

# The Scene Language: Representing Scenes with Programs, Words, and Embeddings

## – Supplementary Material –

Yunzhi Zhang<sup>1</sup>   Zizhang Li<sup>1</sup>   Matt Zhou<sup>2</sup>   Shangzhe Wu<sup>1</sup>   Jiajun Wu<sup>1</sup>

<sup>1</sup>Stanford University   <sup>2</sup>UC Berkeley

### A. Overview

This file contains representation details (Appendix B), experiment details (Appendix C), extended experiment results (Appendix D), discussions of limitations (Appendix E), and full language model prompts and outputs (Appendix F).

### B. Representation Details

#### B.1. Domain-Specific Language

The complete DSL is listed in Tab. 5. We explain the four macros introduced in Sec. 3.2, also listed in Tab. 5, as follows.

- Macro `call` retrieves `<entity-func>` bound to the input word, applies the function on the input embeddings, and outputs a data object of type `Entity`. Specifically, `(car embedding-list)` is the embedding of the output entity corresponding to `z` from Eq. (2), and `(cdr embedding-list)` is the embeddings of its descendent entities corresponding to `γ`.
- Macro `union` composes transformed entities by aggregating inputs into a list.
- Macro `union-loop` applies `union` in a for loop.
- Macro `transform` pairs an entity with its pose.

#### B.2. Details of Graphics Renderers

This section expands the instantiation of three graphics renderers from Sec. 4 in detail. For each renderer, we will discuss its parameter space  $\Theta$  and  $\mathcal{T}$ , rendering operation  $\mathcal{R}$ , and the reparameterization function  $g_{\text{reparam}}$ .

##### B.2.1. SDS-Based Renderer

**Parameter Space with 3D Gaussians.** For this renderer,  $\Theta$  is the space of 3D Gaussian parameters and  $\mathcal{T}$  is the space of 3D affine transformation matrices. In particular, each primitive is parameterized as a set of  $K$  3D Gaussians under a 3D affine transformation  $t$ , written as  $(\theta, t) = (K, \{\phi_i\}_{i=1}^K, t) \in \Theta \times \mathcal{T}$ , where  $\phi_i$  is the set of parameters for a single 3D Gaussian, and  $t$  is a 3D transformation matrix. Each Gaussian parameter  $\phi$  is defined as

$\phi := (\mu, \alpha, s, q, c)$ , denoting the 3D center position, opacity, scale, rotation in quaternion, and color of the Gaussian, respectively. An entity consisting of  $N$  primitives is parameterized as  $\{(\theta_j, t_j)\}_{j=1}^N = \{(K_j, \{\phi_i^j\}_{i=1}^{K_j}, t_j)\}_{j=1}^N$ .

**Differentiable Rendering.** The rendering operation  $\mathcal{R}$  for the 3D Gaussian renderer is as follows.

Following [18], a single Gaussian is defined by

$$G(x) = e^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)},$$

where  $x \in \mathbb{R}^3$  is a point in world coordinate,  $\Sigma := (RS)(RS)^T$  the 3D covariance matrix,  $R$  the rotation matrix computed from  $q$ , and  $S$  the scaling matrix computed from  $s$ .

A Gaussian under transformation  $t \in \mathcal{T}$  with  $t(x) = R_t S_t x + p_t$ , where  $R_t, S_t, p_t$  are the rotation, scaling, and translation components, respectively, is then computed with  $G_t$  satisfying the follows:

$$G_t(t(x)) = G(x).$$

We assume that diagonal entries of the scaling matrix  $S_t$  are all positive, and therefore  $t$  is invertible. Solving for the above equation gives

$$G_t(x) = e^{-\frac{1}{2}(x-\mu_t)^T \Sigma_t^{-1}(x-\mu_t)},$$

where  $\mu_t = t(\mu)$  and  $\Sigma_t = ((R_t R)(S_t S))((R_t R)(S_t S))^T$ . Let  $\tilde{t}(\phi)$  be the Gaussian after applying transformation  $t$  on  $\phi$ . Then  $\tilde{t}(\phi)$  has center  $\mu_t$ , rotation  $R_t R$ , scale  $S_t S$ , and has  $\alpha$  and  $c$  remaining unchanged as derived above.

The rendering operation  $\mathcal{R}$  to convert an entity consisting of  $N$  primitives,  $\{(\theta_j, t_j)\}_{j=1}^N = \{(K_j, \{\phi_i^j\}_{i=1}^{K_j}, t_j)\}_{j=1}^N$ , to the image space simply amounts to rendering all post-transformation 3D Gaussians in the scene,  $\{\tilde{t}_j(\theta_j)\}_j := \{\tilde{t}_j(\phi_i)\}_{i,j}$ , following the projection and blending process from [18].

**Primitive Reparameterization via SDS Guidance.** Recall that  $g_{\text{reparam}}$  aims to obtain 3D Gaussian primitive parameters for per-primitive conditional embeddings  $\{z_j\}_{j=1}^N$  and

Data Types	
<code>Word</code>	// <code>Word</code> specifying semantics
<code>Embedding</code>	// <code>Embedding</code> specifying an entity's attributes
<code>Matrix</code>	::= <code>Array[Array[Float]]</code> // Transformation in $GA(3, \mathbb{R})$
<code>Entity</code>	::= <code>Tuple[Tuple[Word, Embedding], List[Tuple[Entity, Matrix]]]</code>
<code>Vector</code>	::= <code>Array[Float]</code> // Vector in $\mathbb{R}^3$
Grammar	
<code>&lt;START&gt;</code>	::= <code>&lt;bind-expr&gt;*</code>
<code>&lt;bind-expr&gt;</code>	::= <code>(bind &lt;word&gt; &lt;entity-func&gt;)</code>
<code>&lt;entity-func&gt;</code>	::= <code>(lambda (embedding::Embedding embedding-list::List[Embedding])</code> <code>    &lt;sub-entities&gt;)</code>
<code>&lt;sub-entities&gt;</code>	::= <code>(union &lt;entity-transform&gt;*)</code> <code>      (union-loop &lt;loop-count&gt; (lambda (i::Integer) &lt;entity-transform&gt;))</code>
<code>&lt;entity-transform&gt;</code>	::= <code>(transform &lt;entity&gt; &lt;matrix&gt;)</code>
<code>&lt;entity&gt;</code>	::= <code>(call &lt;word&gt; &lt;embedding&gt;*)</code>
<code>&lt;word&gt;</code>	:: <code>Word</code>
<code>&lt;entity-func&gt;</code>	:: <code>Embedding -&gt; List[Embedding] -&gt; Entity</code>
<code>&lt;loop-count&gt;</code>	:: <code>Integer</code>
<code>&lt;matrix&gt;</code>	:: <code>Matrix</code>
<code>&lt;embedding&gt;</code>	:: <code>Embedding</code>
<code>&lt;4D-entity-func&gt;</code>	::= <code>(lambda () &lt;create-entity-list&gt;)</code> // Define a function that outputs a 4D scene
<code>&lt;create-entity-list&gt;</code>	::= <code>(list &lt;entity&gt;*)</code> // Represent a 4D scene as a temporal list of entities
Macros	
<code>call</code>	::= <code>(lambda (word . embedding-list) // Return an entity from the semantic class of word</code> <code>    (cons (cons word (car embedding-list))</code> <code>        ((retrieve word) (car embedding-list) (cdr embedding-list))))</code>
<code>union</code>	::= <code>list</code> // Compose transformed entities
<code>union-loop</code>	::= <code>(lambda (loop-count loop-func) // Compose transformed entities using a for loop</code> <code>    (union (map loop-func (iota loop-count))))</code>
<code>transform</code>	::= <code>cons</code> // Transform entity pose
<code>call</code>	:: <code>Word -&gt; List[Embedding] -&gt; Entity</code>
<code>union</code>	:: <code>(Tuple[Entity, Matrix])* -&gt; List[Tuple[Entity, Matrix]]</code>
<code>union-loop</code>	:: <code>Int -&gt; (Int -&gt; Tuple[Entity, Matrix])</code> <code>    -&gt; List[Tuple[Entity, Matrix]]</code>
<code>transform</code>	:: <code>Entity -&gt; Matrix -&gt; Tuple[Entity, Matrix]</code>
<code>translate</code>	:: <code>Vector -&gt; Matrix</code> // Compute translation matrix
<code>rotate</code>	:: <code>Float -&gt; Vector -&gt; Vector -&gt; Matrix</code> // Compute rotation matrix
<code>scale</code>	:: <code>Vector -&gt; Vector -&gt; Matrix</code> // Compute scaling matrix
<code>reflect</code>	:: <code>Vector -&gt; Vector -&gt; Matrix</code> // Compute reflection matrix
<code>@</code>	:: <code>Matrix -&gt; Matrix -&gt; Matrix</code> // Matrix multiplication
<code>compute-shape-center</code>	:: <code>Entity -&gt; Vector</code> // Compute center of an entity's bounding box
<code>compute-shape-min</code>	:: <code>Entity -&gt; Vector</code> // Compute minimum corner of an entity's bounding box
<code>compute-shape-max</code>	:: <code>Entity -&gt; Vector</code> // Compute maximum corner of an entity's bounding box
<code>compute-shape-sizes</code>	:: <code>Entity -&gt; Vector</code> // Compute sizes of an entity's bounding box
Special Forms	
<code>(bind &lt;word&gt; &lt;entity-func&gt;)</code>	// Defines and binds an entity function
<code>(retrieve &lt;word&gt;)</code>	// Retrieves an entity function bound to <code>word</code> , or <code>(lambda (.) (list))</code> if such function does not exist

Table 5. **The Domain-Specific Language.** The table contains the DSL specification used to define our representation. Built-in data types (e.g., `Float`), functions (e.g., `car` and `cdr`), special forms (`lambda`), and conditionals (`if`) are omitted; `<START>` denotes program starts, `::=` denotes rewriting rules; `::` denotes type annotation; `..::` denotes type annotation for an anonymous formal parameter, `*` denotes one or more expressions of the same type.

global condition  $z_{\text{global}}$ , where  $z_j = g_{\text{CLIP}}(c_j)$  is explained in Sec. 4, and  $z_{\text{global}} = g_{\text{CLIP}}(c_{\text{global}})$  is computed from a global scene description in texts,  $c_{\text{global}}$ . We now expand Sec. 4 to describe the optimization process of  $g_{\text{reparam}}$  in detail.

We write the SDS objective originally proposed in [25] as follows:

$$\nabla_{\psi} \mathcal{L}_{\text{SDS}}(x = \mathcal{R}(\psi); z, \hat{\epsilon}) \\ = \mathbb{E}_{\eta \sim \mathcal{U}(0,1), \epsilon \sim \mathcal{N}(0,I)} \left[ w(\eta) (\hat{\epsilon}(\alpha_{\eta} x + \alpha_{\eta} \epsilon, z, \eta) - \epsilon) \frac{\partial x}{\partial \psi} \right],$$

where  $\hat{\epsilon}$  is a pre-trained image denoising network,  $\eta$  is diffusion timestep, and  $w(\cdot)$ ,  $\alpha_{\eta}$  come from diffusion schedule.

For entity  $\{(\theta_j, t_j)\}_{j=1}^N$ , let

$$\begin{aligned} & \mathcal{L}(\{z_j\}_j, z_{\text{global}}, \{t_{\text{init},j}\}_j) \\ & := \mathcal{L}_{\text{SDS}}(\mathcal{R}(\{\tilde{t}_j(\theta_j)\}_j); z_{\text{global}}, \hat{\epsilon}_{\text{ControlNet}}) \\ & \quad + \sum_j \mathcal{L}_{\text{SDS}}(\mathcal{R}(\theta_j); z_j, \hat{\epsilon}_{\text{MVDream}}) \\ & \quad + \sum_j \mathcal{L}_{\text{reg}}(\theta_j, \text{StopGrad}(t_j)) + \sum_j \mathcal{L}_{\text{layout}}(\theta_j, t_{\text{init},j}), \end{aligned}$$

where  $\mathcal{L}_{\text{reg}}$ ,  $\mathcal{L}_{\text{layout}}$  are regularization terms following the definition from [44] and `StopGrad` stops gradients from backpropagation. Here,  $\mathcal{L}_{\text{reg}}$  penalizes Gaussian ellipsoids that are too long, and  $\mathcal{L}_{\text{layout}}$  penalizes Gaussians that lie

outside the initial bounding box specified by  $t_{\text{init}}$ .

