3D-GENERALIST: Vision-Language-Action Models for Crafting 3D Worlds

Supplementary Material

6. Additional Details of 3D-GENERALIST

Here, we provide more detail for the implementation of various components in 3D-GENERALIST.

6.1. Scene-Level Policy

The base prompt we use as input to our VLM is:

```
**Task**: As a programmer, you are required to complete
     an implementation. Use a Chain-of-Thought approach
     to break down the problem, create pseudocode, and
     then write the code in Python language. Ensure that
      your code is efficient, readable, and well-
     commented.
Return the requested information from the function you
     create. Remember to call the function your create
     towards the end.

**Dos and Don'ts**:
* Only use the API functions provided. Please do not
     import any of them. You can use the functions
     directly, assuming they are already imported.
* Give scene elements unique and descriptive names. For
     example, brown_leather_sofa, ceiling_light,
     tall_floor_lamp.
* You can duplicate or remove assets and lights by
     manipulating the attribute "placements" of the
     asset instance or just delete the entry from the
     scene.objects dictinoary.
* Make sure to provide detailed but key-word focused
     descriptions of any new objects or materials you
     aim to retrieve. For example, "L-shaped, brown
     leather sofa with plush seat and colorful pillow"
     instead of "a sofa".
* Positions are bounding box center in meters, given as
     [x, y, z]. Rotations are in degrees. The assets are
      upright, so you only have to change and specify
     the one number: z-axis rotation.
* Make all modifications to the SceneDef instance named
     "scene" as we will read out the results from the
     variable named `scene`.
* Do not re-initialize the scene by calling SceneDef()
     again. The scene is already initialized and you can
      access the current scene using the variable "scene
     ".
* Note that the lighting and material attributes are by
     default None. You need to set them by instantiating
      the class Lighting or Material if you want to
     change the lighting or material of the scene
     element.
* Make sure to always specify the position and rotation
     of newly added objects, otherwise the objects will
     be added at the origin.

**3D Convention:**
- We use a right-handed coordinate system.
- For the scene, the X-Y plane is the floor, the Z axis
     points up to the ceiling.
- For each asset, the front face points in the positive
     X axis. The Z-axis points up. For example, a z-axis
      rotation of 90 means that the object will be
     rotated to face the positive Y axis.
- The bounding box of the asset is aligned with the
     asset's local frame. The local origin is at the
     center in the X-Y plane and the bottom of the asset
      in the Z axis. An object having a z-axis position
     of 0 means that object is on the floor.
```

Scene Domain Specific Language Here, we detail the types of descriptors defined in our Scene DSL. The *Category* descriptor, C, specifies the type of the scene element, where $C \in \{\text{floors}, \text{walls}, \text{ceilings}, \text{objects}\}$. The *Placement* descriptor, $P = [(x, y, z), \theta_z, s]$, defines the spatial attributes of the element, where $(x, y, z) \in \mathbb{R}^3$ specifies the position in 3D space, $\theta_z \in [0, 2\pi]$ represents the rotation around the z-axis, and $s \in \mathbb{R}^3$ is the scale of the element. Placement can include a list of such tuples to allow for multiple placements of a particular scene element. The *Material* descriptor, $M = \text{desc}$, provides a natural language description ($\text{desc} \in \text{NL}$) of the element's surface properties. The *Lighting* descriptor, $L = (t, i, c)$, specifies the type of the light, $t \in \{\text{point}, \text{directional}, \text{area}\}$, along with its intensity $i \in \mathbb{R}^+$ and color $c \in \mathbb{R}^3$, where $c = (r, g, b)$ and $r, g, b \in [0, 1]$ represent the normalized RGB color values. Finally, the *Metadata* descriptor, D, captures additional category-specific attributes, details of which are provided in the supplementary materials. Below, we provide the implementation in Python:

```
from pydantic import BaseModel, Field, conint
from typing import List, Optional, Dict, Literal, Any
import math
import numpy as np

class Placement(BaseModel):
    position: List[float] = Field(description="(x, y, z)
          position of the asset. z=0 means the asset is
         on the ground.", default=[0, 0, 0])
    rotation: List[float] = Field(description="(x, y, z-
         axis) Rotation of the asset in degrees. Most
         assets are upright in the source data, so
         rotation is mostly around the z-axis.", default
         =[0, 0, 0])
    scale: float = Field(description="Axis-aligned size
         of the bounding box before the rotation is
         applied", default=1.)

class Material(BaseModel):
    id: str = Field(description="ID of the material")
    description: str = Field(description="Description of
          the material")

class Lighting(BaseModel):
    is_ceiling_light: bool = Field(description="Whether
         the lighting is a ceiling light", default=False
         )
    light_type: Literal["spotlight", "directional", "
         point"] = Field(description="Type of the light
         ", default="point")
    energy: float = Field(description="Energy /
         intensity of the light in lux (indoor lights
         typically range from 100 to 1000)", default
         =100)
    color: List[float] = Field(description="Color of the
          light in RGB format (0-1, not 0-255)", default
         =[1, 0.75, 0.6])

class SceneElement(BaseModel):
    description: str = Field(description="Detailed
         natural language description of the scene
         element", default="")
    category: Literal["floors", "walls", "ceilings", "
         doors", "windows", "objects"] = Field(
         description="Category of the scene element")
    placements: List[Placement] = Field(description="
         Placement information for the scene element")
    bbox_size: Optional[List[float]] = Field(description
         ="Axis-aligned size of the bounding box before
         the rotation is applied", default=None)
    material: Optional[Material] = Field(description="
         Default material applied to the scene element (
         applicable for floors, walls, ceilings, doors,
         windows)", default=None)
    lighting: Optional[Lighting] = Field(description="
         Lighting attached to the scene element that
         emits light (only applicable for 'objects'
         category)", default=None)
    metadata: Optional[Dict[str, Any]] = Field(
         description="Additional metadata for the scene
         element", default=None)

class SceneDef(BaseModel):
    objects: Dict[str, SceneElement] = Field(description
         ="Dictionary of assets in the scene with asset
         variable name as keys. The asset variable name
         should be a valid Python variable name.",
         default={})
    room_type: Optional[str] = Field(description="room
         type", default="an interior scene")

scene = SceneDef()
```

