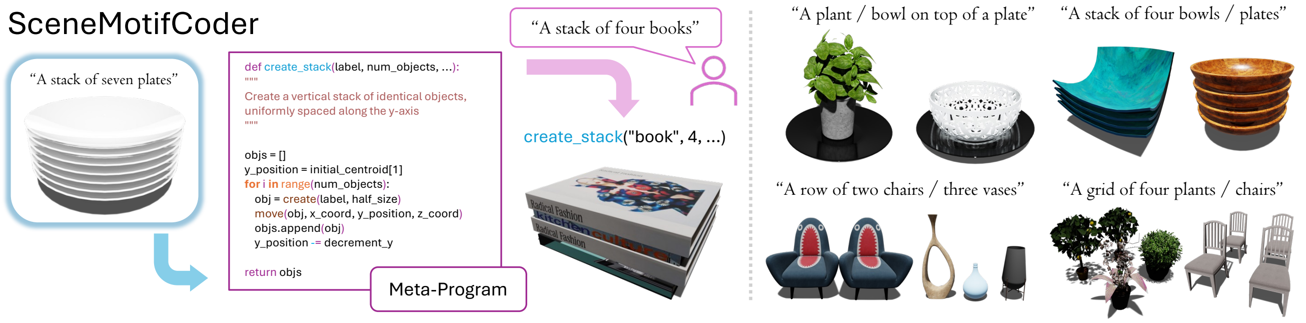
# SceneMotifCoder: Example-driven Visual Program Learning for Generating 3D Object Arrangements

Hou In Ivan Tam[1] Hou In Derek Pun[1] Austin T. Wang[1] Angel X. Chang[1,2] Manolis Savva[1]
[1]Simon Fraser University [2]Alberta Machine Intelligence Institute (Amii)
3dlg-hcvc.github.io/smc


SceneMotifCoder
"A stack of seven plates"
"A stack of four books"
Meta-Program
"A plant / bowl on top of a plate"
"A stack of four bowls / plates"
"A row of two chairs / three vases"
"A grid of four plants / chairs"

```
def create_stack(label, num_objects, ...):
"""
Create a vertical stack of identical objects,
uniformly spaced along the y-axis
"""

objs = []
y_position = initial_centroid[1]
for i in range(num_objects):
   obj = create(label, half_size)
   move(obj, x_coord, y_position, z_coord)
   objs.append(obj)
   y_position -= decrement_y

return objs
```

```
create_stack("book", 4, ...)
```

Figure 1. We present SceneMotifCoder (SMC), an example-driven approach for generating 3D object arrangements. **Left**: given an example arrangement and text description, SMC writes a meta-program with arguments that capture the underlying spatial patterns of the arrangement. Then, to generate a new arrangement, SMC takes a text description as input, retrieves a meta-program, and writes a function call with appropriate arguments to execute the meta-program. Through mesh retrieval and a geometry optimization step, the final arrangement is physically plausible and conforms to the text description. **Right**: Example output 3D object arrangements for various input descriptions. SMC generates arrangements with programs learned from 1 to 3 examples.

## Abstract

*Despite advances in text-to-3D generation methods, generation of multi-object arrangements remains challenging. Current methods exhibit failures in generating physically plausible arrangements that respect the provided text description. We present SceneMotifCoder (SMC), an example-driven framework for generating 3D object arrangements through visual program learning. SMC leverages large language models (LLMs) and program synthesis to overcome these challenges by learning visual programs from example arrangements. These programs are generalized into compact, editable meta-programs. When combined with 3D object retrieval and geometry-aware optimization, they can be used to create object arrangements varying in arrangement structure and contained objects. Our experiments show that SMC generates high-quality arrangements using meta-programs learned from few examples. Evaluation results demonstrates that object arrangements generated by SMC better conform to user-specified text descriptions and are more physically plausible when compared with state-of-the-art text-to-3D generation and layout methods.*

## 1. Introduction

Digital 3D representations of indoor scenes are essential for many domains including interior design, game development, visual effects, virtual and augmented reality, and embodied AI simulations. However, authoring these scenes is laborious and requires 3D modeling expertise. Thus, indoor scene synthesis has been a research focus for more than a decade [2, 6, 23]. Recent progress enabled conditional scene generation given a room type [5], floor plan [28], or graph specifying objects and relations [38].

Despite this progress, existing scene synthesis methods focus on large furniture objects such as beds, cabinets, tables, and chairs and typically ignore smaller objects. For example, generated scene layouts lack plates, cups or decorative objects on dining tables, making the scenes look unrealistically empty. Such objects are ubiquitous in real life and are vital to making synthetic scenes more realistic.

Nevertheless, modeling these object arrangements is challenging. Existing 3D scene datasets rarely capture small-scale arrangements, making standard generative model training strategies impractical. Additionally, the highly compositional and geometrically tight nature of these arrangements complicates placement of objects. For instance, plates in a stack of plates are in contact with plates above and below without gaps or intersections, requiring precise placement that is hard to achieve with existing layout generation methods.

In this paper, we present *SceneMotifCoder*, an example-driven framework for 3D object arrangement generation based on visual program learning. We show how the code generation capabilities of Large Language Models (LLMs) can be leveraged to create visual programs that capture abstractions of common 3D object arrangements, or *motifs*, from few examples. These motif visual programs then allow generation of new object arrangements exhibiting substantially different content (i.e. different objects) and structure (i.e. different numbers and positions of objects), from text descriptions provided by a user.

Our key insight is that object arrangements have underlying motifs that can be extracted from a few examples. These motifs can be treated as templates for creating new arrangements. To this end, programs are an ideal representation as the control flow compactly and interpretably captures the motif structure. By generalizing motif programs into meta-programs, we further concretize common structures and parameters within a motif type and allow for easy instantiation of new arrangements.

We show that SceneMotifCoder (SMC) requires only 1 to 3 examples to learn a meta-program. By leveraging these re-usable meta-programs we can significantly streamline 3D content creation, especially for designers who routinely create layouts using a 3D object library. Unlike black box 3D generative models, SMC's human-readable programs allow user modification to intuitively adjust generation results. Furthermore, SMC's generation capability is easily extendable. Given new examples, SMC learns new motifs without resource intensive and time consuming re-training. Thus, the learned meta-program library can efficiently and progressively grow into a comprehensive knowledge base for a wide range of realistic object arrangements.

We evaluate SMC against state-of-the-art layout and text-to-3D generative models and show that our object arrangements better conform to input descriptions and exhibit higher physical plausibility, measured both algorithmically and through human perceptual studies. In summary: 1) we propose the SMC framework, an example-driven approach for learning visual programs that represent 3D object arrangements; 2) we use the learned programs to generate complex and realistic arrangements for diverse objects and show how our framework admits for interpretable and easily editable arrangement generation using simple text descriptions; and 3) we systematically evaluate against state-of-the-art layout and text-to-3D generation methods and show our approach outperforms prior work in both alignment to text description and physical plausibility.

