VULCAN: Tool-Augmented Multi Agents for Iterative 3D Object Arrangement

Supplementary Material

1. Comparison on Abstract Prompts

We demonstrate that our proposal can flexibly handle both abstract prompts (requiring model to decompose the task into action plans) and detailed multi-step instructions (requiring the model to faithfully execute instructions sequentially). Qualitative examples for both cases are provided in Fig. 5, Fig. 6, Fig.7, and Fig.8.

While the quantitative results for multi-step instructions are detailed in the main paper, we provide an additional quantitative comparison here using abstract instructions across 12 scenes. Unlike the multi-instruction setting where we evaluated all intermediate edited imagery, here we focus solely on the final output (i.e., the scene state after all edits are completed). We compare our approach against two baselines: BlenderAlchemy [3] and Blender-MCP [2]. As shown in Tab. 1, our model significantly outperforms all baselines.

Method	Coll.%↓	Fl.%↓	Plaus.↑	Const.↑
BlenderAlchemy [3]	0.500	0.333	3.433	2.833
Blender-MCP [2]	0.416	0.333	3.800	2.517
Ours	**0.000**	**0.000**	**4.000**	**3.383**

Table 1. **Evaluation on General Instructions.** Our model outperforms baselines and produces artifact-free outputs.

2. Details of the Multi-Tool Library

We provide detailed specifications of our multi-tool library, implemented using Blender's Python API (bpy) [1]. We start with the visual API functions, followed by our collision-free solver, the constraint loss functions, and finally the collision/floating detector.

2.1. Visual Probing Functions

The visual probing functions are iteratively called by the *Executor* agent to analyze the scene and select relevant objects or planes for arrangement. The library consists of three primary functions:

ListObjectsInArea returns the names of all objects within a specified image region. This is achieved by rendering an instance map of the scene, extracting all unique IDs from pixels within the input area, and retrieving the object names corresponding to those IDs.

RayProbe returns information regarding the first hitting point of a camera ray given its pixel coordinates. We implement this using bpy's internal ray-casting function to retrieve the position, the surface normal and the object name of the ray's first hit. Additionally, this function extracts the planar surface of the object intersected by the ray. This is accomplished via a breadth-first traversal to identify all neighboring mesh faces of the intersected face that have surface normals within a cosine distance of 0.05.

RenderWithHighlight generates a rendered image with selected objects highlighted in distinct colors. Similar to the previous functions, this process begins by rendering an instance map. Pixels corresponding to instance IDs in the selection list are then repainted with their assigned colors.

2.2. Constraint Loss Functions

In the main paper, we proposed a set of constraints for the solver's input. Here, we detail how each constraint is formulated as a differentiable loss.

CloseToPix penalizes object placements that deviate from the provided pixel coordinates. Given the object's 3D location $\boldsymbol{p}$ and the normalized target pixel coordinates $\boldsymbol{c}$, the loss is calculated as:

$$\mathcal{L}_{c2p} = \lambda_{c2p} \|\boldsymbol{c} - \mathrm{Proj}_C(\boldsymbol{p})\|_2^2, \tag{1}$$

where $\mathrm{Proj}_C(\cdot)$ denotes the camera projection function of camera C, and the weight parameter λ_{c2p} is set to 0.5.

Contact penalizes placements where the object fails to contact the target plane. Let the object's plane and the target's plane be defined by groups of vertices $\boldsymbol{p}_{1,\dots,n}$ and $\boldsymbol{q}_{1,\dots,m}$, respectively, and the target plane's normal direction denoted as $\boldsymbol{n}$. The loss consists of two components: $\mathcal{L}_{contact}^{touch}$, which enforces that the object's plane touches the target plane; and $\mathcal{L}_{contact}^{above}$, which ensures the object's plane resides fully on the outward side of the target plane. Mathematically, let $\bar{q} = \max_{j=1,\dots,m}(\boldsymbol{q}_j \cdot \boldsymbol{n})$ denote the plane's position along the normal direction. The losses are defined as:

$$\mathcal{L}_{contact}^{touch} = \min_{i=1,\dots,n}(|\bar{q} - \boldsymbol{p}_i \cdot \boldsymbol{n}|_1), \tag{2}$$