Asset-level policy details After mesh-based surface detection, we find valid surfaces on the receptacle object by identifying all planes with normals deviating from [0, 0, 1] by less than 10 degrees then cluster them together and keep the clusters above a threshold area.

Functional APIs We provide the set of functions that we provide to our VLM in *Scene-Level Policy*, mainly **retrieve_material** to retrieve materials and **add_object** to retrieve 3D assets from language.

```
This is the documentation for the functions you have
     access to. You may call any of these functions to
     help you complete the task.

retrieve_material(description: str) -> scene_definition.
     Material:
'retrieve_material' is a tool that can retrieve a PBR
     material from language description.
    It returns a dictionary containing the id and
         description of the retrieved material.

    Parameters:
        description (str): The description of the
             material to change the wall to

    Returns:
        Material: A dictionary containing the id and
             description of the retrieved material.


add_object(description: str, bbox_size: list = None, num
     : int = 1) -> scene_definition.SceneElement:

    'add_object' is a tool that can retrieve an asset
         from the database from language description.
    It is necessary to call this function to add any new
          object to the scene, including lighting
         related objects.
    It returns a dictionary containing the id,
         description, size, and other optional metadata
         of the retrieved asset.
    Note that the asset is added to the origin (center
         of the scene) with no rotation so you might
         need to adjust the position and rotation of the
          retrieved asset.

    Parameters:
        description (str): The description of the asset
             to retrieve
        bbox_size (list, optional): A list of 3 floats [
             x, y, z] specifying the desired dimensions
             of the object in meters.
                                   This controls the
                                        size of the
                                        generated object
                                        . The z-axis
                                        value denotes
                                        the height of
                                        the object.
                                        Defaults to
                                        [1.0, 1.0, 1.0].
        num (int, optional): Number of instances of this
              object to create. Defaults to 1.

    Returns:
        SceneElement: a scene element containing the id,
              description, size, and other optional
             metadata of the retrieved asset.


load_image(image_path: str) -> numpy.ndarray:
'load_image' is a utility function that loads an image
     from the given file path string.

    Parameters:
        image_path (str): The path to the image.

    Returns:
        np.ndarray: The image as a NumPy array.
```