## 2. Related Work

We summarize prior work on various strategies for 3D scene synthesis: compositional 3D layout generation using object retrieval, layout generation leveraging LLMs, text-to-3D generation using differentiable rendering, and visual program synthesis methods.

**3D layout generation.** There is much work on layout generation, ranging from rule-based [23, 47], to data-driven [6], to neural network methods [37]. While early work [6] focused on cluttered object arrangements, more recent work using deep learning [19, 27, 28, 34, 37, 38, 40, 51] focuses on larger furniture-sized objects, typically ignoring smaller objects on tabletops or other surfaces. One reason is that available 3D scene datasets mostly contain furniture and are sparsely populated with 'detail objects'. Our approach is tailored for generation of such detail object arrangements.

**Layout generation with LLMs.** A recent trend uses LLMs for more open-vocabulary layout generation and generation conditioned on text input. Early work relied on custom rules [4] or data-driven priors [2, 3] to determine spatial relations and retrieve objects with keyword search. Recent work [1, 5, 11, 12, 20, 33, 41, 45, 49] leverages LLMs more flexibly 1) as a source of spatial priors, 2) to retrieve 3D assets based on vision-language embeddings, or 3) to generate modeling software API calls that generate 3D scenes. In contrast, we combine LLMs with visual program synthesis to learn self-contained reusable programs that generate 3D object layouts given a small set of examples. These programs capture arrangement *motifs* that give the user more control and editability compared to relying on API calls directly generated by LLMs.

**Text-to-3D generation.** Text-to-3D-shape generation models have advanced rapidly thanks to recent breakthroughs in text-to-image diffusion. Various works demonstrated the use of image diffusion models to generate 3D shapes [24, 30, 36, 46, 50]. While some work attempted to generate scenes as single unstructured geometry [10], the output cannot be easily decomposed and manipulated as separate objects. Thus, recent works generate multiple objects either from a prespecified layout [21, 29] or by tackling both the layout (often using an LLM to determine how objects should be arranged) and generation together [9, 35, 48]. Despite their success, these methods often generate incorrect numbers of objects, fail to respect spatial relationships, or suffer from the 'Janus problem' (multiple implausible front sides). Our work does not rely on a text-to-image diffusion model to obtain spatial relationship priors between objects. Rather, given an object arrangement as an example we directly learn the spatial arrangement pattern as a visual program and use it to instantiate new object arrangements.

**Visual program synthesis.** Visual programs are attractive for 3D shape representation as they are more interpretable and easily editable than low-level 3D representations. In addition, they are easier to generate with LLMs due to the large volume of code in LLM training data. Synthesis of visual programs has been applied to computer-aided design (CAD) sketches [8, 26, 31] and CAD models [18, 42, 44]. It has also been used to describe 3D object arrangements using program abstraction through interactive or in-context learning [22, 39]. A recent line of work [7, 14–16] applied visual programs to encode 3D shapes as primitives (e.g., a chair as a set of cuboids). In particular, ShapeMOD [15] and ShapeCoder [16] demonstrate programs can extract abstractions of high-level 3D object geometry patterns, facilitating downstream editing tasks. Our work is similar to Liu et al. [22] in that we use visual programs to encode collections of objects. However, we focus on the challenging task of learning dense 3D spatial arrangements given a text description and a few examples, rather than inferring programs from simple shapes (e.g., cuboids), and we generalize our inferred programs to create new arrangements.

## 3. Method

In our problem setting, a user provides examples of an object arrangement $A$ paired with text $T$ describing the arrangement. The object arrangement consists of a set of labeled 3D assets with appropriate transformations. Our method then takes these input examples consisting of a few $(T, A)$ pairs and produces a program $P$ that can generate new variations of arrangements (with different types of objects) for that motif type. The learned program is stored in a library, to be retrieved at inference (i.e. generation) time, and used to generate new arrangements. Our visual programs use a simple Domain Specific Language (DSL) based on Python (Sec. 3.1).

Our approach, SceneMotifCoder (SMC), consists of two phases (see Fig. 2): the *learning* phase (Sec. 3.2) where we extract visual programs from a few (1-3) examples, and the *inference* phase (Sec. 3.3) where the program is retrieved and applied to a new text description. In the *learning* phase, we first represent the arrangement as a *naïve program* that specifies the exact objects to be instantiated with their label, position, and orientation. Then, we use an LLM to do a series of transformations that go from the list-based *naïve program* to a more structured *motif program* that specifies the given arrangement but uses higher-level constructs such as loops, to a final *meta-program* (i.e. function with arguments) that we store in the meta-program library. During *inference*, we retrieve the appropriate meta-program and use an LLM to determine appropriate call arguments. Based on the arrangement, we retrieve 3D objects and perform a geometry-aware optimization before outputting the final arrangement. This strategy leverages the strong code generation capabilities of LLMs to produce the meta-programs, while still relying on the concrete mesh retrieval and geometric optimization to create the final arrangement.

### 3.1. Domain Specific Language

We use Python as the basis for our DSL for its readability and high-level constructs. In addition to built-in Python functions, the numpy[1] library and the following three constructs are available to describe motifs using programs:

1. `create(label,size)` : Instantiate object of type `label` with dimensions `size` $\in \mathbb{R}^3$ at scene origin.
2. `move(object,x,y,z)` : Position `object` at location $(x, y, z) \in \mathbb{R}^3$ in scene coordinates.
3. `rotate(object,axis,angle)` : Orient `object` through rotation by `angle` around `axis` in object local coordinates.

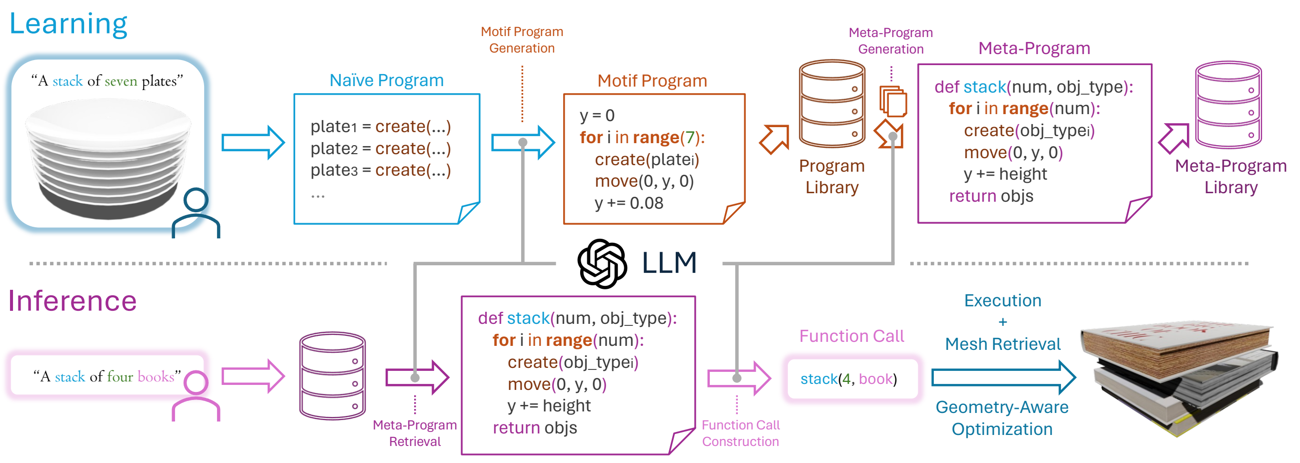
We avoid higher-level constructs such as relative position and relative rotation between objects, as we want to investigate whether an LLM can capture such relationships in the program structure and using arithmetic operations.

