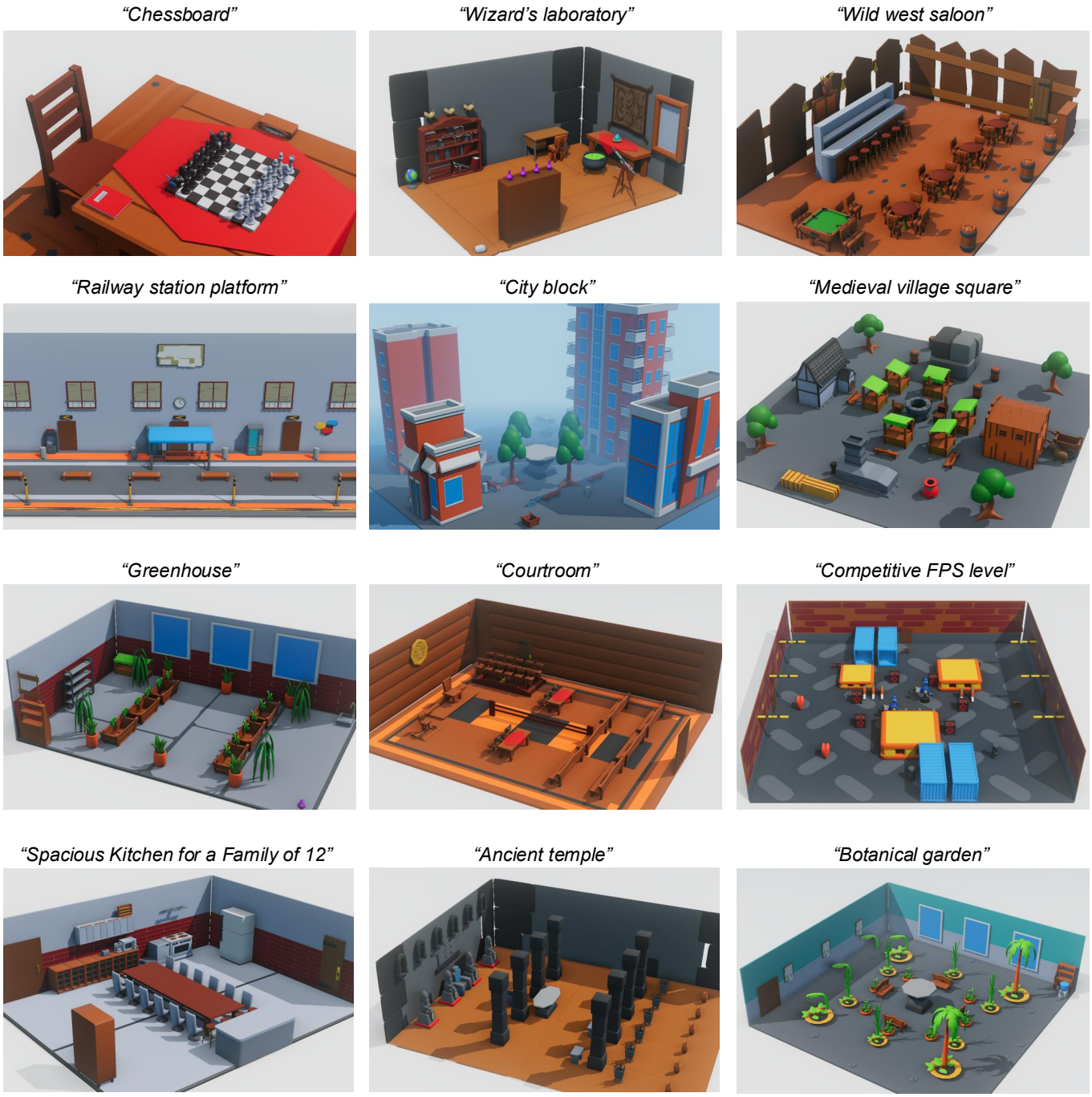

Fig. 5. More scenes synthesized using our imperative layout generation method with error correction.