$$\mathcal{L}_{contact}^{above} = \frac{1}{n} \sum_{i=1,\dots,n} \mathrm{Relu}(\bar{q} - \boldsymbol{p}_i \cdot \boldsymbol{n}), \tag{3}$$

$$\mathcal{L}_{contact} = \lambda_{contact}^{touch} \mathcal{L}_{contact}^{touch} + \lambda_{contact}^{above} \mathcal{L}_{contact}^{above}. \tag{4}$$

where $\lambda_{contact}^{touch}$ and $\lambda_{contact}^{above}$ are both set to 100. Note that the contact loss also applies to non-parallel planes.

NoOverHang enforces that the object is placed within the bounds of the target plane. While objects can typically be placed fully within the target surface (e.g., a book

on a shelf), certain scenarios (e.g., stacking books) may require part of the object to overhang. To address this, we developed two modes for this loss: *full* mode, which penalizes any part of the object's plane extending beyond the target's convex hull; and *center-only* mode, which only evaluates the center of the object's plane. We first project the vertices of both planes ($\boldsymbol{p}_{1,\dots,n}$ and $\boldsymbol{q}_{1,\dots,m}$) onto the target plane's 2D space (denoted as $\hat{\boldsymbol{p}}$ and $\hat{\boldsymbol{q}}$). We then determine if any $\hat{\boldsymbol{p}}$ lies inside the convex hull $\hat{\boldsymbol{q}}^{cvx}_{1,\dots,m'} = \mathrm{ConvexHull}(\hat{\boldsymbol{q}}_{1,\dots,m})$ by applying a cross-product examination with the convex hull's edges $\hat{\boldsymbol{e}}_i = \hat{\boldsymbol{q}}^{cvx}_{i+1} - \hat{\boldsymbol{q}}^{cvx}_i$. The two modes are defined as:

$$\mathcal{L}^{full}_{OH} = \lambda_{OH} \max_{i,j}(\mathrm{ReLU}((\hat{\boldsymbol{p}}_i - \hat{\boldsymbol{q}}^{cvx}_j) \times \hat{\boldsymbol{e}}_j)), \tag{5}$$

$$\mathcal{L}^{center}_{OH} = \lambda_{OH} \max_{j}(\mathrm{ReLU}((\mathrm{avg}(\hat{\boldsymbol{p}}) - \hat{\boldsymbol{q}}^{cvx}_j) \times \hat{\boldsymbol{e}}_j)) \tag{6}$$

$$+ \lambda^{align}_{OH} \|\mathrm{avg}(\hat{\boldsymbol{p}}) - \mathrm{avg}(\hat{\boldsymbol{q}}^{cvx})\|, \tag{7}$$

where λ_{OH} and λ^{align}_{OH} are set to 20 and 1, respectively. In the *center-only* mode, an additional L_2 loss between the centers of the two planes is applied to encourage placement near the target plane's center. In practice, the agent selects the mode via an argument in the NOOVERHANG constraint. If *full* mode is selected, only the first formulation is applied. Otherwise, the solver first attempts to solve using *full* mode and switches to *center-only* mode if no solution is found.

DISTANCE constrains the distance between two objects to be close to a target value, $dist$. This is implemented as an L_2 loss between $dist$ and the Euclidean distance between the centers of the two objects:

$$\mathcal{L}_{dist} = \lambda_{dist} \|\boldsymbol{x}_o - \boldsymbol{x}_{tgt}\|^2_2, \tag{8}$$

where $\boldsymbol{x}_o$ and $\boldsymbol{x}_{tgt}$ represent the positions of the placed object and the target object, respectively, and λ_{dist} is set to 0.3.