### 3.2. Learning Phase

SceneMotifCoder (SMC) learns to write meta-programs to capture the underlying motifs of object arrangements that can be used to generate new arrangements in the learning phase. To do so, we generate and refine the program in three stages: 1) we first generate a *naïve program* $P_0$ for each example arrangement based on the specific positioning and rotation observed in the arrangement; 2) we then prompt the LLM to make observations about patterns in the program to guide the LLM to rewrite the program to a more structured *motif program* $P_{\text{motif}}$; and 3) finally, given a small set of motif programs (of the same type), we ask the LLM to create a *meta-program* $P$ that captures commonalities across the motif programs, and extracts important axes of variation as parameters. The top half of Fig. 2 illustrates this phase. For (2) and (3), as the LLM may make errors, we adopt an observe-generate-validate approach, in which we ask the LLM to make observations about the input, and after the LLM generates an initial program, we validate whether the generated program is executable and preserves the required functionality. We iteratively prompt the LLM to refine the generated program until the validation is passed (typically within three iterations). We also use chain-of-thought style prompting to encourage the LLM to reflect and reason about the program it is generating. Figure 3 shows a conceptual outline of these steps, which we describe in detail below.

**Naïve-program extraction.** Given a text description and the corresponding object arrangement in individual meshes, SMC first extracts each object's label, bounding box centroid, and local coordinate axes from the input arrangement. These attributes indicate the objects that exist in the arrangement, as well as their positions and local rotations. Using these attributes, SMC writes a *naïve program* $P_0$ for the arrangement by creating and posing each object one by

[1] https://numpy.org



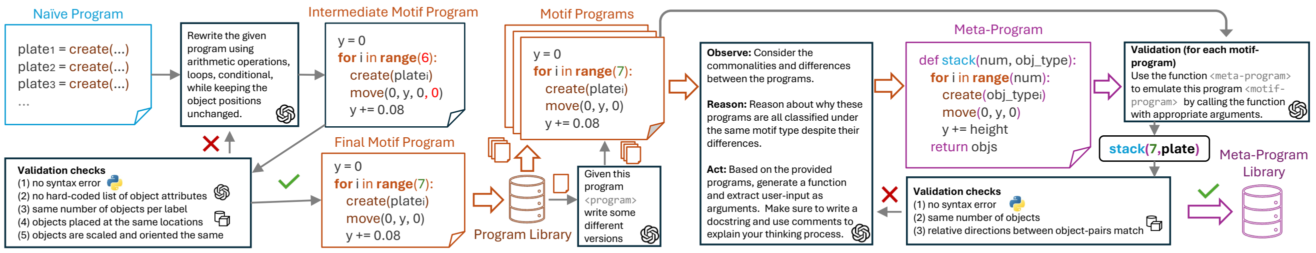
```
plate1 = create(...)
plate2 = create(...)
plate3 = create(...)
...
```

```
y = 0
for i in range(7):
    create(platei)
    move(0, y, 0)
    y += 0.08
```

```
def stack(num, obj_type):
    for i in range(num):
        create(obj_typei)
        move(0, y, 0)
        y += height
    return objs
```

```
stack(4, book)
```

Figure 2. The SceneMotifCoder (SMC) framework operates in two phases. **Top:** In the learning phase, given an object arrangement and its text description, SMC extracts a naïve program which is refined into a more structured motif program. The spatial patterns in the motif are then abstracted into a meta-program with reusable arguments to enable editing and generalization. **Bottom:** In the inference phase, given a text description as input, SMC selects a meta-program from the program library. Then, a new object arrangement is generated by writing a function call that executes the meta-program with appropriate arguments. Mesh retrieval and geometry-aware optimization then produces the final output which is a new object arrangement conforming to the input description.



```
plate1 = create(...)
plate2 = create(...)
plate3 = create(...)
...
```

Rewrite the given program using arithmetic operations, loops, conditional, while keeping the object positions unchanged.

```
y = 0
for i in range(6):
    create(platei)
    move(0, y, 0, 0)
    y += 0.08
```

**Validation checks**
(1) no syntax error
(2) no hard-coded list of object attributes
(3) same number of objects per label
(4) objects placed at the same locations
(5) objects are scaled and oriented the same

```
y = 0
for i in range(7):
    create(platei)
    move(0, y, 0)
    y += 0.08
```

Given this program `<program>` write some different versions

```
y = 0
for i in range(7):
    create(platei)
    move(0, y, 0)
    y += 0.08
```

**Observe:** Consider the commonalities and differences between the programs.

**Reason:** Reason about why these programs are all classified under the same motif type despite their differences.

**Act:** Based on the provided programs, generate a function and extract user-input as arguments. Make sure to write a docstring and use comments to explain your thinking process.

```
def stack(num, obj_type):
    for i in range(num):
        create(obj_typei)
        move(0, y, 0)
        y += height
    return objs
```

**Validation (for each motif-program)**
Use the function `<meta-program>` to emulate this program `<motif-program>` by calling the function with appropriate arguments.

```
stack(7,plate)
```

**Validation checks**
(1) no syntax error
(2) same number of objects
(3) relative directions between object-pairs match

Figure 3. SceneMotifCoder learning phase overview. **Left**: A naïve program converted to a motif program and added to the program library. **Right**: The motif program is generalized through iterative refinement to a meta-program and added to the meta-program library.

one in a flat list of statements. This program serves as the foundation for subsequent reasoning tasks.

**Motif-program generation.** Using the naïve program $P_0$, SMC then prompts an LLM to make four high-level observations about $P_{\text{motif}}$. These include: 1) the number of objects and their labels; 2) the relative displacements between objects; 3) whether there are any commonsense patterns (e.g., symmetry, repetition); and 4) whether there are any significant patterns in the x, y, and z coordinates. These observations guide the LLM to look at useful patterns for extracting the underlying motif. With these observations made, SMC then asks the LLM to divide the objects into groups, disentangling the overall motif into smaller spatial patterns that are easier to express as code. Finally, SMC prompts the LLM to classify the arrangement into one of the motif types. See the supplement for the LLM prompts.

The next step rewrites $P_0$ into a more structured *motif program* $P_{\text{motif}}$ to better capture the motif's spatial patterns. This more structured program makes it easier to reason about high-level patterns across different $P_{\text{motif}}$ for generating a final meta-program that can handle variations. We instruct the LLM to rewrite $P_0$ using arithmetic operations and programming constructs such as loops and conditions, while keeping object placements unchanged.