Finally, we have

$$g_{\text{reparam}} = \arg \min_{\{(\theta_j, t_j)\}_{j=1}^N} \mathcal{L}.$$

During optimization, if primitives  $j_1$  and  $j_2$  have the same condition and initial normalized bounding box scale, *i.e.*,  $(z_{j_1} = z_{j_2}) \wedge (\frac{S_{t_{j_1}}}{\|S_{t_{j_1}}\|_2} = \frac{S_{t_{j_2}}}{\|S_{t_{j_2}}\|_2})$ , they are enforced to have the same parameters  $\theta$  (but still distinct  $t_{j_1}$  and  $t_{j_2}$ ), which greatly reduces the number of parameters in the solution space.

In practice, for certain scenes, LM outputs treat detailed object parts as primitives, *e.g.*, the hat rim and hat top from the first example in Fig. 11, and the backbone model for SDS guidance cannot effectively model such fine-grained parts. Therefore, we treat the hat as a primitive, whose pose is computed from the minimum bounding box containing both the hat rim and hat top, before carrying out the above optimization. This process effectively adapts the granularity of the computation graph, originally specified in LM inference outputs, to the graphics renderer being used, by assigning intermediate nodes from the original computation graph as the new leaf nodes.

### B.2.2. Mitsuba Renderer

**Parameter Space.** For this renderer,  $\Theta$  is the parameter space for three types of graphics primitives supported by Mitsuba: `cube`, `sphere`, and `cylinder`, as specified in the function header for `primitive_call` in Appendix F.1.1.  $\mathcal{T}$  is the 3D affine transformation space.

**Renderer.** We use the path tracer with maximum depth 8 implemented in Mitsuba. In this work, we use Mitsuba as a generic graphics engine and do not take advantage of its differentiability.

**Reparameterization.** Since we directly prompt LM to generate Mitsuba primitive parameters in its outputs as specified in Appendix F.1.1, the function parameters from raw LM outputs are already in the parameter space  $\Theta$  and are directly used for rendering, instead of being encoded into CLIP embeddings  $z \in \mathcal{Z}_{\text{CLIP}}$ .

### B.2.3. Minecraft Renderer

**Parameter Space.** For this renderer,  $\Theta$  is the asset parameters for Minecraft blocks, and  $\mathcal{T}$  is the space of 3D similarity transformation matrices, *i.e.*, of scaling and translation transformations. Note that we prevent rotation transformations in Minecraft, since that could lead to shapes that are impossible to render correctly in Minecraft.

Specifically,  $\Theta$  is specified in the docstring from Appendix F.1.4 and is expanded below. We introduce two types of primitives that let us construct in-game elements.

The first is `set_cuboid`. This primitive facilitates the creation of a cuboid within the Minecraft

environment. The function accepts three arguments: (1) A string denoting the Minecraft block type (*e.g.*, `"minecraft:white_concrete"`); (2) A tuple of three integers representing the scaling along the x, y, and z axes; (3) A boolean flag, `fill`, that specifies whether the cuboid should be solid or hollow. The cuboid is anchored at the coordinate origin  $(0, 0, 0)$ , which corresponds to its front-left-bottom vertex.

The second is `delete_blocks`. This primitive allows for the deletion of a previously placed cuboid. It accepts a single parameter, which is a tuple of three integers denoting the scaling along the x, y, and z axes. This operation removes the cuboid with its front-left-bottom vertex at the origin  $(0, 0, 0)$ , effectively clearing the designated space.

Note that we do not provide the Minecraft block type in the prompt, but instead let the model choose this parameter. Since there is a large amount of Minecraft data files on the web, the model performs decently well in choosing appropriate Minecraft blocks. We also augment this by building safety checks; for example, if the model chooses a Minecraft block that doesn't exist in our version of Minecraft, we will use semantic similarity to choose the most similar block from our library.

**Renderer.** We use WebGL<sup>1</sup> and Deepslate<sup>2</sup> for rendering Minecraft builds.

**Reparameterization.** Similar to Mitsuba, function parameters from LM-generated programs are directly used for rendering without CLIP encoding or reparameterization.

### B.2.4. Text-to-Image (T2I) Model Renderer

**Parameter Space.** We employ MIGC [42] as the backbone model for this renderer, which originally uses a CLIP text encoder [26] and a pre-trained UNet from Stable Diffusion [28] for layout-conditioned text-to-image generation. The parameter space  $\Theta$  for this renderer is the CLIP text embedding space.

**Renderer.** We first project the 3D bounding boxes of primitives from an execution output of our representation to a 2D layout under a specified camera viewpoint, and then run the forward pass of the T2I model conditioned on the 2D layout, where each 2D bounding box corresponds to an aforementioned CLIP embedding  $\theta \in \Theta$ .

**Reparameterization.** Function parameters from LM-generated programs are directly encoded by the CLIP text encoder using the language templates described in Sec. 5.

<sup>1</sup><https://get.webgl.org/>

<sup>2</sup><https://misode.github.io/deepslate/>

## C. Experiment Details

### C.1. Computation Cost

On one scene, LM inference takes <1 min, primitives-based rendering takes <1 min, SDS-based rendering takes ~30 min/object. All experiments run on 1 A5000 GPU with 48GB memory.

### C.2. Textual Inversion Optimization

To obtain image-conditioned embedding, we follow the procedure proposed in [6]. For the input image  $I$  and text prompt  $c_j$ , we first use  $c_j$  as guidance of GroundingSAM [27] to obtain the desired mask of the corresponding entity. The cropped region is pad to square and resized to desired resolution, resulting in image target  $I_j$ . The background of  $I_j$  is set to random grayscale color as used in [31].

We adopt the language template "<class>, 3d model, in the style of <style>" in all the textual inversion experiments. The template is first converted into token embeddings, then using CLIP text-encoder  $g_{\text{CLIP}}$  to transform to embeddings  $z_j$  for diffusion model  $\hat{\epsilon}_{\text{MVDream}}$ . In each textual-inversion iteration, we optimize the token embeddings  $v_{j1}, v_{j2}$  for <class> and <style> while freezing others. We use the similar objective as in diffusion model training:

$$v_{j1}^*, v_{j2}^* = \arg \min_{v_{j1}, v_{j2}} \mathbb{E}_{\eta, \epsilon} \|\epsilon - \hat{\epsilon}_{\text{MVDream}}(\alpha_{\eta} I_j + \alpha_{\eta} \epsilon, \eta, z_j(v_{j1}, v_{j2}))\|_2^2.$$

For each entity, we optimize the corresponding embeddings for 100 iterations with learning rate 1e-2. Empirically we find this setting is enough to fit the image conditions. After textual inversion, the embedding  $z_j$  is computed with optimized token embeddings, and used to guide the entity optimization as explained in Appendix B.2.

### C.3. GraphDreamer Implementation

Since the original paper didn't release the script for automatic scene graph generation, we follow the descriptions in the paper and re-implement this stage to query LM to output scene graphs in json format to avoid manually converting LM outputs to model configurations. The full system prompt is shown below:

```
You are helpful agent and can only write output
↳ wrapped in ```json```.
```

The full user prompt is shown below, where the given example input and output are taken from the teaser figure of the original paper [7]. In below, {task} is a placeholder for input text prompts of scenes.

```
Please follow the examples in the Visual Genome
↳ dataset and generate a scene graph in json
↳ format that best describes an input text. The
↳ output must contain four fields: "scene",
↳ "nodes", "edges", and "attributes".
```

- "scene" is the description of the input scene.
- "nodes" is a list of objects in the scene.
  - ↳ Maximum is three objects.
- "edges" is a cyclic list of relationships between
  - ↳ objects. Namely, each edge is a list of three
  - ↳ elements: [object1, relationship, object2],
  - ↳ where object1 and object2 are in the "nodes"
  - ↳ list. The number of edges must be no more than
  - ↳ number of possible pairs of objects in the
  - ↳ "nodes" list.
- "attributes" is a dictionary where each key is an
  - ↳ object in the "nodes" list and the value is a
  - ↳ list of its attributes.

Examl input:

```
A Wizard standing in front of a Wooden Desk, gazing
↳ into a Crystal Ball placed on the Wooden Desk,
↳ with a Stack of Ancient Spell Books sitting on
↳ the Wooden Desk and next to the crystal ball.
```

Example output:

```
```json
{
  "scene": "A Wizard standing in front of a
↳ Wooden Desk, gazing into a Crystal Ball
↳ placed on the Wooden Desk, with a Stack of
↳ Ancient Spell Books sitting on the Wooden
↳ Desk and next to the crystal ball.",
  "nodes": ["Wizard", "Wooden Desk", "Crystal
↳ Ball", "Stack of Ancient Spell Books"],
  "edges": [
    ["Wizard", "standing in front of", "Wooden
↳ Desk"],
    ["Crystal Ball", "placed on", "Wooden
↳ Desk"],
    ["Wizard", "gazing into", "Crystal Ball"],
    ["Stack of Ancient Spell Books", "sitting
↳ on", "Wooden Desk"],
    ["Stack of Ancient Spell Books", "next
↳ to", "Crystal Ball"]
  ],
  "attributes": {
    "Wizard": ["bearded", "robed",
↳ "wise-looking"],
    "Crystal Ball": ["small", "glowing",
↳ "transparent", "mystical"],
    "Stack of Ancient Spell Books":
↳ ["leather-bound", "weathered",
↳ "rune-etched"],
    "Wooden Desk": ["sturdy", "carved", "aged"]
  }
}
```
```

Your input:

```
```
{task}
```
```

Your output starts here. The output must be in a  
↳ valid json format.

After generating scene graphs with aforementioned prompt, we follow the released implementation from GraphDreamer to optimize for the final 3D representation, where each object in the node list is represented as an individual object SDF and color field. SDS loss is used to optimize each object and object pairs described in the edge list. When the raw scene graph output contains too many objects and exceeds the memory limitation required in the optimization, we rerun the graph generation step and add "The maximum number of objects is three" in the text prompt and rerun the optimization.



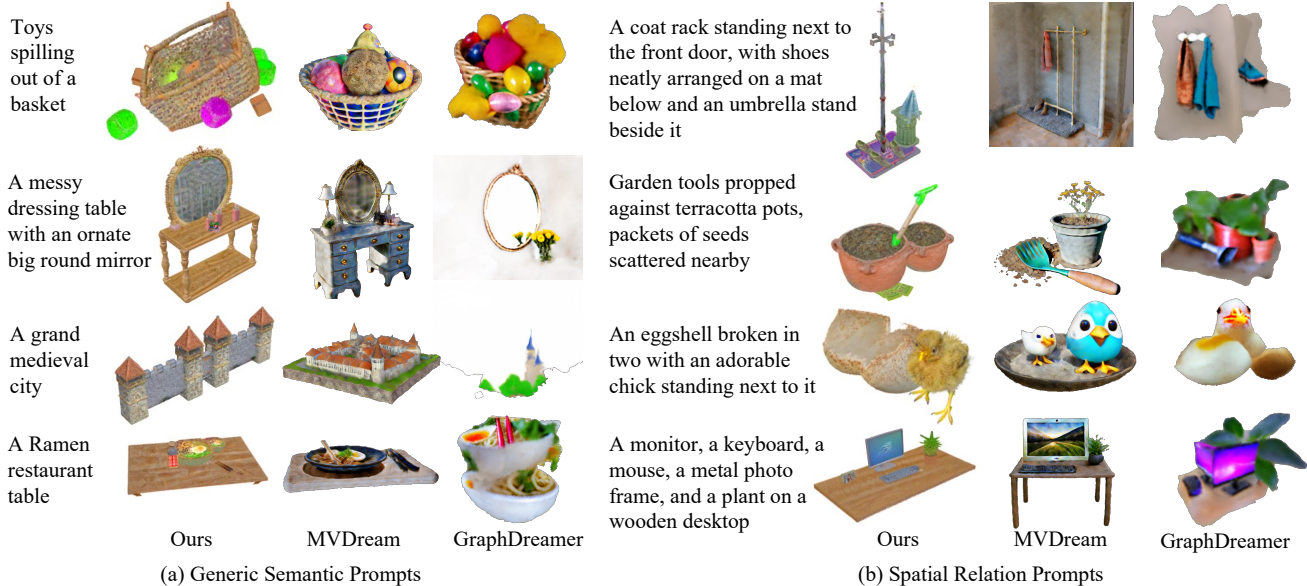


Figure 10. **Text-Prompted 3D Generation Comparisons** extending Fig. 4.