FACETO enforces that an object faces a specific target object, plane, or camera. Let $\boldsymbol{v}_o$ denote the object's facing direction and $\boldsymbol{v}_{tgt}$ the target facing direction. The loss is calculated as the cosine distance between $\boldsymbol{v}_o$ and the projection of $\boldsymbol{v}_{tgt}$ onto the object's azimuth plane. Letting $\boldsymbol{u}_o$ be the up vector of the object, the loss is defined as:

$$\mathcal{L}_{FT} = \lambda_{FT} \mathrm{CosDist}(\boldsymbol{v}_o, \boldsymbol{v}_{tgt} - \mathrm{SG}((\boldsymbol{u}_o \cdot \boldsymbol{v}_{tgt})\boldsymbol{u}_o)), \tag{9}$$

where SG denotes the stop-gradient function, and λ_{FT} is set to 0.5. Analogously, we implement a **BACKTO** constraint using the same formulation to align the object's back with the target.

ROTATE explicitly sets the object's rotation to a specific degree. This is achieved by setting the initial rotation angle of the object to the input degree and fixing it during the optimization process. This constraint is deactivated if any FACETO/BACKTO constraint is present.

2.3. Collision-Free Solver

As outlined in Algorithm 1 of the main paper, the solver first perturbs the target pixel coordinates in the CLOSETOPIX constraint with a fixed standard deviation of 0.2, generating a batch of variant coordinates. It then optimizes a batch of target poses corresponding to each perturbed coordinate.

The positions are initialized at the first intersection point of a ray cast from the pixel coordinates, as determined by the RAYPROBE function. Initial rotation angles are set to zero, unless a ROTATE constraint is explicitly present. We employ the AdamW optimizer for 800 iterations, with a learning rate initialized at $1e-1$ and linearly decayed to $1e-4$. To simplify the optimization, we fix the rotation along the pitch and roll axes, optimizing only the rotation around the vertical (yaw) axis.

Upon completion, we re-evaluate the loss for each solution in the batch using the original, unperturbed coordinates for the CLOSETOPIX target. Solutions exceeding a loss threshold of $1e-1$ are discarded.

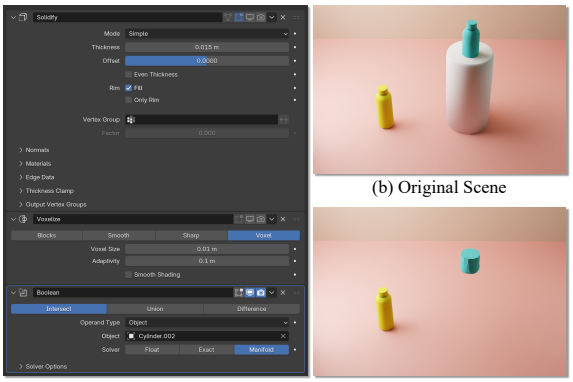
Finally, we assess the remaining solutions for collisions, processing them in ascending order of error. Since the standard Blender Python API (bpy) does not expose the native physics engine's collision detection, we leverage geometry modifiers to approximate this. Specifically, we first apply a SOLIDIFY modifier to inflate the meshes, ensuring robust detection for thin structures such as walls or floors. Then, we convert the source and target objects into manifold meshes via the REMESH modifier. We then employ a BOOLEAN modifier to compute the intersection volume between objects. A collision is declared if the intersection geometry penetrates the object's bounding box above a threshold of 0.01. The first solution found to be collision-free is selected as the final output. Figure 1 illustrates the modifiers in Blender, and demonstrates how our detector effectively calculates intersection volumes.

2.4. Floating Detection

Upon completion of the object arrangement, we conduct a floating check on all objects within the scene. This is implemented by casting a ray from the center of the bottom face of the object's bounding box. If the ray intersects another object within a distance of 0.01, the object is classified as grounded (i.e., not floating). The validation passes only if all objects that were grounded in the initial state remain grounded in the edited scene.

3. Efficiency Analysis

Our model requires minimal GPU memory ($\approx$ 3 GB) for rendering and the collision-free solver. Due to Gemini API latency variability, we report runtime excluding Gemini API processing. Tested on an NVIDIA A100, our model takes 184K tokens and 222.42s per scene in average (vs.



(a) Modifier Interface in Blender (b) Original Scene (c) Scene w/ Modifiers

Figure 1. **Collision Detector.** *(a)* Configuration of our collision detector, consisting of three modifiers with parameter settings as illustrated. *(b-c)* Visualization of the intersection: upon activation, the modifiers extract the collision geometry, i.e. the portion of the bottle below the cylinder's surface.