*Validation.* We then validate the rewritten program by executing it and checking against the original arrangement using the following criteria: (1) no syntax error, (2) no hard-coded list of object attributes, (3) same number of objects per label, (4) objects placed at same locations, and (5) objects scaled and oriented the same. These criteria ensure that: the program is executable, i.e. there are no Python interpreter errors (1), and the rewritten program is extracting the underlying motif instead of memorizing object placements (2). We validate (2) by asking the LLM to judge the rewritten program. Criteria (3) through (5) ensure the program does not deviate from the original arrangement, and are validated using simple logic checks and by computing volumetric Intersection-over-Union (IoU)s between the bounding boxes of the motif from the rewritten program and the original arrangement. If any criterion fails, we provide feedback regarding what failed to the LLM (e.g., listing incorrectly placed object coordinates and corresponding correct coordinates), and try rewriting the program again. These validations are crucial for the program to represent the input arrangement's motif precisely. Once all criteria pass, SMC stores the rewritten program, a *motif program*, in the program library under the classified motif type.

**Meta-program generation.** To write a meta-program that captures the essence of the motif type, we need to reason about what *changes* and what *remains unchanged* across different object arrangement instances of the motif type. To this end, SMC gathers all motif programs of the same type from the program library, and provides them as input to the LLM, asking the LLM to observe commonalities and differences between them. If there is only one motif program in the library, we prompt the LLM to hypothesize different versions of the program and make observations based on them. We further prompt the LLM to reason about why the programs are classified under the same motif type despite their differences. The goal is to extract the fundamental spatial patterns of the motif type from the programs and ignore extraneous elements. This then serves as guidance for writing the meta-program.

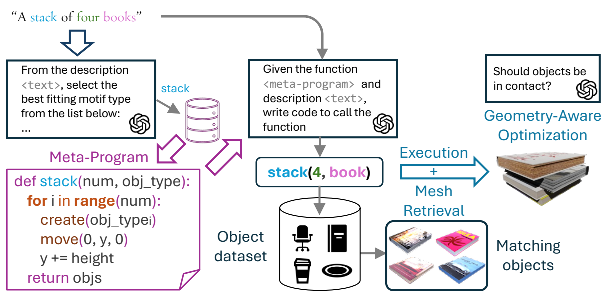
The last step in the learning phase is to write a generalized meta-program for the motif type with arguments that can be called to create a new motif of this type. More specifically, SMC tasks the LLM to extract user-controllable parameters (e.g., number of objects) from the programs and include them as arguments in the meta-program function signature. This ensures that the meta-program is flexible and can be used to generate new motifs given varying text descriptions. In the prompt, we specifically ask the LLM to write docstrings and comments that explain the code and serve as guides for using the meta-program.

*Validation.* As in the program rewrite step, we validate the meta-program by checking it against all motif programs it generalizes. We prompt the LLM to write one function call per program to execute the meta-program multiple times and check the following three criteria: (1) no syntax error; (2) same number of objects as in the corresponding motif program; (3) relative directions between all pairs of objects match those in the motif program. These validations ensure the meta-program captures a generalization of the spatial patterns of the motif type and that it can recreate the original motif programs. Once the meta-program is validated, SMC stores it in the meta-program library to be used to generate new motifs.

## 3.3. Inference Phase

In the inference phase, given a text description as input, SMC generates a new object arrangement using previously learned meta-programs. The steps involved are described below and outlined in Fig. 4.

**Retrieval.** Using the text description, SMC first retrieves the meta-program of the same motif type in the library by classifying the description into one of the motif types through an LLM prompt that enumerates the motif types and provides the text of the description.



From the description <text>, select the best fitting motif type from the list below: ...

Given the function <meta-program> and description <text>, write code to call the function

Should objects be in contact?

```
def stack(num, obj_type):
  for i in range(num):
    create(obj_typei)
    move(0, y, 0)
    y += height
  return objs
```

stack(4, book)

Figure 4. SceneMotifCoder inference phase overview. **From top left**: The input text is used to retrieve an appropriate meta-program from the library. Arguments are inferred to form a function call. The arrangement is then instantiated by executing the call, retrieving appropriate objects from a database, and optimizing the layout.

**Execution.** To execute the meta-program, we feed it along with the description to an LLM to construct a function call with suitable arguments. We check for syntax errors and execute the call to get a motif for the input text description. To instantiate the motif into an object arrangement, SMC retrieves meshes from an object dataset using the label and the bounding box dimensions of each object. We prompt the LLM for whether each of the objects can have multiple common orientations in real life scenarios. If yes, we rotate each candidate mesh around its local axes to determine the best orientation for matching the object's bounding box. Otherwise, we do not rotate. For each object, we rank all available meshes based on the difference in dimensions between the mesh and the object's bounding box, and randomly pick one of the top five candidates as the mesh for the object. Alternatively, we can retrieve meshes by finding the best match between an object description and renderings of the meshes using a pre-trained CLIP model (e.g., OpenCLIP [13]). See the supplement for a discussion.

**Geometry-aware optimization.** Since the retrieved objects may not match the specified dimensions, and the motif from executing the meta-program may contain inaccurate object placements, we optimize the final object placements with an iterative geometry-aware procedure. We start with the first two objects in the arrangement and add a new object in each iteration, until all objects are processed. In each iteration, we detect intersections between meshes and resolve them by moving the objects apart in the direction opposite to the contact points. We then prompt the LLM using the text description to check whether objects in the described arrangement should be in contact with neighbouring objects. If yes, we start an iterative ray-casting procedure to move the objects closer. In each ray-cast iteration, we sample points on the surface of a newly added object facing the nearest neighbour, and shoot rays in the direction of the neighbouring object. We use ray-mesh intersection tests to determine the intersection distances from the ray origins and move the object towards the ray direction by the minimum of the distances, dampened by a factor that grows for each iteration. Finally, once ray-cast iterations are finished, we cast rays from the object towards the ground plane to determine if it is supported by other objects. We move the object to the ground if it is not supported. This procedure repeats until all objects are optimized. The resulting arrangement conforms to the text description and is physically plausible.

## 4. Experimental Setup

We evaluate the SceneMotifCoder (SMC) framework using 3D assets from the Habitat Synthetic Scenes Dataset (HSSD-200) [17]. HSSD consists of human-authored 3D scenes that closely mirror real-life indoor scenes, using a set of high quality, diverse 3D object assets. The scenes contain a large variety of object arrangements which are ideal for learning real-world motifs.