In-Context Library Our in-context library is a curated collection of code snippets that have demonstrated at least a 10% improvement in CLIP-alignment metrics when modifying a 3D environment. These snippets serve as in-context examples during generation. By randomly sampling from this library when producing candidate actions, the policy is nudged toward more diverse and effective action codes. After each self-improvement training round, newly discovered high-quality prompt and action code pairs are appended to this library. Below, we showcase an example code snippet from our in-context library for the prompt: *a chic hair salon with round mirrors, pink chairs, and pot plants*.

```
# Add pink salon chairs
pink_salon_chairs = add_object("plush pink salon chair",
     num=4)
scene.objects['pink_salon_chairs'] = pink_salon_chairs

# Position the chairs along a wall
scene.objects['pink_salon_chairs'].placements[0].
     position = [-1.5, 1.5, 0]
scene.objects['pink_salon_chairs'].placements[0].
     rotation = [0, 0, 0]
scene.objects['pink_salon_chairs'].placements[1].
     position = [-0.5, 1.5, 0]
scene.objects['pink_salon_chairs'].placements[1].
     rotation = [0, 0, 0]
scene.objects['pink_salon_chairs'].placements[2].
     position = [0.5, 1.5, 0]
scene.objects['pink_salon_chairs'].placements[2].
     rotation = [0, 0, 0]
scene.objects['pink_salon_chairs'].placements[3].
     position = [1.5, 1.5, 0]
```

```
scene.objects['pink_salon_chairs'].placements[3].
     rotation = [0, 0, 0]

# Add round mirrors on the walls above the chairs
round_mirrors = add_object("large round mirror", num=4)
scene.objects['round_mirrors'] = round_mirrors

scene.objects['round_mirrors'].placements[0].position =
     [-1.5, 1.5, 1.5]  # Positional match to the chair
scene.objects['round_mirrors'].placements[0].rotation =
     [0, 0, 0]
scene.objects['round_mirrors'].placements[1].position =
     [-0.5, 1.5, 1.5]
scene.objects['round_mirrors'].placements[1].rotation =
     [0, 0, 0]
scene.objects['round_mirrors'].placements[2].position =
     [0.5, 1.5, 1.5]
scene.objects['round_mirrors'].placements[2].rotation =
     [0, 0, 0]
scene.objects['round_mirrors'].placements[3].position =
     [1.5, 1.5, 1.5]
scene.objects['round_mirrors'].placements[3].rotation =
     [0, 0, 0]

# Add pot plants
pot_plants = add_object("stylish pot plant", num=4)
scene.objects['pot_plants'] = pot_plants
scene.objects['pot_plants'].placements[0].position =
     [-2.9, 3.9, 0]  # Corners of the room
scene.objects['pot_plants'].placements[0].rotation = [0,
      0, 0]
scene.objects['pot_plants'].placements[1].position =
     [-0.9, 3.9, 0]
scene.objects['pot_plants'].placements[1].rotation = [0,
      0, 0]
scene.objects['pot_plants'].placements[2].position =
     [1.1, 3.9, 0]
scene.objects['pot_plants'].placements[2].rotation = [0,
      0, 0]
scene.objects['pot_plants'].placements[3].position =
     [2.9, 3.9, 0]
scene.objects['pot_plants'].placements[3].rotation = [0,
      0, 0]

# Update materials
scene.objects['walls_0'].material = retrieve_material("
     smooth light pink plaster wall")
scene.objects['floors'].material = retrieve_material("
     glossy light wood flooring")
scene.objects['ceilings'].material = retrieve_material("
     sleek white matte ceiling")

# Add modern lighting
modern_ceiling_lights = add_object("modern ceiling light
      fixture", num=1)
scene.objects['ceiling_lights'] = modern_ceiling_lights
scene.objects['ceiling_lights'].placements[0].position =
      [0, 0, 3.5]  # Center of the ceiling
scene.objects['ceiling_lights'].placements[0].rotation =
      [0, 0, 0]
# Define the ceiling light parameters
modern_lighting = Lighting(
    is_ceiling_light=True,
    light_type="point",
    energy=700,  # Bright energy
    color=[1.0, 1.0, 1.0]  # Neutral white light for
         natural illumination
)

# Attach the lighting to the ceiling light fixture
scene.objects['ceiling_lights'].lighting =
     modern_lighting
```

6.2. Asset-Level Policy

We provide the base prompts we fed to our VLM policy (i.e., Molmo-7B [13]) at each step of the iteration:

- Description prompt

```
You are iteratively placing small objects on the {
     base_object_description} through multiple rounds
Your current goal is to add one object to the current
     scene as shown in the image.
Give a concrete description in less than 15 words of the
      appearance of a small object that makes sense to
     be placed on {base_object_description} given what
     you already see placed.
No need to justify your choice, just describe the placed
      object in less than 15 words.
```

- Location prompt

```
You are looking at the following object: {
     base_object_description}.
Point to one location where it makes the most sense to
     place {retrieved_object_description} such that it
     does not collide with other objects but could even
     be on inside or on top of another object.
```

Initially, we prompted the VLM for $\mathcal{O}$, a set of plausible placement locations, and then iterated through them, prompting the VLM for descriptions of objects to place at each location o_i'. This did not work as the pointing mechanism of the VLM tended to cluster points nearest the camera, and much of the object was occluded from view. Empirically, randomizing view points can alleviate this issue. We also found qualitatively that it is important to use the VLM prior to decide where to place an asset given the language description of the asset, as illustrated in Figure 8.