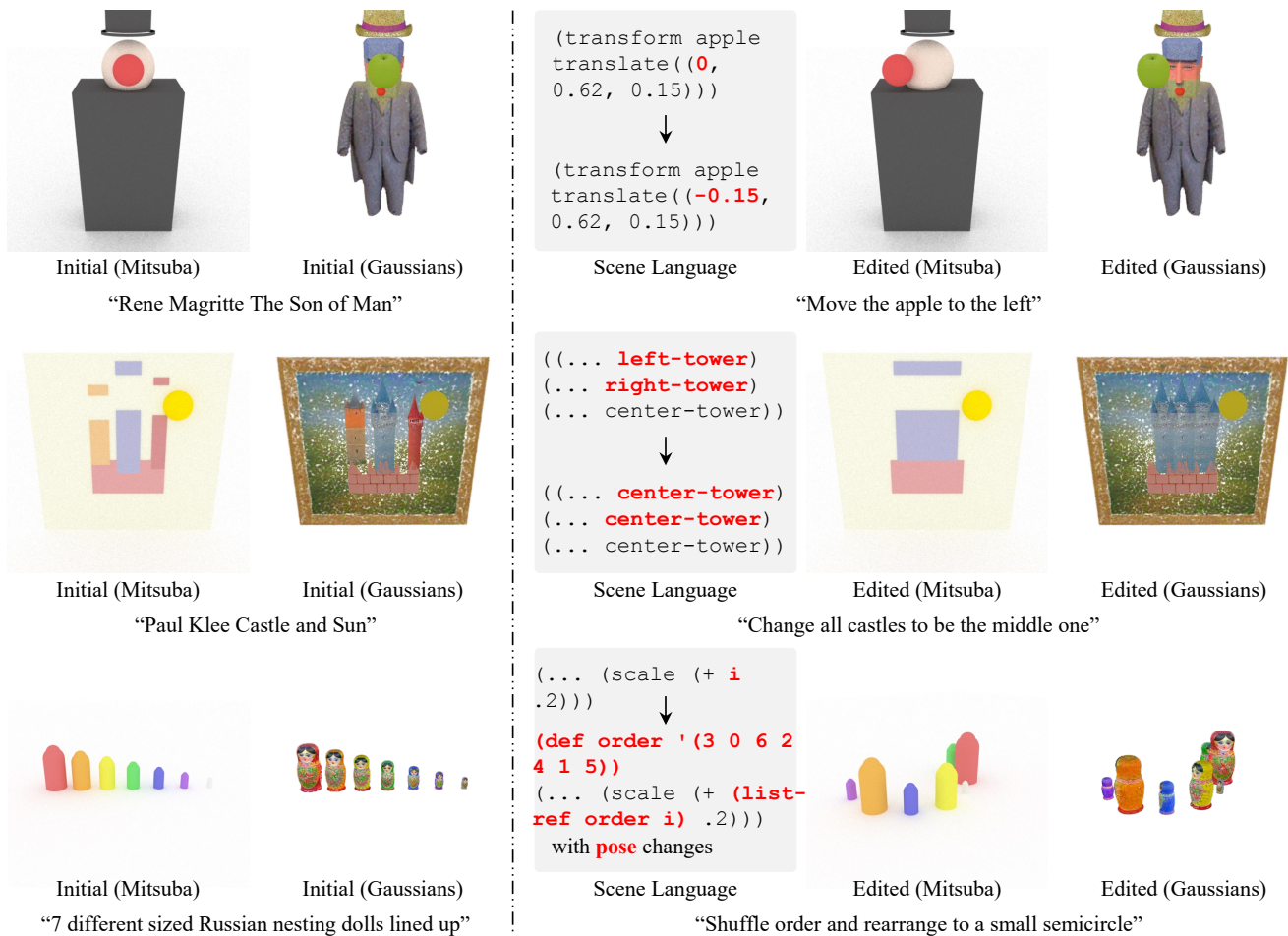


Figure 11. **Scene Editing with Language Instructions** extending Fig. 7.

| Methods                       | CLIP [13] ( $\uparrow$ ) | Dynamic [12] ( $\uparrow$ ) |
|-------------------------------|--------------------------|-----------------------------|
| 4D-fy [2]                     | 0.352                    | 0.2%                        |
| 4D-fy [2] (w/ prompt variant) | <b>0.354</b>             | 0.6%                        |
| Ours                          | 0.341                    | <b>5.9%</b>                 |

Table 6. **Evaluation for Text-Prompted 4D Generation** extending Tab. 4.

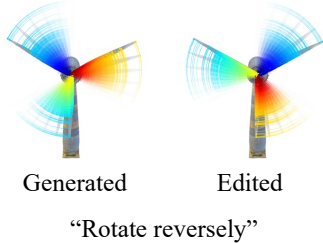


Figure 12. **4D Editing.**

## C.4. Evaluation Metrics

### C.4.1. User Study

The user study is conducted on Prolific<sup>3</sup> with 103 users. We provide the following instruction in the user study: “In the following questions, you’ll be provided a list of text descriptions and corresponding three generation results. Please choose the one that has best text-alignment, taking into account both the visual quality and the adherence to the layout structure outlined in the text prompt.” For each of the 9 scenes being evaluated, we ask the user to choose one of the three video rendering results, generated by our method and two baselines in randomized order. All videos shown in the user study are displayed on the project page.

### C.4.2. CLIP-Based Evaluation

We use OpenCLIP model for measuring image-text similarity (Tabs. 3 and 4). We use the model variant with the highest ImageNet zero-shot classification accuracy, OpenCLIP-ViT-H-14-378-quickgelu.

## D. Extended Experiment Results

### D.1. Text-Prompted 3D Scene Generation.

Qualitative examples for numeric scenes and generic scenes are shown on the website and in Fig. 10, respectively.

### D.2. Text-Prompted 4D Scene Generation

**Representation.** The 4D entity function mentioned in Sec. 6.1 is defined as  $\langle 4D\text{-entity-func} \rangle$  in the DSL defined in Tab. 5.

**Analysis.** Allowing for a flexible set of primitive entities is crucial to make our representation suitable for generating 4D scenes of different scales, including objects with moving

parts (e.g., the wind turbine from Fig. 6) and scenes with moving objects (e.g., the carousel). This is in contrast to prior works using primitives chosen from a fixed set [30, 36] or fixing the granularity to be object-centric [15].

Moreover, the hierarchical scene structure encapsulated by our program-based representation makes it possible to represent 4D scenes compactly, serving as a regularization for generation output. Multiple entities, such as outputs from the function `horse` from the carousel scene in Fig. 6, can be grouped into one function `horses` and thereby share the same temporal transformation. Writing composable functions for entity grouping effectively reduces the dimension of the temporal motion space and improves motion fidelity.

**Extended Quantitative Results.** On top of the evaluation in Tab. 4 with the same set of prompts for 4D-fy and ours, we evaluate 4D-fy with prompt variants that more explicitly indicate scene dynamics. Specifically, we use the following three prompts: “a solar system model with moving planets”, “a toy wind turbine with rotating blades”, and “carousel with a small canopy rotating”. Results are reported as 4D-fy (w/ prompt variants) in Tab. 6. We observe a small increment in CLIP similarity and dynamic degrees in 4D-fy results with the prompt variants, but still with a relatively small motion compared to ours.

### D.3. Text-Prompted 3D and 4D Scene Editing

Fig. 11 shows further qualitative experiment results under the same setup as Fig. 7. The same method applies to 4D editing, with results included in Fig. 12.

### D.4. Roles of Representation Components

Below we discuss the roles of representation components,  $P$  (programs),  $W$  (words), and  $Z$  (embeddings).

We ablate the roles of  $P$  and  $W$  on the 3D editing task as follows: (i) *No- $P$* , replacing the scene language with a leaf entity list (i.e., a list of word-pose tuples) before querying language models (LMs) to edit, (ii) *No- $W$* , replacing function names with random strings, and (iii) *No- $P$ -No- $W$* , removing words from (i).

We run these variants, together with baseline methods, to edit the top-left scene in Fig. 10. Input prompts and results are shown in Fig. 13(a). Removing  $P$  and/or  $W$  degrades the accuracy of the results, suggesting that both components contribute to an intuitive and accurate editing interface. On the other hand, the two baseline methods do not encode precise spatial information of scene components and require regenerating the scene (we use prompt “A basket upside-down with toys in the basket”), failing to preserve the original generated content.

Embeddings  $Z$  are necessary for image-prompted editing (Fig. 8). For generation tasks,  $W$  and  $Z$  are both required by neural renderers, and we ablate  $P$  by query-

<sup>3</sup><https://www.prolific.com/>

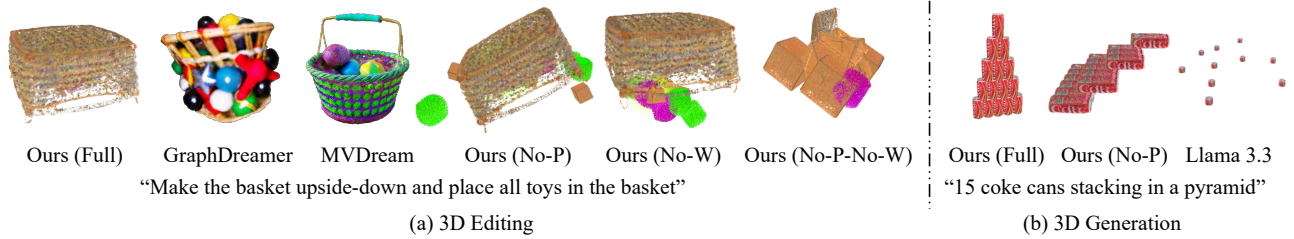


Figure 13. Ablations.

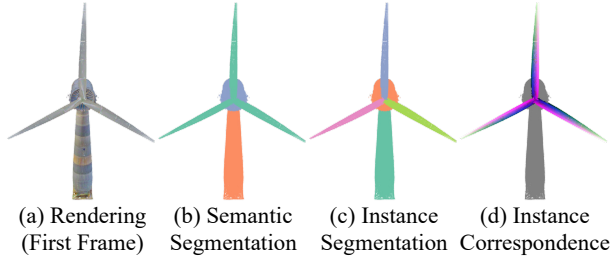


Figure 14. Visualizations of Discriminative Maps.

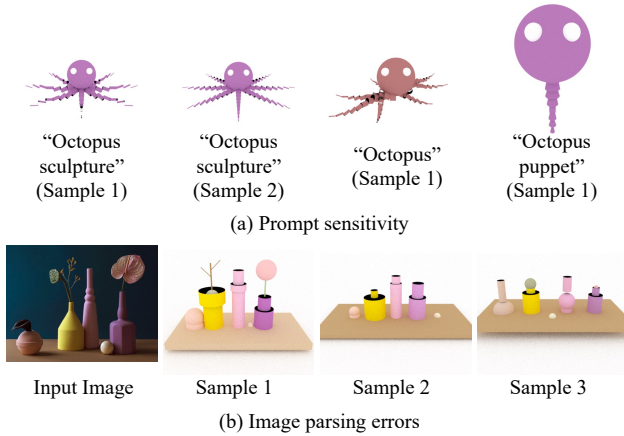


Figure 15. Failure Cases.

ing LMs to directly generate entity lists. As shown in Fig. 13(b), removing programs harms LM accuracy; alternative backbone Llama has inferior performance.

## D.5. Visualization of Discriminative Information

Several pieces of discriminative information can be directly obtained with the proposed Scene Language: semantic maps in Fig. 14(b), as words represent per-entity semantics; instance segmentation in Fig. 14(c), as the representation is compositional with separable instances; correspondence of the repeated instances in Fig. 14(d), as programs specify repetitions existing in a scene; dense temporal correspondence for 4D scenes, as shown in Fig. 6.

## E. Limitations

The current inference pipeline inherits limitations of the backbone pre-trained models.

**LM/VLM errors.** For text-conditioned tasks, minor variations in textual scene descriptions can lead to large quality differences in the output as shown in Fig. 15 (a). For image-conditioned tasks, input images are parsed with the backbone visual language model. In the example in Fig. 15 (b), with the same input image, parsing results have high variance across multiple inference runs.

**T2I Model Bias.** While the non-neural engines adopted are fully controllable and correct, neural-based engines may introduce additional errors, *e.g.*, the rendered umbrella is not fully folded (Fig. 10) due to the bias in T2I models. Furthermore, the controls over texture and shapes are not disentangled for neural engines, resulting in mixed texture and shape changes in Fig. 8 as they are both affected by embeddings.

While this work provides a viable inference method for the proposed representation that leverages the commonsense knowledge and code-writing capability of LMs and expressive renderings from T2I models, addressing the weaknesses inherited from these pre-trained models would further improve the robustness, complexity, and output quality for downstream tasks. We leave these as future directions to improve the inference of the Scene Language.