To create a comprehensive evaluation of arrangement generation quality, we compile 181 everyday object arrangement descriptions belonging to 12 motif types: `stack`, `pile`, `row`, `grid`, `left_of`, `in_front`, `on_top`, `surround`, `rectangular_perimeter`, `wall_column`, `wall_row`, and `wall_grid`. We also include 21 descriptions for letter-shaped decorative arrangements resembling the letters `A`, `G`, `H`, `I`, `P`, `R`, and `S`. In total, our test set includes 202 distinct input text descriptions. These descriptions mention combinations of various everyday objects such as bowls, plates, cups, towels, books, and chairs. There are a total of 50 unique object categories across all descriptions. The everyday motifs are represented by between 5 and 27 descriptions, with a median of 16 descriptions per motif type, while the letter-shaped layouts each has 3 descriptions.

SMC learns a meta-program for each of the motif types given 1 to 3 example arrangements. These examples are extracted directly from the HSSD scenes. We manually annotate them with one text description each to serve as input for SMC's learning phase. At generation time, we retrieve 3D objects from HSSD (not limited to the objects that were used in the examples). We use GPT-4-Turbo (*gpt-4-turbo-2024-04-09*) [25] as the LLM for synthesizing visual programs. The average cost for LLM calls per learned motif meta-program is approximately $0.57.

### 4.1. Baselines

SMC generates tightly arranged spatial motifs of smaller objects, so we compare it with prior work that shares similar constraints. In particular, we compare with two text-to-3D generative models: **MVDream** [32] and **GraphDreamer** [9]. We choose MVDream as the representative approach for general text-to-3D generation as it excels at generating objects that are consistent across views, which is especially important in the multi-object setting. GraphDreamer is a recent work that claims to excel at generating multi-object arrangements, making it the closest to our task. It takes as input a scene graph generated by an LLM from text that describes the objects and their spatial relationships and optimizes an SDF for each object iteratively. See supplement for baseline implementation details.

### 4.2. Metrics

There is not yet a clear consensus on how to evaluate 3D generative models. GPTEval3D [43] showed that GPT-4V is capable of evaluating 3D shapes along various criteria. While GPTEval3D was not originally intended to evaluate layouts, we empirically observed that recent advances in LVLMs improved visual content perception and assessment of object placements. Therefore, we report the text-asset alignment (**Align**) and 3D plausibility (**Plaus**) metrics from GPTEval3D.We also report the combination of these metrics (**Overall**). The final score for SMC and the baselines are computed using 120 multi-view RGB images per output arrangement. For these metric computations with GPTEval3D, we use GPT-4o instead of the default GPT-4-Vision-Preview and GPT-4-Turbo used for result generation, as we found GPT-4o to perform better at visual perception tasks which form the core of the evaluation.

In addition, we conduct manual verification to evaluate the generation results on three axes: (1) correct number of objects (**# Objs**); (2) same layout as in the text description (**Layout**); and (3) the objects and their placements are physically plausible and appear the same as they would in real life (**Plaus**). Given pairs of input text prompt and generated arrangement (each represented by two rendered views), an annotator evaluated each method's results under the above criteria by giving a binary judgment per criterion.

Lastly, we carry out a perceptual evaluation study to report human preference of the generated object arrangements. The study is conducted using a two-alternative forced choice setup (A/B test). We randomly pick 20 text prompts and create 60 questions. Each question shows results from two of the three methods using the same input text description. Participants are asked to judge which result is better along two axes: (1) conformance to the text description — i.e. how well the arrangement respects the objects and relations described in the text (**Align**); and (2) realism — i.e. physical plausibility of the arrangement (**Realism**). We instruct participants to ignore texture and styling differences and focus solely on the above criteria. The study was done with 31 participants not involved with this work. We report the percentage of times each method was preferred for each description, compared against the other method. See the supplement for details.

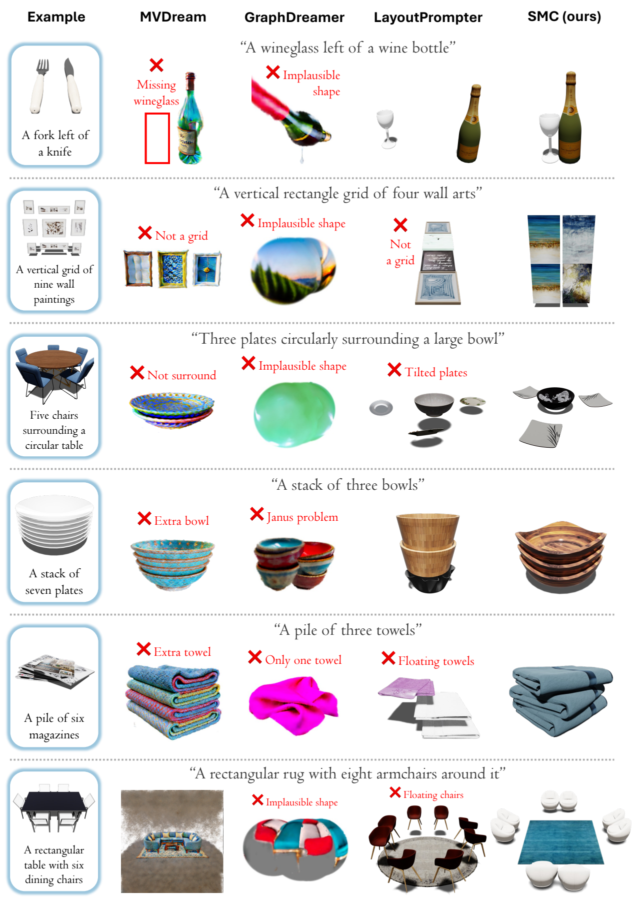


Figure 5. **Qualitative comparison of generated 3D object arrangements.** Each blue box on the left is the text and arrangement pair used for learning a meta-program capturing the arrangement's motif. The text above each row is the input at inference time. The arrangements generated by SceneMotifCoder better respect the input text in terms of number and type of objects present, and exhibit more plausible spatial configurations and object appearance.

## 5. Results

**Comparison with prior work.** Figure 5 shows arrangements generated using SMC and the baselines. Table 1 summarizes the quantitative evaluation. To verify GPTEval3D's ability to evaluate object layouts, we computed the correlation of its pairwise comparisons against manual verification and user study results and found agreement of 88.5% and 85.2%, demonstrating high alignment with human judgment. We also report the mean generation time.

The results show that SMC better conforms to the input text description in terms of the number of objects and the way they are arranged. MVDream sometimes omits an entire object category, and disregards conditions on the number of objects. It also sometimes generates extra objects (e.g., the floor in row 6). GraphDreamer performs worse than MVDream as it almost always disregards the specified object numbers and layout. Moreover, its outputs suffer from the Janus problem (e.g., generating bowls in irregular shapes). Objects are often blended together with indistinguishable boundaries in between. MVDream also suffers from the Janus problem but to a lesser extent. These limitations hinder the practical usefulness of the arrangements in populating digital scenes.

In contrast, using only 1 to 3 examples SMC excels at generating results that respect the input text description. Most generated arrangements have the correct number of objects, and the objects are arranged according to the description. Even in cases where the example SMC learned from is significantly different from the inference input description in terms of object categories, object count, and dimensions, SMC successfully generalizes the underlying motif. See the supplement for more qualitative examples.