81K/95.54s for our single-agent variant and 97K/11.2s for the Blender-MCP baseline). While our multi-agent design increases computational cost, we argue that our primary goal is framed as a **test-time scaling technique** that maximizes the generation quality for this complex task, rather than optimizing for raw efficiency. To justify this trade-off, we compared our model against a naive scale-up baseline (i.e. Blender-MCP, a single-agent and solver-free baseline) that repeats agent calls $3\times$ for self-refinement on every task. As shown in Tab. 2, our structural design significantly outperforms the naive baseline even when token usage is normalized, justifying the complexity of the system.

Method	Token#	Coll.%↓	Fl.%↓	Plaus.↑	Const.↑
Blender-MCP	91K	0.459	0.774	3.348	2.973
Blender-MCP*	272K	0.441	0.640	3.267	2.672
Ours	184K	**0.000**	**0.000**	**3.796**	**3.592**

Table 2. **Comparison with baseline and its augmented version.**

4. Inference Statistics

To finish a scene in our benchmark, our model uses iterative steps to address each task sequentially. For each step, it attempts up to 4 branches until one passes all checks. If all branches fail, Backtracking is triggered to revert to a previous step. To analyze how inference varies by difficulty, we split our benchmark (25 scenes, 111 tasks) into easy (13 scenes, 52 tasks) and hard (12 scenes, 59 tasks) sets. On average, hard/easy sets required $1.33/1.03$ steps per task and explored $2.85/1.80$ branches per step. Of these branches, $25.3\%/7.2\%$ failed due to collision/floating errors, while $16.8\%/16.4\%$ failed due to Evaluator rejection. Our model effectively scales its search effort for harder cases.

5. Prompts for Agents

Planner. We employ distinct prompting strategies for general instructions and per-step instruction settings. For general instructions, we prompt the planner to autonomously generate a plan based on the current state and historical steps. To prevent infinite loops, we impose a maximum step limit for each task, typically ranging from 3 to 6 steps. The model is required to make a feasible plan within this constraint; if the estimated steps exceed this limit, the attempt is automatically classified as a failure.

In the detailed per-step instruction setting, the model is prompted to read the current instruction, analyze it within the global context (considering both past and future steps), and rewrite it such that the locally-operating *Executor* can execute it based solely on the current state. Figures 12 and 13 illustrate the prompts for the general instruction setting, while Fig. 14 shows the prompt for the detailed setting.

Executor. The prompt for the executor is presented in Fig. 15 and Fig. 16. Supplementing this, we provide API documentation (as shown in Fig. 11) for the visual tools, enabling the model to understand the functionality of each function and the definitions of their parameters.

Evaluator. Figure 17 shows the *Evaluator* prompt. This agent accepts the rendered image of the edited state and the instruction from the *Planner* instruction to assess the edit's plausibility and correctness. In practice, we implement an early termination for efficiency: If all *Evaluators* assign a *good* or *excellent* verdict to the current edit, we accept it as the result for that step and skip remaining attempts.

Benchmark Judges. In addition to the multi-agent pipeline, we provide the judge prompt used in our benchmark (Fig. 18), which is adapted from experiments in FirePlace [4]. This agent examines the results of all steps collectively, evaluating them within a global context to assign scores for each step. For each data sample, we employ 5 benchmark evaluators and report the average scores.

6. Adaptive Backtracking

Algorithm 1 presents detailed pseudocode for the adaptive backtracking mechanism employed within our search framework.

7. Additional Discussion on Failure Cases

Our model significantly enhances the backbone MLLM's capability for iterative object arrangement. However, despite robust design components, the system still can be affected by the inherent limitations of the backbone MLLM, as shown in Fig. 2.

Algorithm 1: Plan Search with Adaptive Backtracking SEARCH

Parameter: Object arrangement pipeline $\mathcal{F}$, maximum allowed steps s_{max}, user instruction T.

Input: Previous scenes $\mathcal{S} = \{S_{0,\dots,k-1}\}$, current step k, anchor depth d_a, current maximum depth d_{max}.