## F. Language Model Prompts and Responses

### F.1. Language Model Prompts

#### F.1.1. Text- and Image-Conditioned Scene Generation

In Sec. 5, we introduced an inference method for the representation by prompting LMs. The full system prompt is displayed below. The system prompt defines the data types and the function headers of macros from the DSL in Tab. 5, written in Python.

```
You are a code completion model and can only write
↪ python functions wrapped within ```python```.

You are provided with the following `helper.py`
↪ which defines the given functions and
↪ definitions:
```python
```

```

"""This module contains a Domain-Specific Language
↳ (DSL) designed
with built-in support for loops and functions for
↳ shape construction and transformation.
"""

```

```

from typing import NamedTuple, Any, Callable,
↳ Literal
import math
import numpy as np

```

```

# type aliases and DSL syntax sugar
P = Any # 3D vector, e.g., a point or direction
T = Any # 4x4 transformation matrix
Shape = list[dict[str, Any]] # a shape is a list
↳ of primitive shapes

```

```

# shape function library utils

```

```

def register(docstring: str):
    """
    Registers a function whose name must be unique.
    ↳ Provide keyword argument defaults for
    ↳ easier debugging.
    """

```

```

def library_call(func_name: str, **kwargs) ->
↳ Shape:
    """
    Call a function from the library and return its
    ↳ outputs. You are responsible for
    ↳ registering the function with 'register'.

```

```

    Args:
        func_name (str): Function name.
        **kwargs: Keyword arguments passed to the
        ↳ function.
    """

```

```

def primitive_call(name: Literal['cube', 'sphere',
↳ 'cylinder'], shape_kwargs: dict[str, Any],
↳ color: tuple[float, float, float] = (1.0, 1.0,
↳ 1.0)) -> Shape:
    """

```

```

    Constructs a primitive shape.

```

```

    Args:
        name: str - 'cube', 'sphere', or
        ↳ 'cylinder'.
        shape_kwargs: dict[str, Any] - keyword
        ↳ arguments for the primitive shape.
        - For 'cube': {'scale': P} - 3-tuple of
        ↳ floats for scaling along x, y, z
        ↳ axes.
        - For 'sphere': {'radius': float} -
        ↳ radius of the sphere.
        - For 'cylinder': {'radius': float,
        ↳ 'p0': P, 'p1': P}
        ↳ radius: float - radius of the
        ↳ cylinder.
        ↳ p0: P - 3-tuple of floats for the
        ↳ start point of the cylinder's
        ↳ centerline.
        ↳ p1: P - 3-tuple of floats for the
        ↳ end point of the cylinder's
        ↳ centerline.
        color: Tuple[float, float, float] - RGB
        ↳ color in range [0, 1]^3.

```

```

    Returns:
        Shape - the primitive shape.

```

```

    Examples:
    - `primitive_call('cube',
    ↳ shape_kwargs={'scale': (1, 2, 1)})`
    Returns a cube with corners (-0.5, -1,
    ↳ -0.5) and (0.5, 1, 0.5).
    - `primitive_call('sphere',
    ↳ shape_kwargs={'radius': 0.5})`
    Returns a sphere with radius 0.5, with
    ↳ bounding box corners (-0.5, -0.5,
    ↳ -0.5) and (0.5, 0.5, 0.5).
    - `primitive_call('cylinder',
    ↳ shape_kwargs={'radius': 0.5, 'height':
    ↳ 1})`

```

```

    Returns a cylinder with radius 0.5,
    ↳ height 1, with bounding box corners
    ↳ (-0.5, -0.5, -0.5) and (0.5, 0.5,
    ↳ 0.5).
    """

```

```

# control flows

```

```

def loop(n: int, fn: Callable[[int], Shape]) ->
↳ Shape:
    """
    Simple loop executing a function `n` times and
    ↳ concatenating the results.

```

```

    Args:
        n (int): Number of iterations.
        fn (Callable[[int], Shape]): Function that
        ↳ takes the current iteration index
        ↳ returns a shape.

```

```

    Returns:
        Concatenated shapes from each iteration.
    """

```

```

# shape manipulation

```

```

def concat_shapes(*shapes: Shape) -> Shape:
    """
    Combines multiple shapes into a single shape.
    """

```

```

def transform_shape(shape: Shape, pose: T) ->
↳ Shape:
    """
    Args:
        shape: Shape
        pose: T - If pose is A @ B, then B is
        ↳ applied first, followed by A.

```

```

    Returns:
        The input shape transformed by the given
        ↳ pose.
    """

```

```

# pose transformation

```

```

def rotation_matrix(angle: float, direction: P,
↳ point: P) -> T:
    """
    Args:
        angle (float) : the angle of rotation in
        ↳ radians
        direction (P) : the axis of rotation
        point (P) : the point about which the
        ↳ rotation is performed
    """

```

```

def translation_matrix(offset: P) -> T:
    """

```

```

    Args:
        offset (P) : the translation vector
    """

```

```

def scale_matrix(scale: float, origin: P) -> T:
    """
    Args:
        scale (float) - the scaling factor, only
        ↳ uniform scaling is supported
        origin (P) - the origin of the scaling
        ↳ operation
    """

```

```

def identity_matrix() -> T:
    """
    Returns the identity matrix in SE(3).
    """

```

```

# calculate locations and sizes of shape bounding
↳ boxes

```

```

def compute_shape_center(shape: Shape) -> P:
    """
    Returns the shape center.
    """

```

```

def compute_shape_min(shape: Shape) -> P:
    """
    Returns the min corner of the shape.
    """

```

```

def compute_shape_max(shape: Shape) -> P:
    """
    Returns the max corner of the shape.

```



```

"""
def compute_shape_sizes(shape: Shape) -> P:
    """
    Returns the shape sizes along x, y, and z axes.
    """
    ...

STRICTLY follow these rules:
1. Only use the functions and imported libraries in
   ↳ `helper.py`.
2. You can only write functions. Follow a modular
   ↳ approach and use the `register` decorator to
   ↳ define semantic shapes or shape groups.
3. Camera coordinate system: +x is right, +y is up,
   ↳ +z is backward.
4. You can use shape primitives to approximate
   ↳ shape components that are too complex. You must
   ↳ make sure shape have correct poses. Be careful
   ↳ about `set_mode` and `set_to` from
   ↳ `primitive_call`.
5. You must use `library_call` to call registered
   ↳ functions.
6. Use `compute_shape_*` from `helper.py` if
   ↳ possible to compute transformations.

You should be precise and creative.

```

The full user prompt for image or text-conditioned 3D generation is displayed below. It includes an example valid program, and the task specification indicated with a placeholder {task}. For text-conditioned generation (Sec. 6.1), it is replaced with the input textual scene description. For image-conditioned generation (Sec. 6.2), it is replaced with "Reconstruct the input scene", and the input image is also fed into LM.

```

Here are some examples of how to use `helper.py`:
```python
from helper import *

"""
A pile of books on a desk
"""

@register("book")
def book(scale: P) -> Shape:
    return primitive_call('cube', color=(.6, .3,
    ↳ .1), shape_kwargs={'scale': scale})

@register("books")
def books(width: float, length: float, book_height:
    ↳ float, num_books: int) -> Shape:
    def loop_fn(i) -> Shape:
        book_shape = library_call('book',
        ↳ scale=(width, book_height, length))
        book_shape = transform_shape(book_shape,
        ↳ translation_matrix([np.random.uniform(
        ↳ -0.05, 0.05), i * book_height,
        ↳ np.random.uniform(-0.05, 0.05)])) #
        ↳ FIRST translate
        book_center =
        ↳ compute_shape_center(book_shape) #
        ↳ must be computed AFTER transformation!!
        return transform_shape(book_shape,
        ↳ rotation_matrix(np.random.uniform(-0.1,
        ↳ 0.1), direction=(0, 1, 0),
        ↳ point=book_center)) # THEN tilt

    return loop(num_books, loop_fn)

@register("desk")
def desk(scale: P) -> Shape:
    return primitive_call('cube', color=(.4, .2,
    ↳ .1), shape_kwargs={'scale': scale})

@register('desk with books')

```

```

def desk_with_books() -> Shape:
    desk_shape = library_call('desk', scale=(1, .1,
    ↳ .5))
    books_shape = library_call('books', width=.21,
    ↳ length=.29, book_height=.05, num_books=3)
    ↳ desk_top, _ = compute_shape_max(desk_shape)
    ↳ books_bottom, _ =
    ↳ compute_shape_min(books_shape)
    return concat_shapes(
        desk_shape,
        transform_shape(books_shape,
        ↳ translation_matrix((0, desk_top -
        ↳ books_bottom, 0))) # stack books on
        ↳ top of desk
    )

...
IMPORTANT: THE FUNCTIONS ABOVE ARE JUST EXAMPLES,
↳ YOU CANNOT USE THEM IN YOUR PROGRAM!

Now, write a similar program for the given task:
```python
from helper import *

"""
{task}
"""
...

```

## F.1.2. Scene Editing

For text-prompted scene editing (Sec. 6.1), we prompt the LM in two rounds, first with a textual scene description with the same protocol from Sec. 6.1, and then with an editing instruction, e.g., "move the apple to the left". In the second round, the system prompt remains the same as Appendix F.1.1. The user prompt is as follows, where {program} is the LM output from first round, and {task} is the editing instruction.

```

Here is a program using `helper.py`:
```python
{program}

Now, do minimal edit to the program such that the
↳ scene function, when called, will follow the
↳ instruction: {task}.
Your code starts here.
```python
from helper import *

"""
{task}
"""

```

## F.1.3. 4D Generation

For 4D generation, we include one more macro definition in the system prompt as shown below, with the remaining system prompt the same as above:

```

```python
def register_animation(docstring: str | None =
    ↳ None):
    """
    Registers an animation function which is stored
    ↳ in the global `animation_func`. You can
    ↳ pass an optional docstring.

    If you register a function, there a couple of
    ↳ rules:
    - That function should never be called
    ↳ anywhere else in the program. This
    ↳ function gets used later by the
    ↳ rendering engine.
    - This function needs a return type of
    ↳ `Generator[Shape, None, None]`.
    """

```

```
...
```

The full user prompt for 4D generation is displayed below:

```
Here are some examples of how to use `helper.py`:
'''python
from helper import *

'''
three ghosts chasing a yellow pacman
'''

@register()
def pacman() -> Shape:
    return primitive_call('cube', color=(1, 1, 0),
        ↪ scale=.8)

@register()
def ghosts() -> Shape:
    return loop(3, lambda i: transform_shape(
        library_call('ghost', color=(i / 3, 1 - i /
        ↪ 3, 1 - i / 3)),
        translation_matrix([i, 0, 0])
    ))

@register()
def ghost(color) -> Shape:
    return primitive_call('sphere', color=color,
        ↪ scale=.8)

@register_animation()
def pacman_chase_animation() -> Generator[Shape,
    ↪ None, None]:
    # an animated scene
    total_frames = 4 # Number of frames in the
    ↪ animation

    for frame in range(total_frames):
        pacman_x = - frame / total_frames
        ghost_x_offset = - 2 * frame / total_frames

        # Move pacman and ghost
        pacman =
        ↪ transform_shape(library_call('pacman'),
        ↪ translation_matrix([pacman_x, 0, 0]))
        ghosts =
        ↪ transform_shape(library_call('ghosts'),
        ↪ translation_matrix([2 + ghost_x_offset,
        ↪ 0, 0]))

        # Export the shape, which is a frame in the
        ↪ animation
        yield concat_shapes(pacman, ghosts)

'''

IMPORTANT: THE FUNCTIONS ABOVE ARE JUST EXAMPLES,
    ↪ YOU CANNOT USE THEM IN YOUR PROGRAM!

Now, write a similar program for the given task:
'''python
from helper import *

'''
{task}
'''
'''
```

## F.1.4. Minecraft Rendering

We adapt the prompts from Appendix F.1.1 to prompt LM to generate Minecraft-compatible outputs. Note that programs generated in this section can still be rendered using Mitsuba renderer, as shown in Fig. 5, by converting each Minecraft block from LM outputs to a Mitsuba cube primitive with the corresponding location and with the color set as the average color of the Minecraft block.