**Ablations.** We compare against three ablated versions of SceneMotifCoder: 1) direct arrangement generation given descriptions of the DSL (w/o program); 2) without step-by-step approach of making high-level observations to rewrite the program with feedback (w/o observations); and 3) replace meta-program with in-context learning using example arrangements as reference directly (w/o meta-program).

We also report results from LayoutPrompter [20], a recent work on LLM-based 2D graphics layout generation. LayoutPrompter is similar to the ablated SMC without meta-program as it also uses in-context learning with a few examples. We adapt its CSS-based formulation, adding 3D object pose attributes to generate 3D layouts, and use mesh retrieval to turn the layouts into object arrangements.

Table 2 shows the quantitative results. Without the generalization capabilities of the program learning module, the system has a hard time generating arrangements with structural variations in the motif (e.g., `surround`), and generated programs are often incorrect (less than 40% of programs are correct, mostly easy motifs such as `left_of`). Without the step-by-step approach and feedback, generated meta-programs often fail to recognize patterns in the motif and exhibit naïve memorization, making them less adaptable to new input descriptions. Without meta-programs, the system is free to change program parameters arbitrarily, making it more likely to write programs with erroneous parameters. This is especially problematic when the target arrangement has more objects than the in-context examples, as there is no explicit constraint on where to put extra objects. Similarly, LayoutPrompter struggles at placing objects realistically in 3D. Floating objects and wrong rotations are the most common failure modes (see Fig. 5). Even with LLMs trained on large corpora of 2D layouts, it is not straightforward to transfer such knowledge to 3D.

**Applications.** Each meta-program powerfully captures general motifs. Figure 6 shows how meta-programs learned by SceneMotifCoder can be leveraged to efficiently and realistically populate indoor scenes. Figure 7 shows how

|  | Manual Verification |  |  | GPTEval3D [43] |  |  | Perceptual Study |  |  |
|---|---|---|---|---|---|---|---|---|---|
|  | # Objs ↑ | Layout ↑ | Plaus ↑ | Align ↑ | Plaus ↑ | Overall ↑ | Align ↑ | Realism ↑ | Time ↓ |
| GraphDreamer [9] | 0.18 | 0.24 | 0.11 | 858.87 | 844.81 | 866.67 | 0.21 | 0.13 | 95 min |
| MVDream [32] | 0.42 | 0.64 | 0.51 | 1083.57 | 1127.38 | 1089.83 | 0.44 | 0.54 | 43 min |
| SceneMotifCoder (ours) | **0.93** | **0.90** | **0.76** | **1162.28** | **1176.43** | **1153.67** | **0.85** | **0.83** | **2 min** |

Table 1. Evaluation with manual verification of generated arrangement quality by a human expert annotator, algorithmic evaluation with GPTEval3D [43], and a perceptual user study. We also report mean time to generate arrangement. Our approach outperforms the baselines across all metrics, particularly in alignment to the text description, as measured by the # Objs and Layout metrics.

|  | GPTEval3D |  |  |
|---|---|---|---|
|  | Align ↑ | Plaus ↑ | Overall ↑ |
| SceneMotifCoder (ours) | **1162.28** | **1176.43** | **1153.67** |
| — w/o meta-program | 1093.56 | 1119.31 | 1095.08 |
| — w/o observations | 1080.74 | 1094.83 | 1076.70 |
| — w/o program | 1036.18 | 1056.30 | 1035.42 |
| LayoutPrompter [20] | 1067.13 | 1066.16 | 1057.53 |

Table 2. We show the impact of key modules in SceneMotifCoder. LayoutPrompter and the SMC ablations generate arrangements with lower text-asset alignment, plausibility, and overall quality.

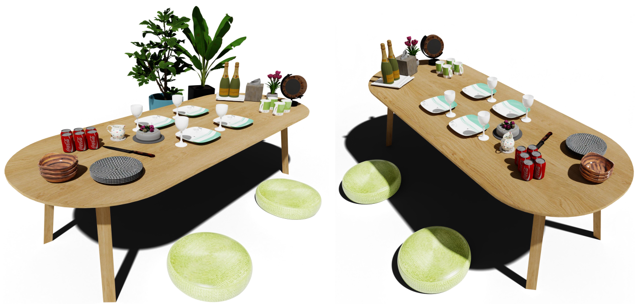
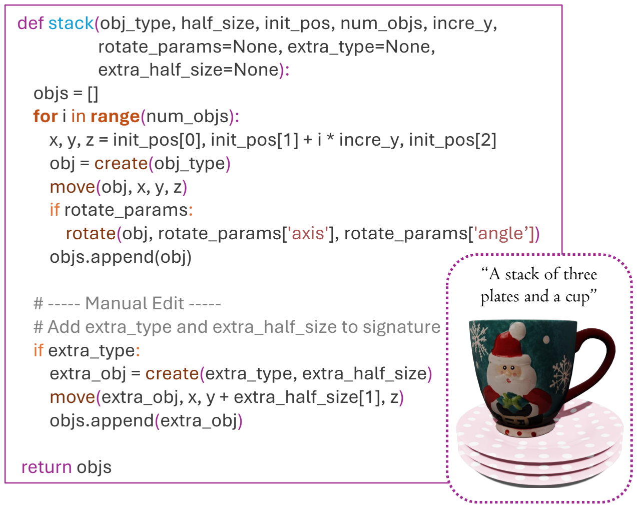


Figure 6. We populate the dining table scene shown with dense, realistic object arrangements by instantiating six learned motif types (`stack`, `row`, `grid`, `left_of`, `in_front`, and `on_top`).



```
def stack(obj_type, half_size, init_pos, num_objs, incre_y,
          rotate_params=None, extra_type=None,
          extra_half_size=None):
    objs = []
    for i in range(num_objs):
        x, y, z = init_pos[0], init_pos[1] + i * incre_y, init_pos[2]
        obj = create(obj_type)
        move(obj, x, y, z)
        if rotate_params:
            rotate(obj, rotate_params['axis'], rotate_params['angle'])
        objs.append(obj)

    # ----- Manual Edit -----
    # Add extra_type and extra_half_size to signature
    if extra_type:
        extra_obj = create(extra_type, extra_half_size)
        move(extra_obj, x, y + extra_half_size[1], z)
        objs.append(extra_obj)

    return objs
```

Figure 7. Editing a learned `stack` motif meta-program to add an object at the top of the stack (manually added code indicated in code comment). The edited meta-program preserves the generality of the original while adapting the motif with a user modification.

meta-programs can be edited and adapted. See supplement for more examples of meta-program generalization.

**Limitations.** We focused on simple motifs and relied on a library of curated 3D objects (i.e. pre-aligned, sized, and categorized). While the motifs we addressed are simple, our approach easily extends to more complex arrangements and using CLIP or text-to-3D generation can loosen the curated 3D objects requirement and enable more fine-grained retrieval. See the supplement for examples of more complex arrangements and using CLIP for retrieval. We can also use LVLMs to determine the size and front orientation of objects following prior work [45]. Another limitation is that we do not consider stylistic consistency between objects.