Output: A sequence of edited scenes $S_{0,\dots,k}$.

```
/* Run model for current step.    */
```
$state, S_k \leftarrow \mathcal{F}(\mathcal{S}, T, k)$;

if $state =$ *"Complete"* **then**

```
    /* Search is complete.        */
```
    return $\mathcal{S}$;

else if $k < s_{max}$ & $state =$ *"Edited"* **then**

```
    /* Current step is done.      */
```
    if $k + 1 > d_{max}$ **then**

```
        /* Reaches a breakthrough.
           Update anchor depth.   */
```
        $d_{max}, d_a \leftarrow k + 1$;

    SEARCH$(\mathcal{S} \cup \{S_k\}, k + 1, d_a, d_{max})$;

else

```
    /* Failed.  Backtracking.     */
```
    $d_a \leftarrow d_a/2$;

    SEARCH$(\{S_{0,\dots,d_a-1}\}, d_a, d_a, d_{max})$;

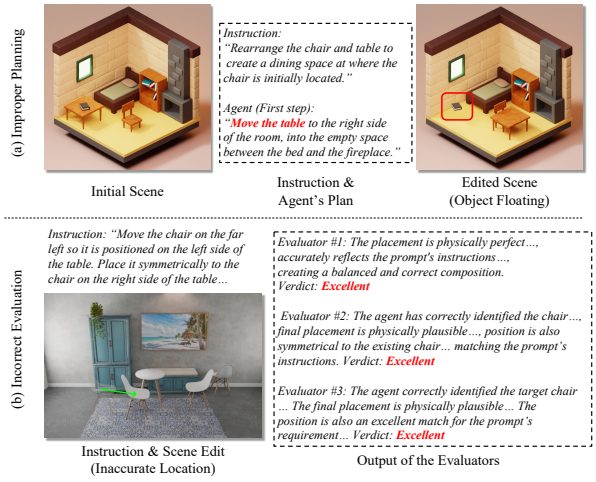


Figure 2. **Failure Cases.** *Top*: The *Planner* generates an improper plan (i.e., moving the table before clearing it), resulting in floating artifacts. *Bottom*: All three *Evaluators* provide false positive verdicts despite of the arrangement deviating from the instruction.

A primary failure mode is planning error. The MLLM may generate an incorrect plan for the current step, resulting in infeasible solutions. While our backtracking algorithm enables recovery from such errors, this trial-and-error process significantly impacts the model's efficiency.

Secondly, as noted in the main paper, the *Evaluator* may occasionally yield false positives (i.e., validating an erroneous arrangement) due to MLLM hallucinations. Although our polling mechanism mitigates this risk, there remains possibilities that the *Evaluators* will reach a consensus on an incorrect verdict. While future advancements in MLLMs may eliminate these issues, we believe that developing more robust evaluation agents for current MLLMs represents a promising direction for future research.

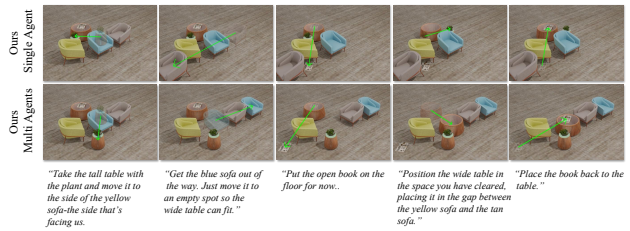


Figure 3. **Ablation on Multi-Agent Design.** Our model with multiple agents achieves more reasonable object arrangements compared to the single-agent variant.

Figure 4. **Results on Outdoor Scenes.** Our model generalizes to outdoor environments, successfully arranging objects in open-space scenes with diverse backgrounds and lighting conditions.

8. Additional Results

We provide supplementary visual results, including qualitative comparisons against baseline methods and extended demonstrations of our model's performance.

Additional Qualitative Comparisons. We present further visual comparisons with baseline methods. Figures 5 and 6 illustrate four examples under the per-step instruction setting, while Figures 7 and 8 shows four examples using general instructions.