Specifically, we remove `rotation_matrix` and `reflection_matrix` from the system prompt in Appendix F.1.1 and change the function header for `primitive_call` to the follows:

```
'''python
def primitive_call(name: Literal['set_cuboid',
    ↪ 'delete_blocks'], **kwargs) -> Shape:
    '''
    Args:
        name: str - the name of the primitive
        ↪ action
            support 'set_cuboid', 'delete_blocks'
        ...: Any - additional arguments for the
        ↪ primitive action
        For 'set_cuboid':
            - block_type: a string that denotes
            ↪ the block type, e.g. 'oak_log'.
            ↪ THESE MUST BE VALID LITEMATIC
            ↪ BLOCK TYPES.
            - block_kwargs: a dict[str, str] of
            ↪ additional properties to define
            ↪ a block's state fully, e.g. for
            ↪ 'oak_log', we need to define
            ↪ the axis with possible values
            ↪ 'x', 'y', or 'z'
            - scale: a list of 3 elements,
            ↪ denoting the scaling along the
            ↪ positive x, y, and z axes
            ↪ respectively. IMPORTANT: THESE
            ↪ CAN ONLY BE INTEGERS!
            - fill: a boolean, describing
            ↪ whether the cuboid should be
            ↪ filled, or be hollow. Hint:
            ↪ this can be useful for creating
            ↪ structures that should be
            ↪ hollow, such as a building.
        For 'delete_blocks':
            - scale: a list of 3 elements,
            ↪ denoting the scaling along the
            ↪ positive x, y, and z axes
            ↪ respectively. IMPORTANT: THESE
            ↪ CAN ONLY BE INTEGERS!

    Returns:
        Shape -
        For 'set_cuboid': a cuboid composed of
        ↪ Minecraft blocks, with the closest
        ↪ block at (0, 0, 0) and furthest
        ↪ (right, back-most) block at
        ↪ (scale[0], scale[1], scale[2]).
        For 'delete_blocks': an empty
        ↪ cuboid-shaped space without any
        ↪ blocks, starting from the closest
        ↪ block at (0, 0, 0) and furthest
        ↪ (right, back-most) block at
        ↪ (scale[0], scale[1], scale[2]).

    '''
'''
```

And we change the example program for user prompt accordingly to the follows:

```
'''python
from helper import *

'''
A red cube on the top left of a blue pyramid of
    ↪ height 4.
'''

@register()
def cube_set() -> Shape:
    return concat_shapes(
        library_call('red_cube'), # expects a cube
        ↪ with left-bottom-front corner block at
        ↪ (-2, 7, 2) and dims 2x2x2
        library_call('blue_pyramid'), # expects a
        ↪ blue pyramid of height 4
    ) # hint: these library calls must be
    ↪ implemented to be compatible with the usage
```

```

@register()
def red_cube() -> Shape:
    return transform_shape(
        primitive_call('set_cuboid',
            ↳ block_type='minecraft:redstone_block',
            ↳ scale=(2, 2, 2), fill=True),
        translation_matrix([-2, 7, 2]))

@register()
def blue_pyramid(n: int = 4) -> Shape:
    def create_pyramid_layer(i):
        # Logic here is that for the ith layer, it
        ↳ has dims (2*i + 1) x(2*i + 1).
        # We need to then shift that in the x
        ↳ dimension to center it, and then also
        ↳ in the y dimension to lift to the right
        ↳ layer of the pyramid.
        side_length = i * 2 + 1
        last_layer_length = n * 2 + 1
        x_z_offset = (last_layer_length -
            ↳ side_length) // 2
        y_offset = n - i - 1
        return transform_shape(
            primitive_call('set_cuboid',
                ↳ block_type='minecraft:lapis_block',
                ↳ scale=(side_length, 1,
                ↳ side_length),
                fill=True),
            translation_matrix([x_z_offset,
                ↳ y_offset, x_z_offset]))

    return loop(4, create_pyramid_layer)

"""
A forest of trees of varying heights.
"""

@register()
def forest(leaf_size: int = 3) -> Shape:
    # Double for loop for placing the trees
    tree_padding = leaf_size * 2 + 3 # This is how
    ↳ far the center point of each tree should be
    ↳ from each other
    return loop(4, lambda i: loop(4, lambda j:
        transform_shape(library_call('simple_tree',
            ↳ height=random.randint(3, 7)), # Make it
            ↳ random to give the appearance of having
            ↳ varying heights
            translation_matrix([i *
                ↳ leaf_size + tree_padding,
                ↳ 0, j * leaf_size +
                ↳ tree_padding]))))

@register()
def simple_tree(height: int = 4) -> Shape:
    return concat_shapes(
        library_call('trunk', trunk_height=height),
        transform_shape(library_call('leaves',
            ↳ leaf_size=3), # If you pass in extra
            ↳ arguments to library_call, they need to
            ↳ be NAMED arguments. Passing in 3 here
            ↳ without "leaf_size" will error.
            translation_matrix([-1,
                ↳ height, -1]) # Center
            ↳ the leaves on top of
            ↳ the trunk
            ))

@register()
def leaves(leaf_size: int = 3) -> Shape:
    return primitive_call('set_cuboid',
        ↳ block_type='minecraft:oak_leaves',
        ↳ block_kwargs={'distance': '7',
        ↳ 'persistent': "true", 'waterlogged':
        ↳ "false"}, scale=(leaf_size, leaf_size,
        ↳ leaf_size), fill=True)

@register()
def trunk(trunk_height: int = 4) -> Shape:
    return primitive_call('set_cuboid',
        ↳ block_type='minecraft:oak_log',
        ↳ block_kwargs={'axis': 'y'}, scale=(1,
        ↳ trunk_height, 1), fill=True)

```

## F.2. Language Model Outputs

### F.2.1. Text-Conditioned 3D Generation

**Example 1.** This example corresponds to Fig. 1.

Input: "Moai in Ahu Akivi, with slight variations".

Output:

Here's a program to create Moai statues in Ahu  
↳ Akivi with slight variations:

```

"""python
from helper import *

"""
Moai in Ahu Akivi, with slight variations
"""

@register()
def moai_head(scale: float) -> Shape:
    head = primitive_call('cube', color=(0.5, 0.5,
        ↳ 0.5), shape_kwargs={'scale': (scale,
        ↳ scale*1.5, scale*0.8)})
    nose = primitive_call('cube', color=(0.45,
        ↳ 0.45, 0.45), shape_kwargs={'scale':
        ↳ (scale*0.2, scale*0.4, scale*0.3)})
    nose = transform_shape(nose,
        ↳ translation_matrix((0, -scale*0.2,
        ↳ scale*0.4)))
    return concat_shapes(head, nose)

@register()
def moai_body(scale: float) -> Shape:
    body = primitive_call('cube', color=(0.55,
        ↳ 0.55, 0.55), shape_kwargs={'scale':
        ↳ (scale*0.8, scale*2, scale*0.6)})
    return body

@register()
def moai(scale: float) -> Shape:
    head = library_call('moai_head', scale=scale)
    body = library_call('moai_body', scale=scale)

    head_min = compute_shape_min(head)
    body_max = compute_shape_max(body)

    head = transform_shape(head,
        ↳ translation_matrix((0, body_max[1] -
        ↳ head_min[1], 0)))

    return concat_shapes(body, head)

@register()
def ahu_platform(width: float, height: float,
    ↳ depth: float) -> Shape:
    platform = primitive_call('cube', color=(0.3,
        ↳ 0.3, 0.3), shape_kwargs={'scale': (width,
        ↳ height, depth)})
    return platform

@register()
def moai_with_variation(base_scale: float) ->
    ↳ Shape:
    scale_variation = np.random.uniform(0.9, 1.1)
    rotation_variation = np.random.uniform(-0.05,
        ↳ 0.05)

    moai_statue = library_call('moai',
        ↳ scale=base_scale * scale_variation)
    moai_center = compute_shape_center(moai_statue)

    return transform_shape(moai_statue,
        ↳ rotation_matrix(rotation_variation, (0, 1,
        ↳ 0), moai_center))

@register()
def ahu_akivi_scene() -> Shape:
    platform = library_call('ahu_platform',
        ↳ width=14, height=1, depth=4)
    platform_top = compute_shape_max(platform)[1]

```

```

def create_moai(i: int) -> Shape:
    moai = library_call('moai_with_variation',
        ↳ base_scale=1.5)
    moai_bottom = compute_shape_min(moai) [1]
    x_position = (i - 3) * 2 # Distribute 7
        ↳ Moai evenly
    return transform_shape(moai,
        ↳ translation_matrix((x_position,
        ↳ platform_top - moai_bottom, 0)))

moai_statuses = loop(7, create_moai)

... return concat_shapes(platform, moai_statuses)

```

This program creates a scene of Moai statues in Ahu  
↳ Akivi with the following components:

1. `'moai_head'`: Creates a simplified Moai head  
↳ using cubes for the main head and nose.
2. `'moai_body'`: Creates a simplified Moai body  
↳ using a cube.
3. `'moai'`: Combines the head and body to form a  
↳ complete Moai statue.
4. `'ahu_platform'`: Creates the stone platform (ahu)  
↳ on which the Moai stand.
5. `'moai_with_variation'`: Applies slight variations  
↳ in scale and rotation to each Moai.
6. `'ahu_akivi_scene'`: Creates the entire scene with  
↳ a platform and seven Moai statues.

The scene features seven Moai statues standing on a  
↳ stone platform, similar to the real Ahu Akivi  
↳ site. Each Moai has slight variations in size  
↳ and rotation to add visual interest and realism  
↳ to the scene.

## Example 2. This example corresponds to Fig. 2.

Input: "a chessboard at game start".

Output:

Here's a program to create a chessboard at game  
↳ start using the provided `'helper.py'`:

```

'''python
from helper import *

'''

a chessboard at game start
'''

@register("chess_square")
def chess_square(color: tuple[float, float, float])
    ↳ -> Shape:
    return primitive_call('cube', color=color,
        ↳ shape_kwargs={'scale': (0.125, 0.02,
        ↳ 0.125)})

@register("chessboard")
def chessboard() -> Shape:
    def loop_fn(i) -> Shape:
        row = i // 8
        col = i % 8
        color = (0.9, 0.9, 0.9) if (row + col) % 2
        ↳ == 0 else (0.1, 0.1, 0.1)
        square = library_call('chess_square',
        ↳ color=color)
        return transform_shape(square,
        ↳ translation_matrix((col * 0.125, 0, row
        ↳ * 0.125)))

    return loop(64, loop_fn)

@register("chess_pawn")
def chess_pawn(color: tuple[float, float, float])
    ↳ -> Shape:
    base = primitive_call('cylinder', color=color,
        ↳ shape_kwargs={'radius': 0.05, 'p0': (0, 0,
        ↳ 0), 'p1': (0, 0.02, 0)})
    body = primitive_call('cylinder', color=color,
        ↳ shape_kwargs={'radius': 0.04, 'p0': (0,
        ↳ 0.02, 0), 'p1': (0, 0.08, 0)})

```

```

head = primitive_call('sphere', color=color,
    ↳ shape_kwargs={'radius': 0.035})
head = transform_shape(head,
    ↳ translation_matrix((0, 0.115, 0)))
return concat_shapes(base, body, head)

@register("chess_rook")
def chess_rook(color: tuple[float, float, float])
    ↳ -> Shape:
    base = primitive_call('cylinder', color=color,
        ↳ shape_kwargs={'radius': 0.05, 'p0': (0, 0,
        ↳ 0), 'p1': (0, 0.02, 0)})
    body = primitive_call('cylinder', color=color,
        ↳ shape_kwargs={'radius': 0.04, 'p0': (0,
        ↳ 0.02, 0), 'p1': (0, 0.1, 0)})
    top = primitive_call('cube', color=color,
        ↳ shape_kwargs={'scale': (0.09, 0.03, 0.09)})
    top = transform_shape(top,
        ↳ translation_matrix((0, 0.115, 0)))
    return concat_shapes(base, body, top)

@register("chess_knight")
def chess_knight(color: tuple[float, float, float])
    ↳ -> Shape:
    base = primitive_call('cylinder', color=color,
        ↳ shape_kwargs={'radius': 0.05, 'p0': (0, 0,
        ↳ 0), 'p1': (0, 0.02, 0)})
    body = primitive_call('cylinder', color=color,
        ↳ shape_kwargs={'radius': 0.04, 'p0': (0,
        ↳ 0.02, 0), 'p1': (0, 0.08, 0)})
    head = primitive_call('cube', color=color,
        ↳ shape_kwargs={'scale': (0.06, 0.08, 0.04)})
    head = transform_shape(head,
        ↳ translation_matrix((0.01, 0.11, 0)))
    head = transform_shape(head,
        ↳ rotation_matrix(0.3, (0, 0, 1), (0, 0.11,
        ↳ 0)))
    return concat_shapes(base, body, head)

@register("chess_bishop")
def chess_bishop(color: tuple[float, float, float])
    ↳ -> Shape:
    base = primitive_call('cylinder', color=color,
        ↳ shape_kwargs={'radius': 0.05, 'p0': (0, 0,
        ↳ 0), 'p1': (0, 0.02, 0)})
    body = primitive_call('cylinder', color=color,
        ↳ shape_kwargs={'radius': 0.04, 'p0': (0,
        ↳ 0.02, 0), 'p1': (0, 0.1, 0)})
    head = primitive_call('sphere', color=color,
        ↳ shape_kwargs={'radius': 0.03})
    head = transform_shape(head,
        ↳ translation_matrix((0, 0.13, 0)))
    top = primitive_call('cylinder', color=color,
        ↳ shape_kwargs={'radius': 0.01, 'p0': (0,
        ↳ 0.13, 0), 'p1': (0, 0.16, 0)})
    return concat_shapes(base, body, head, top)

@register("chess_queen")
def chess_queen(color: tuple[float, float, float])
    ↳ -> Shape:
    base = primitive_call('cylinder', color=color,
        ↳ shape_kwargs={'radius': 0.055, 'p0': (0, 0,
        ↳ 0), 'p1': (0, 0.02, 0)})
    body = primitive_call('cylinder', color=color,
        ↳ shape_kwargs={'radius': 0.045, 'p0': (0,
        ↳ 0.02, 0), 'p1': (0, 0.12, 0)})
    crown = primitive_call('sphere', color=color,
        ↳ shape_kwargs={'radius': 0.04})
    crown = transform_shape(crown,
        ↳ translation_matrix((0, 0.16, 0)))
    top = primitive_call('cylinder', color=color,
        ↳ shape_kwargs={'radius': 0.01, 'p0': (0,
        ↳ 0.16, 0), 'p1': (0, 0.2, 0)})
    return concat_shapes(base, body, crown, top)

@register("chess_king")
def chess_king(color: tuple[float, float, float])
    ↳ -> Shape:
    base = primitive_call('cylinder', color=color,
        ↳ shape_kwargs={'radius': 0.06, 'p0': (0, 0,
        ↳ 0), 'p1': (0, 0.02, 0)})
    body = primitive_call('cylinder', color=color,
        ↳ shape_kwargs={'radius': 0.05, 'p0': (0,
        ↳ 0.02, 0), 'p1': (0, 0.14, 0)})
    crown = primitive_call('cube', color=color,
        ↳ shape_kwargs={'scale': (0.1, 0.03, 0.1)})