Our final implementation prompts the VLM for an object to place then prompts for a pixel location.

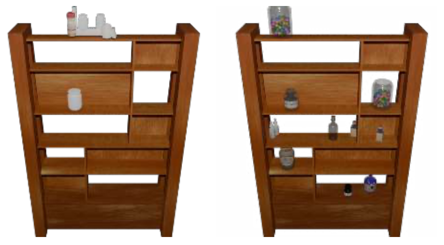

Figure 8. We experimented with two approaches for the asset level policy: prompt the VLM to point to a location then prompt for a description of the object to add (left) or prompt the VLM for a description of the next object to add then prompt it for a location (right). Note that in the right image, we provide the object as context to the VLM which helps to better space out the placements.

6.3. Limitations

A key limitation of 3D-GENERALIST is that it is fine-tuned to generate prompt-aligned 3D environments measure by CLIP similarity score, which is a metric not inherently sensitive to different spatial arrangements of the same set of assets. As a result, generated scenes often lack nuanced spatial reasoning, leading to layout that is not necessary *functionally correct* as shown in the top-right example in Figure 9.

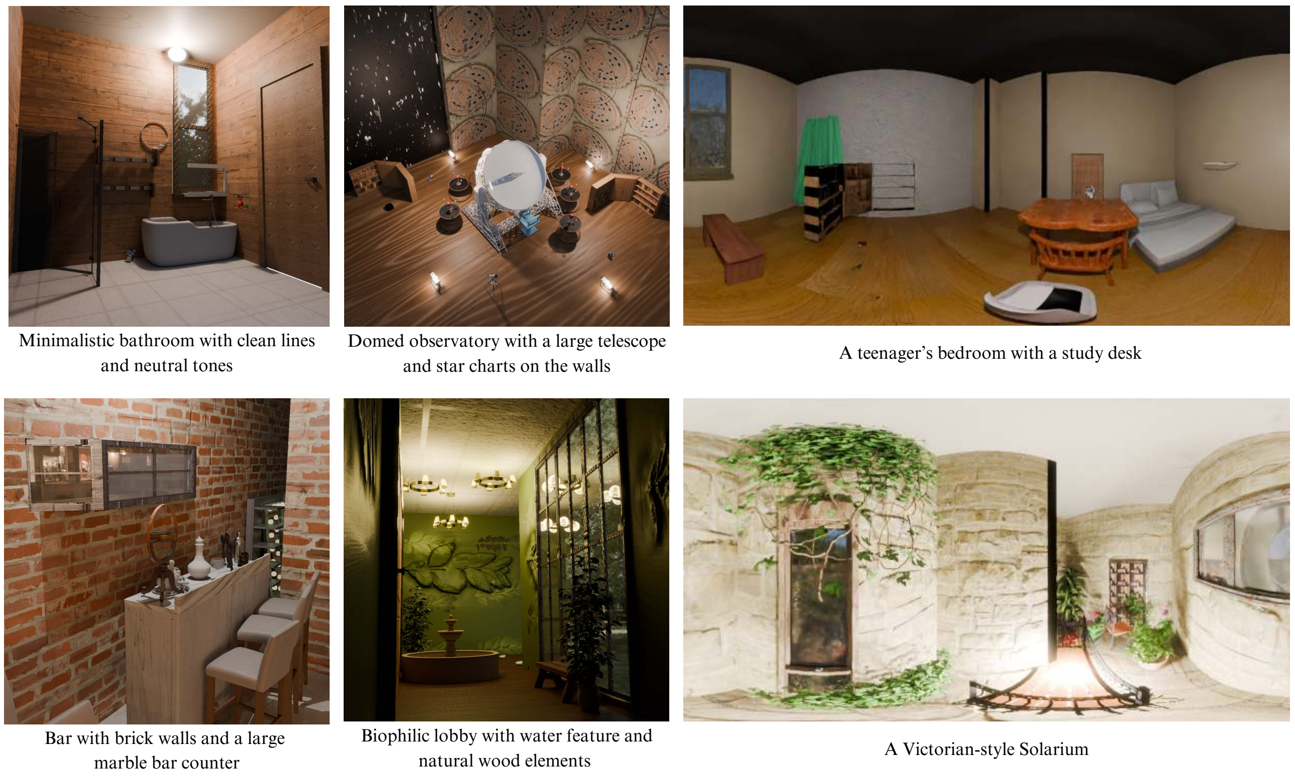
Minimalistic bathroom with clean lines and neutral tones
Domed observatory with a large telescope and star charts on the walls
A teenager's bedroom with a study desk
Bar with brick walls and a large marble bar counter
Biophilic lobby with water feature and natural wood elements
A Victorian-style Solarium

Figure 9. More results from 3D-GENERALIST. The rightmost column contains two examples of panoramic image renderings.