Additional Qualitative Ablations. Fig. 3 qualitatively demonstrates the improvement of our multi-agent design over the single-agent variant, serving as a supplement to the quantitative ablation study in the main paper.

Additional Results. Figures 9 and 10 display additional results generated by our method under per-step and general instruction settings, respectively. In addition, we show results on outdoor scenes in Fig. 4.

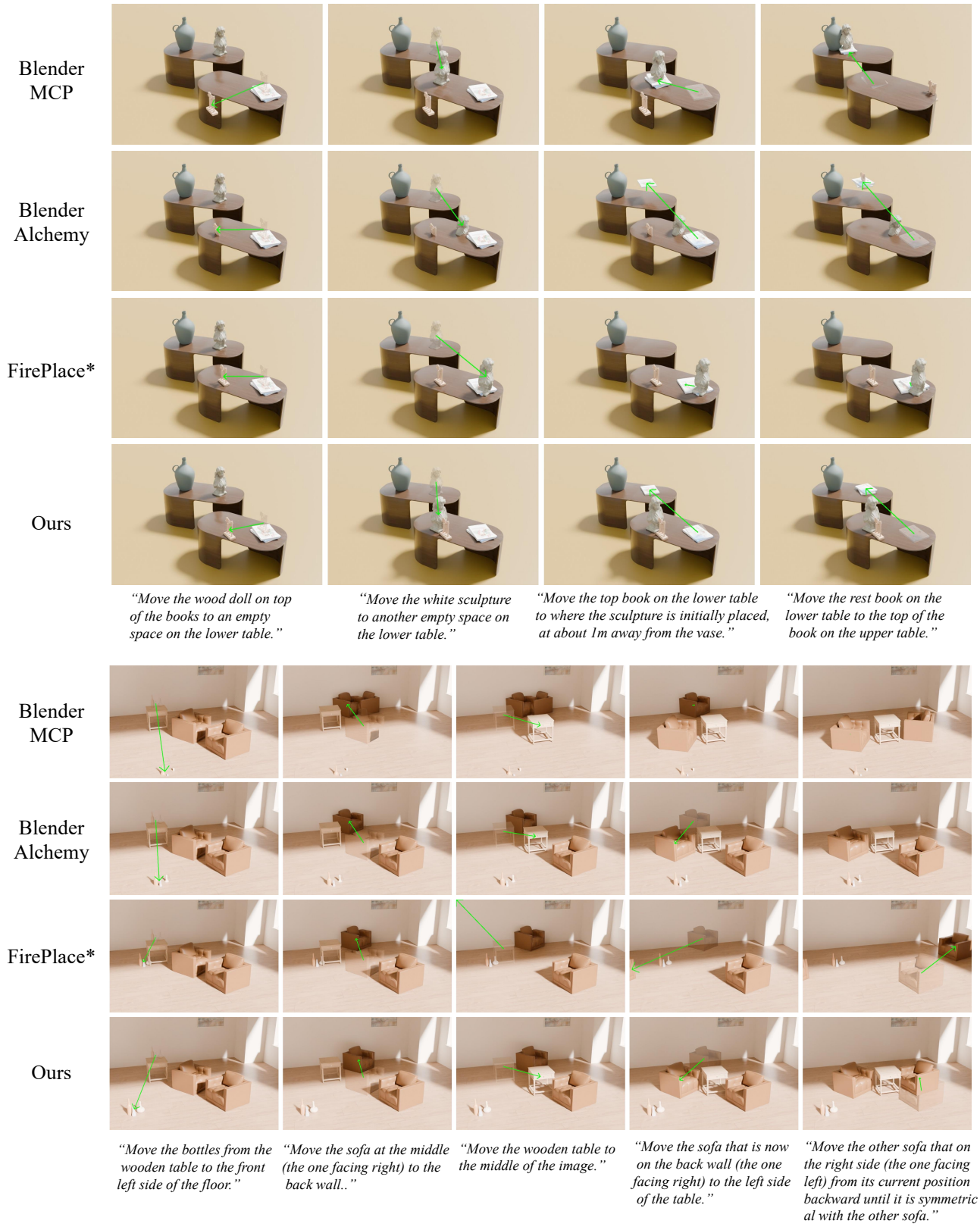


Figure 5. **Comparison on Per-step Instruction Examples.** The input per-step instructions are shown below the images.