```

```

crown = transform_shape(crown,
↳ translation_matrix((0, 0.17, 0)))
cross_v = primitive_call('cube', color=color,
↳ shape_kwargs={'scale': (0.02, 0.06, 0.02)})
cross_v = transform_shape(cross_v,
↳ translation_matrix((0, 0.215, 0)))
cross_h = primitive_call('cube', color=color,
↳ shape_kwargs={'scale': (0.06, 0.02, 0.02)})
cross_h = transform_shape(cross_h,
↳ translation_matrix((0, 0.215, 0)))
return concat_shapes(base, body, crown,
↳ cross_v, cross_h)

@register("chess_pieces")
def chess_pieces() -> Shape:
white = (0.9, 0.9, 0.9)
black = (0.1, 0.1, 0.1)
pieces = []

# Place pawns
for i in range(8):
white_pawn = library_call('chess_pawn',
↳ color=white)
black_pawn = library_call('chess_pawn',
↳ color=black)
pieces.append(transform_shape(white_pawn,
↳ translation_matrix((i * 0.125, 0.02,
↳ 0.125))))
pieces.append(transform_shape(black_pawn,
↳ translation_matrix((i * 0.125, 0.02,
↳ 0.75))))

# Place other pieces
piece_order = ['rook', 'knight', 'bishop',
↳ 'queen', 'king', 'bishop', 'knight',
↳ 'rook']
for i, piece in enumerate(piece_order):
white_piece =
↳ library_call(f'chess_{piece}',
↳ color=white)
black_piece =
↳ library_call(f'chess_{piece}',
↳ color=black)
pieces.append(transform_shape(white_piece,
↳ translation_matrix((i * 0.125, 0.02,
↳ 0))))
pieces.append(transform_shape(black_piece,
↳ translation_matrix((i * 0.125, 0.02,
↳ 0.875))))

return concat_shapes(*pieces)

@register("chessboard_at_game_start")
def chessboard_at_game_start() -> Shape:
board = library_call('chessboard')
pieces = library_call('chess_pieces')
return concat_shapes(board, pieces)

```

This program creates a chessboard at game start  
↳ with the following components:

1. A chessboard made of alternating white and black squares.
2. Chess pieces for both white and black sides, including pawns, rooks, knights, bishops, queens, and kings.
3. The pieces are placed in their correct starting positions on the board.

To use this program and generate the chessboard at  
↳ game start, you would call:

```

```python
chessboard_scene =
↳ library_call('chessboard_at_game_start')
```

```

This will return a complete chessboard with all  
↳ pieces in their starting positions.

tions. LM outputs in the first round are omitted. Rendering outputs are shown in Fig. 7 and Fig. 11.

**Example 1.** Input scene description: "Bramante Staircase, Vatican Museums". Input editing instruction: "shrink staircase radius by 80%".

Output:

Here's the minimally edited program that shrinks  
↳ the staircase radius by 80%:

```

```python
from helper import *

"""
Shrink staircase radius by 80%
"""

@register()
def stair_step(width: float, height: float, depth:
↳ float) -> Shape:
return primitive_call('cube', color=(0.8, 0.8,
↳ 0.8), shape_kwargs={'scale': (width,
↳ height, depth)})

@register()
def spiral_staircase(radius: float, height: float,
↳ num_steps: int) -> Shape:
def loop_fn(i: int) -> Shape:
angle = (i / num_steps) * 2 * math.pi
step_width = 0.8 * radius
step_height = height / num_steps
step_depth = 0.3 * radius

step = library_call('stair_step',
↳ width=step_width, height=step_height,
↳ depth=step_depth)

y_offset = i * step_height
x_offset = radius * math.cos(angle)
z_offset = radius * math.sin(angle)

step = transform_shape(step,
↳ translation_matrix((x_offset, y_offset,
↳ z_offset)))
return transform_shape(step,
↳ rotation_matrix(angle, (0, 1, 0),
↳ (x_offset, y_offset, z_offset)))

return loop(num_steps, loop_fn)

@register()
def central_column(radius: float, height: float) ->
↳ Shape:
return primitive_call('cylinder', color=(0.7,
↳ 0.7, 0.7),
shape_kwargs={'radius':
↳ radius, 'p0': (0, 0,
↳ 0), 'p1': (0, height,
↳ 0)})

@register()
def bramante_staircase() -> Shape:
staircase_radius = 5.0 * 0.2 # Shrink
↳ staircase radius by 80%
staircase_height = 15.0
num_steps = 60
column_radius = 0.5 * 0.2 # Shrink column
↳ radius proportionally

stairs = library_call('spiral_staircase',
↳ radius=staircase_radius,
↳ height=staircase_height,
↳ num_steps=num_steps)
column = library_call('central_column',
↳ radius=column_radius,
↳ height=staircase_height)

return concat_shapes(stairs, column)

```

In this edited version, I've made the following  
↳ changes to shrink the staircase radius by 80%:

## F.2.2. Text-Conditioned 3D Editing

In this task, for each example, we prompt LM in two rounds—the first round with a scene description as task specification, and the second round with editing instruc-



1. In the `bramante_staircase` function, I've modified the `staircase_radius` by multiplying it by 0.2 (which is equivalent to reducing it by 80%):  

```
python
staircase_radius = 5.0 * 0.2 # Shrink staircase
radius by 80%
```
2. To maintain proportions, I've also adjusted the `column_radius` by the same factor:  

```
python
column_radius = 0.5 * 0.2 # Shrink column
radius proportionally
```

These changes will result in a Bramante staircase that has a radius 80% smaller than the original, while maintaining the same height and number of steps. The central column has also been proportionally reduced to maintain the overall look of the structure.

**Example 2.** Input scene description: "a 8-layer 3-color Jenga set at game start". Input editing instruction: "remove 2 blocks from second top layer and the tower should not fall".

Output:

```
Here's the minimally edited program to remove 2
blocks from the second top layer of the Jenga
tower:

python
from helper import *

"""
Remove 2 blocks from second top layer and the tower
should not fall
"""

@register()
def jenga_block(color: tuple[float, float, float])
-> Shape:
    """Create a single Jenga block"""
    return primitive_call('cube', color=color,
        shape_kwargs={'scale': (0.75, 0.5, 2.25)})

@register()
def jenga_layer(is_horizontal: bool, y_offset:
float, colors: list[tuple[float, float, float]],
remove_blocks: list[int] = []) ->
Shape:
    """Create a layer of 3 Jenga blocks, optionally
removing specified blocks"""
    def block_fn(i: int) -> Shape:
        if i in remove_blocks:
            return []
        block = library_call('jenga_block',
            color=colors[i])
        if is_horizontal:
            transform = translation_matrix((i *
                0.75 - 0.75, y_offset, 0))
        else:
            transform = translation_matrix((0,
                y_offset, i * 0.75 - 0.75))
            transform = transform @
            rotation_matrix(math.pi/2, (0, 1,
                0), (0, 0, 0))
        return transform_shape(block, transform)

    return loop(3, block_fn)

@register()
def jenga_tower() -> Shape:
    """Create an 8-layer Jenga tower with 2 blocks
removed from the second top layer"""
    colors = [(0.8, 0.4, 0.3), (0.9, 0.7, 0.5),
        (0.6, 0.3, 0.1)]
```

```
def layer_fn(i: int) -> Shape:
    is_horizontal = i % 2 == 0
    y_offset = i * 0.5
    layer_colors = [colors[i % 3], colors[(i+1)
        % 3], colors[(i+2) % 3]]
    remove_blocks = [0, 2] if i == 6 else [] #
    Remove blocks 0 and 2 from the second
    top layer (index 6)
    return library_call('jenga_layer',
        is_horizontal=is_horizontal,
        y_offset=y_offset, colors=layer_colors,
        remove_blocks=remove_blocks)

return loop(8, layer_fn)

@register()
def jenga_set() -> Shape:
    """Create a complete Jenga set with 2 blocks
removed from the second top layer"""
    tower = library_call('jenga_tower')
    base = primitive_call('cube', color=(0.4, 0.2,
        0.1), shape_kwargs={'scale': (3, 0.1, 3)})

    tower_min = compute_shape_min(tower)
    base_max = compute_shape_max(base)

    return concat_shapes(
        base,
        transform_shape(tower,
            translation_matrix((0, base_max[1] -
                tower_min[1], 0)))
    )
```

This modified version of the program removes 2 blocks from the second top layer of the Jenga tower. Here are the changes made:

1. The `jenga_layer` function now accepts an optional `remove_blocks` parameter, which is a list of block indices to remove from the layer.
2. In the `jenga_tower` function, we modify the `layer_fn` to remove blocks 0 and 2 (the leftmost and rightmost blocks) from the second top layer (index 6, since we're counting from 0).
3. The rest of the program remains the same, ensuring that the tower is still placed on the base and the overall structure is maintained.

These changes will result in a Jenga tower with two blocks missing from the second top layer, while the rest of the tower remains intact. The tower should not fall because the top layer is still fully supported by the remaining block in the second top layer.

**Example 3.** Input scene description: "a fractal tree-like structure". Input editing instruction: "make branching structure to be trinary and 3D".

Output:

```
Here's a program that creates a simple but
interesting shape using function recursion,
without being a pyramid:

python
from helper import *

"""
A fractal tree-like structure
"""

@register()
def branch(length: float, radius: float, depth:
int) -> Shape:
    if depth == 0:
```

```

    return primitive_call('cylinder',
        ↳ shape_kwargs={'radius': radius, 'p0': (0,
        ↳ (0, 0, 0), 'p1': (0, length, 0)}, color=(0.6,
        ↳ color=(0.6, 0.4, 0.2))

main_branch = primitive_call('cylinder',
    ↳ shape_kwargs={'radius': radius, 'p0': (0,
    ↳ (0, 0), 'p1': (0, length, 0)}, color=(0.6,
    ↳ 0.4, 0.2))

sub_branch_length = length * 0.7
sub_branch_radius = radius * 0.6

left_branch = library_call('branch',
    ↳ length=sub_branch_length,
    ↳ radius=sub_branch_radius, depth=depth-1)
right_branch = library_call('branch',
    ↳ length=sub_branch_length,
    ↳ radius=sub_branch_radius, depth=depth-1)

angle = 0.5 # ~28.6 degrees

left_transform = translation_matrix((0, length,
    ↳ 0)) @ rotation_matrix(angle, (0, 0, 1), (0,
    ↳ 0, 0))
right_transform = translation_matrix((0,
    ↳ length, 0)) @ rotation_matrix(-angle, (0,
    ↳ 0, 1), (0, 0, 0))

return concat_shapes(
    main_branch,
    transform_shape(left_branch,
        ↳ left_transform),
    transform_shape(right_branch,
        ↳ right_transform)
)