7. Details of our Experiments

We make design choices for our experiments to remain within a reasonable time and compute budget. Instead of using text or image-to-3D asset generators [59], diffusion models for PBR material synthesis, or the latest state-of-the-art Vision-Language Models to enhance generation quality, we rely on asset and material repositories for retrieval and use GPT-4o as the base model (except that we use Molmo-7B [13] to output pixel locations in *Asset-Level Policy*) and its fine-tuning API. In all experiments and baselines, we applied a combination of manual and automated filtering techniques to Objaverse [12], resulting in a curated set of 200,000 3D assets, following the methodology outlined in [45]. Additionally, we scraped ambientCG[2], obtaining 2,000 PBR materials. It is important to note that all baselines utilized the same asset and material repositories.

Below, we first detail the two chosen general image-domain benchmarks commonly used to evaluate Vision-Language Models in one of our experiments in evaluating 3D-GENERALIST's *Scene-Level Policy*. Subsequently, we discuss our implementation of the baseline in our experiment for *Asset-Level Policy*.

[2]https://ambientcg.com/

7.1. VLM Benchmarks

Here we detail the two benchmarks we use in section 4.1 to evaluate 3D-GENERALIST's *Scene-Level Policy*'s visual grounding abilities.

Object HalBench [38] compares the objects mentioned in the model's generations to the annotated objects from COCO images [27], thus providing an object-level hallucination rate for the model's image descriptions. We follow [58]'s implementation with augmented prompting and `gpt-4-turbo` judging hallucinations, and report the CHAIR scores (frequency of hallucinatory objects in model responses) at both the response level (CHAIRs) and object level (CHAIRi).

AMBER [47] assesses the VLM's hallucinations with fine-grained object, attribute, and relation annotations. We follow the official implementation for the generative task of AMBER and report the metrics CHAIR, COVER (ground-truth object coverage), HAL (rate of hallucinated responses), and COG (rate of hallucinatory objects similar to human cognition).

7.2. Scene-Level Policy's Evaluation

Baseline choice In our experiment, we compared to LayoutGPT [17] and Holodeck [55]. We were unable to compare against SceneCraft [23], since its source code is not available. Although AnyHome [19] has released part of the source code, the full release has not yet occurred. PhyScene [54] is not an *open-vocabulary* layout generation method, it can only generate layout for a fixed set of categories.

7.3. Asset-Level Policy's Evaluation

ProcTHOR [11] also has a method to place small objects on surfaces of "receptacle objects". However, rather than being placed according to a natural language prompt, objects are placed based on the frequency that a given object type appears on the receptacle in the hand-modeled AI2thor and RoboTHOR datasets. Thus, we opt to compare with an inpainting-based baseline mainly.

As ARCHITECT [50]'s code is unreleased at the time of writing this paper, we implemented a similar method that inpaints the placeable areas, uses a VLM to identify the objects, gets each object's bounding box to determine placement location, then retrieves objects according to their description and places them. More specifically, the inpainting model takes as input a rendered view of the object and a mask of the areas to generate. We render a front or top-front view of the "receptacle asset" and generate a mask of the placeable areas (on a table, between shelves) by placing a cube in or on top of the asset slightly smaller than the object's bounding box. We then use GPT-4o to create a list of objects and prompt GroundingDino [36] with the inpainted image to retrieve bounding boxes. Using the pixel location at the center of the bottom of each bounding box and the camera parameters, we generate a 3D ray to get a precise placement location. To remain consistent with our iterative method, we restrict placement to a maximum of 10 of the identified objects.

7.4. GPU cost

Unlike existing monolithic pipelines, 3D-Generalist's iterative design allows rendering and reasoning over complex scenes in a way that can scale with compute. All our experiments are conducted with A100s, with 1-3 minutes needed to generate the 3D scenes shown in Figure 1. Unlike existing monolithic pipelines, 3D-Generalist's iterative design allows rendering and reasoning over complex scenes in a way that can scale with compute. All our experiments are conducted with A100s, requiring 1-3 minutes to generate the 3D scenes shown in Figure 1.