## 6. Conclusion

We presented SceneMotifCoder (SMC), an example-driven framework for learning visual programs to generate 3D object arrangements. While experts can code re-usable functions for specific arrangement, we showed that LLMs can automate this process and create reusable functions for common object arrangements. Using LLMs can also be substantially cheaper and less time consuming. We systematically evaluated our approach against state-of-the-art text-to-3D generation baselines, as well as ablations that show the value of our iterative visual program synthesis. Our results show that arrangements produced by SMC have higher physical plausibility and respect the input text description more closely. Using SMC, we enable users to easily generate and edit 3D object arrangements, and use them to populate indoor scenes. We believe SceneMotifCoder's ease of use and efficiency for generating 3D object arrangements provides a scalable and flexible solution for populating 3D scenes with realistic object arrangements.

**Acknowledgements.** This work was funded in part by a CIFAR AI Chair, a Canada Research Chair, NSERC Discovery Grant, and supported by Digital Research Alliance of Canada. We thank Qirui Wu, Jiayi Liu, and Han-Hung Lee for helpful discussions and feedback.

## References

[1] 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. Open-universe indoor scene generation using LLM program synthesis and uncurated object databases. *arXiv preprint arXiv:2403.09675*, 2024. 2

[2] Angel Chang, Manolis Savva, and Christopher D Manning. Learning spatial knowledge for text to 3D scene generation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 2028–2038, 2014. 1, 2

[3] Angel X Chang, Mihail Eric, Manolis Savva, and Christopher D Manning. SceneSeer: 3D scene design with natural language. *arXiv preprint arXiv:1703.00050*, 2017. 2

[4] Bob Coyne and Richard Sproat. WordsEye: An automatic text-to-scene conversion system. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 487–496, 2001. 2

[5] Weixi Feng, Wanrong Zhu, Tsu-jui Fu, Varun Jampani, Arjun Akula, Xuehai He, Sugato Basu, Xin Eric Wang, and William Yang Wang. LayoutGPT: Compositional visual planning and generation with large language models. *Advances in Neural Information Processing Systems*, 36, 2024. 1, 2

[6] Matthew Fisher, Daniel Ritchie, Manolis Savva, Thomas Funkhouser, and Pat Hanrahan. Example-based synthesis of 3D object arrangements. *ACM Transactions on Graphics (TOG)*, 31(6):1–11, 2012. 1, 2

[7] Aditya Ganeshan, R Kenny Jones, and Daniel Ritchie. Improving unsupervised visual program inference with code rewriting families. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 15791–15801, 2023. 3

[8] Yaroslav Ganin, Sergey Bartunov, Yujia Li, Ethan Keller, and Stefano Saliceti. Computer-aided design as language. *Advances in Neural Information Processing Systems*, 34:5885–5897, 2021. 3

[9] Gege Gao, Weiyang Liu, Anpei Chen, Andreas Geiger, and Bernhard Schölkopf. GraphDreamer: Compositional 3D scene synthesis from scene graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024. 2, 6, 8

[10] Lukas Höllein, Ang Cao, Andrew Owens, Justin Johnson, and Matthias Nießner. Text2Room: Extracting textured 3D meshes from 2D text-to-image models. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 7909–7920, 2023. 2

[11] Ziniu Hu, Ahmet Iscen, Aashi Jain, Thomas Kipf, Yisong Yue, David A Ross, Cordelia Schmid, and Alireza Fathi. SceneCraft: An LLM agent for synthesizing 3D scene as blender code. In *ICLR 2024 Workshop on LLM Agents*, 2024. 2

[12] Ian Huang, Guandao Yang, and Leonidas Guibas. BlenderAlchemy: Editing 3D graphics with vision-language models. *arXiv preprint arXiv:2404.17672*, 2024. 2

[13] Gabriel Ilharco, Mitchell Wortsman, Ross Wightman, Cade Gordon, Nicholas Carlini, Rohan Taori, Achal Dave, Vaishaal Shankar, Hongseok Namkoong, John Miller, Hannaneh Hajishirzi, Ali Farhadi, and Ludwig Schmidt. OpenCLIP, 2021. 5

[14] R Kenny Jones, Theresa Barton, Xianghao Xu, Kai Wang, Ellen Jiang, Paul Guerrero, Niloy J Mitra, and Daniel Ritchie. ShapeAssembly: Learning to generate programs for 3D shape structure synthesis. *ACM Transactions on Graphics (TOG)*, 39(6):1–20, 2020. 3

[15] R Kenny Jones, David Charatan, Paul Guerrero, Niloy J Mitra, and Daniel Ritchie. ShapeMOD: macro operation discovery for 3D shape programs. *ACM Transactions on Graphics (TOG)*, 40(4):1–16, 2021. 3

[16] R Kenny Jones, Paul Guerrero, Niloy J Mitra, and Daniel Ritchie. ShapeCoder: Discovering abstractions for visual programs from unstructured primitives. *ACM Transactions on Graphics (TOG)*, 2023. 3

[17] Mukul Khanna, Yongsen Mao, Hanxiao Jiang, Sanjay Haresh, Brennan Shacklett, Dhruv Batra, Alexander Clegg, Eric Undersander, Angel X. Chang, and Manolis Savva. Habitat Synthetic Scenes Dataset (HSSD-200): An analysis of 3D scene scale and realism tradeoffs for ObjectGoal navigation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024. 6

[18] Changjian Li, Hao Pan, Adrien Bousseau, and Niloy J Mitra. Free2CAD: Parsing freehand drawings into CAD commands. *ACM Transactions on Graphics (TOG)*, 41(4):1–16, 2022. 3

[19] Chenguo Lin and Yadong Mu. InstructScene: Instruction-driven 3D indoor scene synthesis with semantic graph prior. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024. 2

[20] Jiawei Lin, Jiaqi Guo, Shizhao Sun, Zijiang Yang, Jian-Guang Lou, and Dongmei Zhang. LayoutPrompter: Awaken the design ability of large language models. *Advances in Neural Information Processing Systems*, 36, 2024. 2, 7, 8

[21] Yiqi Lin, Haotian Bai, Sijia Li, Haonan Lu, Xiaodong Lin, Hui Xiong, and Lin Wang. CompoNeRF: Text-guided multi-object compositional NeRF with editable 3D scene layout. *arXiv preprint arXiv:2303.13843*, 2023. 2

[22] Yunchao Liu, Zheng Wu, Daniel Ritchie, William T Freeman, Joshua B Tenenbaum, and Jiajun Wu. Learning to describe scenes with programs. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2019. 3

[23] Paul Merrell, Eric Schkufza, Zeyang Li, Maneesh Agrawala, and Vladlen Koltun. Interactive furniture layout using interior design guidelines. *ACM transactions on graphics (TOG)*, 30(4):1–10, 2011. 1, 2