@register()
def fractal_tree(base_length: float = 1.0,
    ↳ base_radius: float = 0.1, depth: int = 5) ->
    ↳ Shape:
    tree = library_call('branch',
        ↳ length=base_length, radius=base_radius,
        ↳ depth=depth)
    tree_min = compute_shape_min(tree)
    return transform_shape(tree,
        ↳ translation_matrix((-tree_min[0],
        ↳ -tree_min[1], 0)))
...

```

This program creates a fractal tree-like structure  
↳ using function recursion. Here's how it works:

1. The `'branch'` function is the core of the  
↳ recursion. It creates a cylinder for the main  
↳ branch and then recursively creates two smaller  
↳ branches at its top, rotated at angles.
2. The recursion depth is controlled by the `'depth'`  
↳ parameter. When `'depth'` reaches 0, it just  
↳ returns a single cylinder.
3. For each recursive call, the length and radius  
↳ of the branches are reduced by a factor.
4. The `'fractal_tree'` function is a wrapper that  
↳ calls `'branch'` and then adjusts the position of  
↳ the entire tree so that its base is at the  
↳ origin.

This shape is interesting because:

- It's self-similar at different scales, which is a  
↳ characteristic of fractals.
- It's not a simple geometric shape like a pyramid,  
↳ but a complex structure built from simple  
↳ components.
- The recursive nature of the function allows for  
↳ easy adjustment of the complexity by changing  
↳ the `'depth'` parameter.
- It resembles natural structures like trees or  
↳ blood vessels, demonstrating how complex  
↳ natural forms can arise from simple recursive  
↳ rules.

You can create trees of different complexity by  
↳ adjusting the `'depth'` parameter when calling  
↳ `'fractal_tree'`. For example:

```

- `library_call('fractal_tree', base_length=1.0,
    ↳ base_radius=0.1, depth=3)` for a simpler tree
- `library_call('fractal_tree', base_length=1.0,
    ↳ base_radius=0.1, depth=7)` for a more complex
    ↳ tree

```

This shape demonstrates the power of recursive  
↳ functions in creating complex, naturalistic  
↳ forms from simple rules.

**Example 4.** Input scene description: "Rene Magritte  
The Son of Man". Input editing instruction: "move  
the apple to the left".

Output:

Here's the minimally edited program that moves the  
↳ apple to the left:

```

'''python
from helper import *

"""
Move the apple to left
"""

@register()
def suit(width: float, height: float, depth: float)
    ↳ -> Shape:
    return primitive_call('cube', color=(0.1, 0.1,
        ↳ 0.1), shape_kwargs={'scale': (width,
        ↳ height, depth)})

@register()
def head(radius: float) -> Shape:
    return primitive_call('sphere', color=(0.9,
        ↳ 0.75, 0.65), shape_kwargs={'radius':
        ↳ radius})

@register()
def apple(radius: float) -> Shape:
    return primitive_call('sphere', color=(0.8,
        ↳ 0.1, 0.1), shape_kwargs={'radius': radius})

@register()
def hat(radius: float, height: float) -> Shape:
    brim = primitive_call('cylinder', color=(0.1,
        ↳ 0.1, 0.1), shape_kwargs={'radius': radius *
        ↳ 1.2, 'p0': (0, 0, 0), 'p1': (0, height *
        ↳ 0.1, 0)})
    top = primitive_call('cylinder', color=(0.1,
        ↳ 0.1, 0.1), shape_kwargs={'radius': radius,
        ↳ 'p0': (0, height * 0.1, 0), 'p1': (0,
        ↳ height, 0)})
    return concat_shapes(brim, top)

@register()
def son_of_man() -> Shape:
    # Create the suit
    suit_shape = library_call('suit', width=0.6,
        ↳ height=1.0, depth=0.3)

    # Create the head
    head_shape = library_call('head', radius=0.15)
    head_shape = transform_shape(head_shape,
        ↳ translation_matrix((0, 0.6, 0)))

    # Create the apple
    apple_shape = library_call('apple',
        ↳ radius=0.08)
    apple_shape = transform_shape(apple_shape,
        ↳ translation_matrix((-0.15, 0.62, 0.15))) #
        ↳ Changed x-coordinate to -0.15

    # Create the hat
    hat_shape = library_call('hat', radius=0.12,
        ↳ height=0.15)
    hat_shape = transform_shape(hat_shape,
        ↳ translation_matrix((0, 0.78, 0)))

    # Combine all shapes
    figure = concat_shapes(suit_shape, head_shape,
        ↳ apple_shape, hat_shape)

```

```

# Add a slight tilt to the figure
figure_center = compute_shape_center(figure)
tilted_figure = transform_shape(figure,
    ↪ rotation_matrix(0.1, (1, 0, 0),
    ↪ figure_center))

... return tilted_figure

The only change made to the program is in the
↪ `son_of_man()` function. Specifically, the
↪ translation of the apple shape has been
↪ modified. The x-coordinate in the translation
↪ matrix for the apple has been changed from 0 to
↪ -0.15:

'''python
apple_shape = transform_shape(apple_shape,
    ↪ translation_matrix((-0.15, 0.62, 0.15))) #
    ↪ Changed x-coordinate to -0.15
'''

This change moves the apple to the left by 0.15
↪ units in the scene's coordinate system, where
↪ the negative x-direction corresponds to the
↪ left side of the figure. The y and z
↪ coordinates remain the same, keeping the apple
↪ at the same height and depth relative to the
↪ face.

```

**Example 5.** Input scene description: "Paul Klee Castle and Sun". Input editing instruction: "change all castles to be the middle one".

Output:

```

Here's the minimally edited program to change all
↪ castles to be the middle one:

'''python
from helper import *

"""
Change all castles to be the middle one
"""

@register()
def rectangle(width: float, height: float, color:
    ↪ tuple[float, float, float]) -> Shape:
    return primitive_call('cube',
    ↪ shape_kwargs={'scale': (width, height,
    ↪ 0.01)}, color=color)

@register()
def triangle(base: float, height: float, color:
    ↪ tuple[float, float, float]) -> Shape:
    triangle_shape = primitive_call('cube',
    ↪ shape_kwargs={'scale': (base, height,
    ↪ 0.01)}, color=color)
    return transform_shape(triangle_shape,
    ↪ translation_matrix((0, height/2, 0)))

@register()
def sun(radius: float) -> Shape:
    return primitive_call('sphere',
    ↪ shape_kwargs={'radius': radius},
    ↪ color=(1.0, 0.8, 0))

@register()
def castle_block(width: float, height: float,
    ↪ color: tuple[float, float, float]) -> Shape:
    return library_call('rectangle', width=width,
    ↪ height=height, color=color)

@register()
def castle_tower(width: float, height: float,
    ↪ color: tuple[float, float, float]) -> Shape:
    tower = library_call('castle_block',
    ↪ width=width, height=height, color=color)
    roof = library_call('triangle', base=width,
    ↪ height=width/2, color=color)
    tower_height = compute_shape_sizes(tower)[1]
    return concat_shapes(

```

```

        tower,
        transform_shape(roof,
    ↪ translation_matrix((0, tower_height,
    ↪ 0)))
    )

@register()
def castle() -> Shape:
    base = library_call('castle_block', width=0.8,
    ↪ height=0.4, color=(0.5, 0.5, 0.7))
    center_tower = library_call('castle_tower',
    ↪ width=0.25, height=0.7, color=(0.5, 0.5,
    ↪ 0.7))

    base_width = compute_shape_sizes(base)[0]
    center_tower_width =
    ↪ compute_shape_sizes(center_tower)[0]

    return concat_shapes(
        base,
        transform_shape(center_tower,
    ↪ translation_matrix((-base_width/4, 0.4,
    ↪ 0))),
        transform_shape(center_tower,
    ↪ translation_matrix((base_width/4, 0.4,
    ↪ 0))),
        transform_shape(center_tower,
    ↪ translation_matrix((0, 0.4, 0)))
    )

@register()
def background() -> Shape:
    return library_call('rectangle', width=2,
    ↪ height=2, color=(0.95, 0.95, 0.8))

@register()
def castle_and_sun() -> Shape:
    bg = library_call('background')
    castle_shape = library_call('castle')
    sun_shape = library_call('sun', radius=0.15)

    return concat_shapes(
        bg,
        transform_shape(castle_shape,
    ↪ translation_matrix((0, -0.3, 0.01))),
        transform_shape(sun_shape,
    ↪ translation_matrix((0.5, 0.5, 0.02)))
    )

In this edited version, I made the following
↪ changes to fulfill the instruction "Change all
↪ castles to be the middle one":

1. In the `castle()` function, I removed the
    ↪ `left_tower` and `right_tower` variables.
2. I changed the `base` color to match the center
    ↪ tower's color: `(0.5, 0.5, 0.7)`.
3. I replaced the left and right towers with two
    ↪ additional center towers.
4. I adjusted the positioning of the three center
    ↪ towers to be evenly spaced across the base.

Now, when the `castle_and_sun()` function is
↪ called, it will create a scene where all the
↪ castle towers are identical to the original
↪ middle tower, fulfilling the given instruction.

```

**Example 6.** Input scene description: "7 different sized Russian nesting dolls lined up". Input editing instruction: "Shuffle order and rearrange to a small semicircle".

Output:

```

Here's the minimally edited program to change all
↪ castles to be the middle one:

'''python
from helper import *

"""
Change all castles to be the middle one
"""

```

```

@register()
def rectangle(width: float, height: float, color:
    tuple[float, float, float]) -> Shape:
    ↪ return primitive_call('cube',
    ↪ shape_kwargs={'scale': (width, height,
    ↪ 0.01)}, color=color)

@register()
def triangle(base: float, height: float, color:
    tuple[float, float, float]) -> Shape:
    ↪ triangle_shape = primitive_call('cube',
    ↪ shape_kwargs={'scale': (base, height,
    ↪ 0.01)}, color=color)
    ↪ return transform_shape(triangle_shape,
    ↪ translation_matrix((0, height/2, 0)))

@register()
def sun(radius: float) -> Shape:
    ↪ return primitive_call('sphere',
    ↪ shape_kwargs={'radius': radius},
    ↪ color=(1.0, 0.8, 0))

@register()
def castle_block(width: float, height: float,
    color: tuple[float, float, float]) -> Shape:
    ↪ return library_call('rectangle', width=width,
    ↪ height=height, color=color)

@register()
def castle_tower(width: float, height: float,
    color: tuple[float, float, float]) -> Shape:
    ↪ tower = library_call('castle_block',
    ↪ width=width, height=height, color=color)
    ↪ roof = library_call('triangle', base=width,
    ↪ height=width/2, color=color)
    ↪ tower_height = compute_shape_sizes(tower)[1]
    ↪ return concat_shapes(
    ↪     tower,
    ↪     transform_shape(roof,
    ↪         ↪ translation_matrix((0, tower_height,
    ↪         ↪ 0)))
    ↪ )

@register()
def castle() -> Shape:
    ↪ base = library_call('castle_block', width=0.8,
    ↪ height=0.4, color=(0.5, 0.5, 0.7))
    ↪ center_tower = library_call('castle_tower',
    ↪ width=0.25, height=0.7, color=(0.5, 0.5,
    ↪ 0.7))

    ↪ base_width = compute_shape_sizes(base)[0]
    ↪ center_tower_width =
    ↪     compute_shape_sizes(center_tower)[0]

    ↪ return concat_shapes(
    ↪     base,
    ↪     transform_shape(center_tower,
    ↪         ↪ translation_matrix((-base_width/4, 0.4,
    ↪         ↪ 0))),
    ↪     transform_shape(center_tower,
    ↪         ↪ translation_matrix((base_width/4, 0.4,
    ↪         ↪ 0))),
    ↪     transform_shape(center_tower,
    ↪         ↪ translation_matrix((0, 0.4, 0)))
    ↪ )

@register()
def background() -> Shape:
    ↪ return library_call('rectangle', width=2,
    ↪ height=2, color=(0.95, 0.95, 0.8))

@register()
def castle_and_sun() -> Shape:
    ↪ bg = library_call('background')
    ↪ castle_shape = library_call('castle')
    ↪ sun_shape = library_call('sun', radius=0.15)

    ↪ return concat_shapes(
    ↪     bg,
    ↪     transform_shape(castle_shape,
    ↪         ↪ translation_matrix((0, -0.3, 0.01))),
    ↪     transform_shape(sun_shape,
    ↪         ↪ translation_matrix((0.5, 0.5, 0.02)))
    ↪ )
...

```

In this edited version, I made the following  
 ↪ changes to fulfill the instruction "Change all  
 ↪ castles to be the middle one":

1. In the `'castle()'` function, I removed the  
 ↪ `'left_tower'` and `'right_tower'` variables.
2. I changed the `'base'` color to match the center  
 ↪ tower's color: `'(0.5, 0.5, 0.7)'`.
3. I replaced the left and right towers with two  
 ↪ additional center towers.
4. I adjusted the positioning of the three center  
 ↪ towers to be evenly spaced across the base.

Now, when the `'castle_and_sun()'` function is  
 ↪ called, it will create a scene where all the  
 ↪ castle towers are identical to the original  
 ↪ middle tower, fulfilling the given instruction.

## E2.3. Text-Conditioned 4D Generation

**Example 1.** Input: "carousel with a small  
 canopy".

Output:

Here's a program that creates a carousel with a  
 ↪ small canopy using the provided `'helper.py'`:

```

'''python
from helper import *

'''
carousel with a small canopy
'''

@register()
def carousel_base() -> Shape:
    ↪ base = primitive_call('cylinder',
    ↪ shape_kwargs={'radius': 2, 'p0': (0, 0, 0),
    ↪ 'p1': (0, 0.2, 0)}, color=(0.6, 0.4, 0.2))
    ↪ return base

@register()
def carousel_platform() -> Shape:
    ↪ platform = primitive_call('cylinder',
    ↪ shape_kwargs={'radius': 1.8, 'p0': (0, 0.2,
    ↪ 0), 'p1': (0, 0.3, 0)}, color=(0.8, 0.8,
    ↪ 0.8))
    ↪ return platform

@register()
def carousel_pole() -> Shape:
    ↪ pole = primitive_call('cylinder',
    ↪ shape_kwargs={'radius': 0.1, 'p0': (0, 0.3,
    ↪ 0), 'p1': (0, 3, 0)}, color=(0.7, 0.7,
    ↪ 0.7))
    ↪ return pole

@register()
def carousel_horse() -> Shape:
    ↪ body = primitive_call('cylinder',
    ↪ shape_kwargs={'radius': 0.2, 'p0': (0, 0,
    ↪ -0.4), 'p1': (0, 0, 0.4)}, color=(1, 1, 1))
    ↪ head = primitive_call('sphere',
    ↪ shape_kwargs={'radius': 0.15}, color=(1, 1,
    ↪ 1))
    ↪ head = transform_shape(head,
    ↪     ↪ translation_matrix((0, 0.2, -0.5)))
    ↪ legs = loop(4, lambda i: transform_shape(
    ↪     ↪     primitive_call('cylinder',
    ↪     ↪     ↪ shape_kwargs={'radius': 0.05, 'p0': (0,
    ↪     ↪     ↪ 0, 0), 'p1': (0, -0.5, 0)}, color=(1,
    ↪     ↪     ↪ 1, 1)),
    ↪     ↪     ↪ translation_matrix((0.15 * (-1 if i % 2 ==
    ↪     ↪     ↪ 0 else 1), -0.2, 0.3 * (-1 if i < 2
    ↪     ↪     ↪ else 1))))
    ↪ )
    ↪ horse = concat_shapes(body, head, legs)
    ↪ return transform_shape(horse,
    ↪     ↪ translation_matrix((0, 1, 0)))

@register()
def carousel_horses() -> Shape:
    ↪ return loop(4, lambda i: transform_shape(

```

```

        library_call('carousel_horse'),
        rotation_matrix(i * math.pi / 2, (0, 1, 0),
        ↪ (0, 0, 0)) @ translation_matrix((1.5,
        ↪ 0, 0))
    ))

@register()
def canopy() -> Shape:
    cone = primitive_call('cylinder',
    ↪ shape_kwargs={'radius': 2, 'p0': (0, 3, 0),
    ↪ 'p1': (0, 3.5, 0)}, color=(1, 0.5, 0.5))
    top = primitive_call('sphere',
    ↪ shape_kwargs={'radius': 0.1}, color=(1,
    ↪ 0.7, 0.7))
    top = transform_shape(top,
    ↪ translation_matrix((0, 3.6, 0)))
    return concat_shapes(cone, top)

@register()
def carousel() -> Shape:
    base = library_call('carousel_base')
    platform = library_call('carousel_platform')
    pole = library_call('carousel_pole')
    horses = library_call('carousel_horses')
    canopy_shape = library_call('canopy')
    return concat_shapes(base, platform, pole,
    ↪ horses, canopy_shape)

@register_animation()
def carousel_animation() -> Generator[Shape, None,
    ↪ None]:
    total_frames = 60
    for frame in range(total_frames):
        angle = frame * 2 * math.pi / total_frames
        rotated_carousel = transform_shape(
            library_call('carousel'),
            rotation_matrix(angle, (0, 1, 0), (0,
            ↪ 0, 0))
        )
    ... yield rotated_carousel

```

This program creates a carousel with a small  
 ↪ canopy. Here's a breakdown of the components:

1. `'carousel_base'`: A cylindrical base for the  
 ↪ carousel.
2. `'carousel_platform'`: A slightly smaller cylinder  
 ↪ on top of the base.
3. `'carousel_pole'`: A central pole for the  
 ↪ carousel.
4. `'carousel_horse'`: A simple horse shape made from  
 ↪ cylinders and a sphere.
5. `'carousel_horses'`: Four horses arranged around  
 ↪ the central pole.
6. `'canopy'`: A small conical canopy with a  
 ↪ spherical top.
7. `'carousel'`: Combines all the components to  
 ↪ create the complete carousel.
8. `'carousel_animation'`: An animation that rotates  
 ↪ the carousel.

The carousel is designed with a modular approach,  
 ↪ allowing easy modifications and adjustments to  
 ↪ individual components. The animation function  
 ↪ rotates the entire carousel, simulating its  
 ↪ movement.

## Example 2. Input: "solar system model".

### Output:

Here's a program that creates a simple solar system  
 ↪ model using the provided `'helper.py'`:

```

'''python
from helper import *

"""
solar system model
"""

@register()
def sun() -> Shape:

```

```

    return primitive_call('sphere',
    ↪ shape_kwargs={'radius': 0.5}, color=(1,
    ↪ 0.7, 0))

```

```

@register()
def planet(radius: float, color: tuple[float,
    ↪ float, float]) -> Shape:
    return primitive_call('sphere',
    ↪ shape_kwargs={'radius': radius},
    ↪ color=color)

```

```

@register()
def orbit(radius: float) -> Shape:
    points = 100
    orbit_shape = []
    for i in range(points):
        angle = 2 * math.pi * i / points
        x = radius * math.cos(angle)
        z = radius * math.sin(angle)
        orbit_point = primitive_call('sphere',
        ↪ shape_kwargs={'radius': 0.01},
        ↪ color=(0.5, 0.5, 0.5))
        orbit_shape.append(transform_shape(orbit_pj
        ↪ oint, translation_matrix((x, 0,
        ↪ z))))
    return concat_shapes(*orbit_shape)

```

```

@register()
def solar_system() -> Shape:
    sun_shape = library_call('sun')
    mercury_orbit = library_call('orbit',
    ↪ radius=1.5)
    venus_orbit = library_call('orbit', radius=2.5)
    earth_orbit = library_call('orbit', radius=3.5)
    mars_orbit = library_call('orbit', radius=4.5)

    return concat_shapes(
        sun_shape,
        mercury_orbit,
        venus_orbit,
        earth_orbit,
        mars_orbit
    )

```

```

@register_animation()
def solar_system_animation() -> Generator[Shape,
    ↪ None, None]:
    total_frames = 120

    for frame in range(total_frames):
        t = frame / total_frames

        sun = library_call('sun')

        mercury = transform_shape(
            library_call('planet', radius=0.1,
            ↪ color=(0.7, 0.7, 0.7)),
            rotation_matrix(t * 4 * math.pi, (0, 1,
            ↪ 0), (0, 0, 0)) @
            ↪ translation_matrix((1.5, 0, 0))
        )

        venus = transform_shape(
            library_call('planet', radius=0.15,
            ↪ color=(0.9, 0.7, 0.4)),
            rotation_matrix(t * 3 * math.pi, (0, 1,
            ↪ 0), (0, 0, 0)) @
            ↪ translation_matrix((2.5, 0, 0))
        )

        earth = transform_shape(
            library_call('planet', radius=0.17,
            ↪ color=(0.2, 0.4, 0.8)),
            rotation_matrix(t * 2 * math.pi, (0, 1,
            ↪ 0), (0, 0, 0)) @
            ↪ translation_matrix((3.5, 0, 0))
        )

        mars = transform_shape(
            library_call('planet', radius=0.13,
            ↪ color=(0.8, 0.3, 0.1)),
            rotation_matrix(t * 1.5 * math.pi, (0,
            ↪ 1, 0), (0, 0, 0)) @
            ↪ translation_matrix((4.5, 0, 0))
        )

```



```

solar_system = library_call('solar_system')

yield concat_shapes(solar_system, sun,
    ↳ mercury, venus, earth, mars)
...

```

This program creates a simple solar system model  
↳ with the following components:

1. A sun at the center.
2. Four planets (Mercury, Venus, Earth, and Mars)  
↳ orbiting around the sun.
3. Orbit paths for each planet.

The `solar_system_animation` function generates an  
↳ animation of the planets orbiting around the  
↳ sun. Each planet moves at a different speed,  
↳ simulating their relative orbital periods.

Here's a breakdown of the functions:

```

- `sun()`: Creates a yellow sphere representing the
  ↳ sun.
- `planet(radius, color)`: Creates a sphere with
  ↳ the given radius and color, representing a
  ↳ planet.
- `orbit(radius)`: Creates a circular orbit path at
  ↳ the given radius.
- `solar_system()`: Combines the sun and orbit
  ↳ paths into a static solar system model.
- `solar_system_animation()`: Animates the planets
  ↳ orbiting around the sun.

```

The animation runs for 120 frames, with each planet  
↳ completing a different fraction of its orbit in  
↳ that time. The sun remains stationary at the  
↳ center while the planets rotate around it.

### Example 3. Input: "a toy wind turbine".

Output:

Here's a program to create a toy wind turbine using  
↳ the provided `helper.py`:

```

```python
from helper import *

"""
a toy wind turbine
"""

@register()
def turbine_base() -> Shape:
    base = primitive_call('cylinder',
        ↳ shape_kwargs={'radius': 0.2, 'p0': (0, 0,
        ↳ 0), 'p1': (0, 2, 0)}, color=(0.5, 0.5,
        ↳ 0.5))
    return base

@register()
def turbine_nacelle() -> Shape:
    nacelle = primitive_call('cube',
        ↳ shape_kwargs={'scale': (0.6, 0.4, 0.8)},
        ↳ color=(0.7, 0.7, 0.7))
    return transform_shape(nacelle,
        ↳ translation_matrix((0, 2.2, 0)))

@register()
def turbine_blade() -> Shape:
    blade = primitive_call('cube',
        ↳ shape_kwargs={'scale': (0.1, 1.5, 0.05)},
        ↳ color=(1, 1, 1))
    return transform_shape(blade,
        ↳ translation_matrix((0, 0.75, 0)))

@register()
def turbine_rotor() -> Shape:
    return loop(3, lambda i: transform_shape(
        ↳ library_call('turbine_blade'),
        ↳ rotation_matrix(i * 2 * math.pi / 3, (0, 0,
        ↳ 1), (0, 0, 0)))

@register()
def wind_turbine() -> Shape:

```

```

base = library_call('turbine_base')
nacelle = library_call('turbine_nacelle')
rotor = transform_shape(library_call('turbine_
    ↳ rotor'), translation_matrix((0, 2.2,
    ↳ 0.4)))
return concat_shapes(base, nacelle, rotor)

```

```

@register_animation()
def wind_turbine_animation() -> Generator[Shape,
    ↳ None, None]:
    total_frames = 60

    for frame in range(total_frames):
        angle = frame * 2 * math.pi / total_frames

        base = library_call('turbine_base')
        nacelle = library_call('turbine_nacelle')
        rotor = transform_shape(
            ↳ library_call('turbine_rotor'),
            ↳ rotation_matrix(angle, (0, 0, 1), (0,
            ↳ 2.2, 0.4)) @ translation_matrix((0,
            ↳ 2.2, 0.4))
        )

        yield concat_shapes(base, nacelle, rotor)
...

```

This program creates a toy wind turbine with the  
↳ following components:

1. `turbine_base`: A cylindrical base for the  
↳ turbine.
2. `turbine_nacelle`: A cuboid representing the  
↳ nacelle (housing) at the top of the turbine.
3. `turbine_blade`: A single blade of the turbine.
4. `turbine_rotor`: Three blades arranged in a  
↳ rotor configuration.
5. `wind_turbine`: The complete wind turbine,  
↳ combining all the components.

The `wind_turbine_animation` function creates an  
↳ animation of the turbine with rotating blades.  
↳ It generates 60 frames, rotating the rotor by a  
↳ full 360 degrees over the course of the  
↳ animation.