[24] Alex Nichol, Heewoo Jun, Prafulla Dhariwal, Pamela Mishkin, and Mark Chen. Point-E: A system for generating 3D point clouds from complex prompts. *arXiv preprint arXiv:2212.08751*, 2022. 2

[25] OpenAI. GPT-4 technical report. *ArXiv*, 2303, 2023. 6

[26] Wamiq Para, Shariq Bhat, Paul Guerrero, Tom Kelly, Niloy Mitra, Leonidas J Guibas, and Peter Wonka. SketchGen:

Generating constrained CAD sketches. *Advances in Neural Information Processing Systems*, 34:5077–5088, 2021. 3

[27] Wamiq Reyaz Para, Paul Guerrero, Niloy Mitra, and Peter Wonka. COFS: Controllable furniture layout synthesis. In *ACM SIGGRAPH Conference Proceedings*, 2023. 2

[28] Despoina Paschalidou, Amlan Kar, Maria Shugrina, Karsten Kreis, Andreas Geiger, and Sanja Fidler. ATISS: Autoregressive transformers for indoor scene synthesis. *Advances in Neural Information Processing Systems*, 34:12013–12026, 2021. 1, 2

[29] Ryan Po and Gordon Wetzstein. Compositional 3D scene generation using locally conditioned diffusion. *arXiv preprint arXiv:2303.12218*, 2023. 2

[30] Ben Poole, Ajay Jain, Jonathan T Barron, and Ben Mildenhall. DreamFusion: Text-to-3D using 2D diffusion. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2023. 2

[31] Ari Seff, Wenda Zhou, Nick Richardson, and Ryan P Adams. Vitruvion: A generative model of parametric CAD sketches. *arXiv preprint arXiv:2109.14124*, 2021. 3

[32] Yichun Shi, Peng Wang, Jianglong Ye, Mai Long, Kejie Li, and Xiao Yang. MVDream: Multi-view diffusion for 3D generation. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024. 6, 8

[33] Chunyi Sun, Junlin Han, Weijian Deng, Xinlong Wang, Zishan Qin, and Stephen Gould. 3D-GPT: Procedural 3D modeling with large language models. *arXiv preprint arXiv:2310.12945*, 2023. 2

[34] Jiapeng Tang, Yinyu Nie, Lev Markhasin, Angela Dai, Justus Thies, and Matthias Nießner. DiffuScene: Scene graph denoising diffusion probabilistic model for generative indoor scene synthesis. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024. 2

[35] Alexander Vilesov, Pradyumna Chari, and Achuta Kadambi. CG3D: Compositional generation for text-to-3D via gaussian splatting. *arXiv preprint arXiv:2311.17907*, 2023. 2

[36] Haochen Wang, Xiaodan Du, Jiahao Li, Raymond A Yeh, and Greg Shakhnarovich. Score Jacobian Chaining: Lifting pretrained 2D diffusion models for 3D generation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 12619–12629, 2023. 2

[37] Kai Wang, Manolis Savva, Angel X Chang, and Daniel Ritchie. Deep convolutional priors for indoor scene synthesis. *ACM Transactions on Graphics (TOG)*, 37(4):1–14, 2018. 2

[38] Kai Wang, Yu-An Lin, Ben Weissmann, Manolis Savva, Angel X Chang, and Daniel Ritchie. PlanIT: Planning and instantiating indoor scenes with relation graph and spatial prior networks. *ACM Transactions on Graphics (TOG)*, 38(4):1–15, 2019. 1, 2

[39] Sida I Wang, Samuel Ginn, Percy Liang, and Christoper D Manning. Naturalizing a programming language via interactive learning. In *Proceedings of the Conference of the Association for Computational Linguistics (ACL)*, 2017. 3

[40] Xinpeng Wang, Chandan Yeshwanth, and Matthias Nießner. SceneFormer: Indoor scene generation with transformers. In *Proceedings of the International Conference on 3D Vision (3DV)*, pages 106–115. IEEE, 2021. 2

[41] Zehao Wen, Zichen Liu, Srinath Sridhar, and Rao Fu. AnyHome: Open-vocabulary generation of structured and textured 3D homes. *arXiv:2312.06644*, 2023. 2

[42] Rundi Wu, Chang Xiao, and Changxi Zheng. DeepCAD: A deep generative network for computer-aided design models. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6772–6782, 2021. 3

[43] Tong Wu, Guandao Yang, Zhibing Li, Kai Zhang, Ziwei Liu, Leonidas Guibas, Dahua Lin, and Gordon Wetzstein. GPT-4V(ision) is a human-aligned evaluator for text-to-3D generation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024. 6, 8

[44] Xiang Xu, Karl DD Willis, Joseph G Lambourne, Chin-Yi Cheng, Pradeep Kumar Jayaraman, and Yasutaka Furukawa. SkexGen: Autoregressive generation of CAD construction sequences with disentangled codebooks. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2022. 3

[45] Yue Yang, Fan-Yun Sun, Luca Weihs, Eli VanderBilt, Alvaro Herrasti, Winson Han, Jiajun Wu, Nick Haber, Ranjay Krishna, Lingjie Liu, et al. Holodeck: Language guided generation of 3D embodied ai environments. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 20–25, 2024. 2, 8

[46] Taoran Yi, Jiemin Fang, Guanjun Wu, Lingxi Xie, Xiaopeng Zhang, Wenyu Liu, Qi Tian, and Xinggang Wang. GaussianDreamer: Fast generation from text to 3D gaussian splatting with point cloud priors. *arXiv preprint arXiv:2310.08529*, 2023. 2

[47] Lap Fai Yu, Sai Kit Yeung, Chi Keung Tang, Demetri Terzopoulos, Tony F Chan, and Stanley J Osher. Make it Home: Automatic optimization of furniture arrangement. *ACM SIGGRAPH Asia Conference Proceedings*, 30(4), 2011. 2

[48] Qihang Zhang, Chaoyang Wang, Aliaksandr Siarohin, Peiye Zhuang, Yinghao Xu, Ceyuan Yang, Dahua Lin, Bolei Zhou, Sergey Tulyakov, and Hsin-Ying Lee. SceneWiz3D: Towards text-guided 3D scene composition. *arXiv preprint arXiv:2312.08885*, 2023. 2

[49] Mengqi Zhou, Jun Hou, Chuanchen Luo, Yuxi Wang, Zhaoxiang Zhang, and Junran Peng. SceneX: Procedural controllable large-scale scene generation via large-language models. *arXiv preprint arXiv:2403.15698*, 2024. 2

[50] Shijie Zhou, Zhiwen Fan, Dejia Xu, Haoran Chang, Pradyumna Chari, Tejas Bharadwaj, Suya You, Zhangyang Wang, and Achuta Kadambi. DreamScene360: Unconstrained text-to-3D scene generation with panoramic gaussian splatting. *arXiv preprint arXiv:2404.06903*, 2024. 2

[51] Yang Zhou, Zachary While, and Evangelos Kalogerakis. SceneGraphNet: Neural message passing for 3D indoor scene augmentation. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 7384–7392, 2019. 2