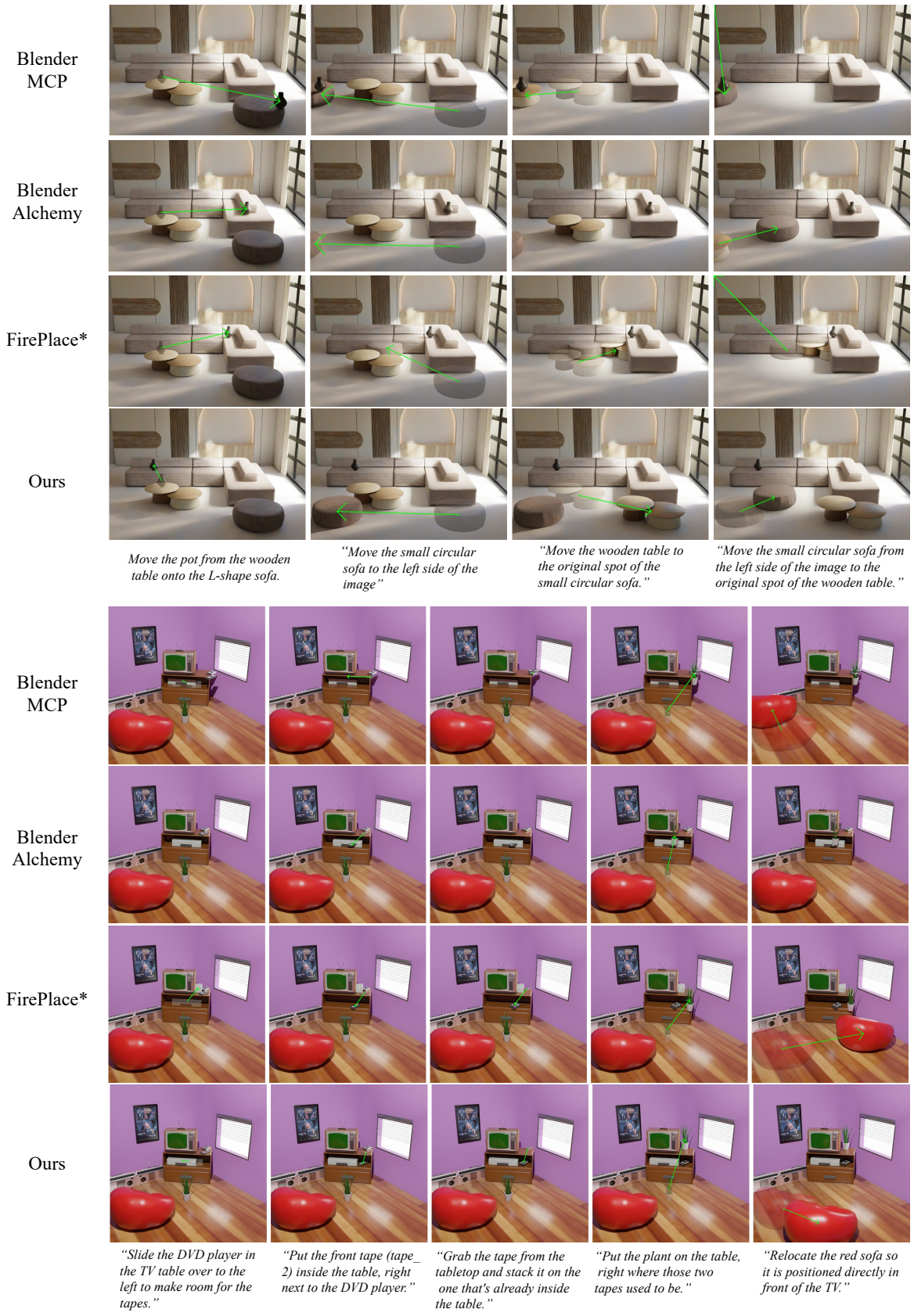


Figure 6. **Comparison on Per-step Instruction Examples (cont'd).**

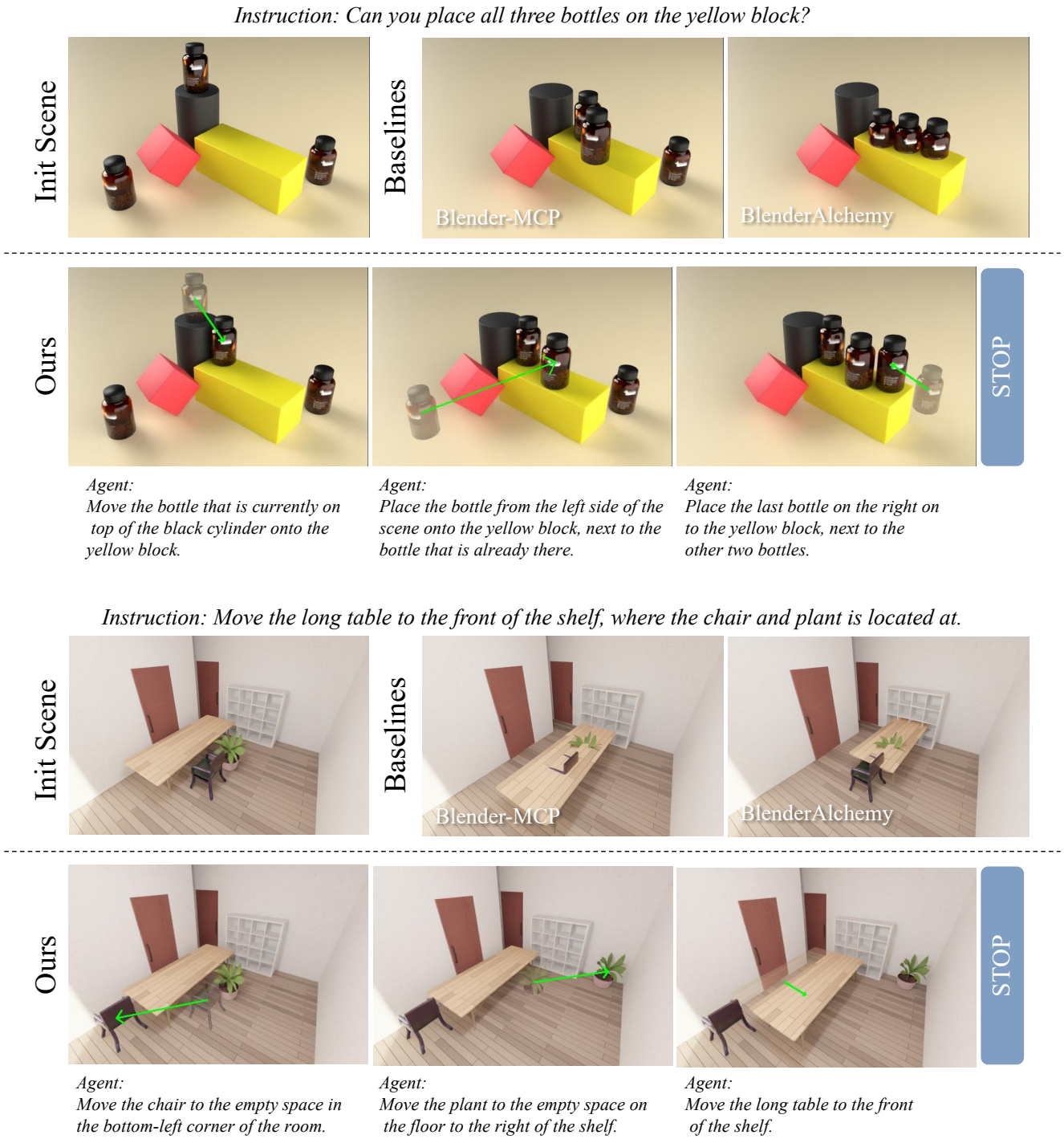


Figure 7. **Comparison on General Instruction Examples.** The input instruction is shown at the top. We show the initial scene, baseline results, and our model's step-by-step outputs. The plans generated by our model are listed below the edited images.

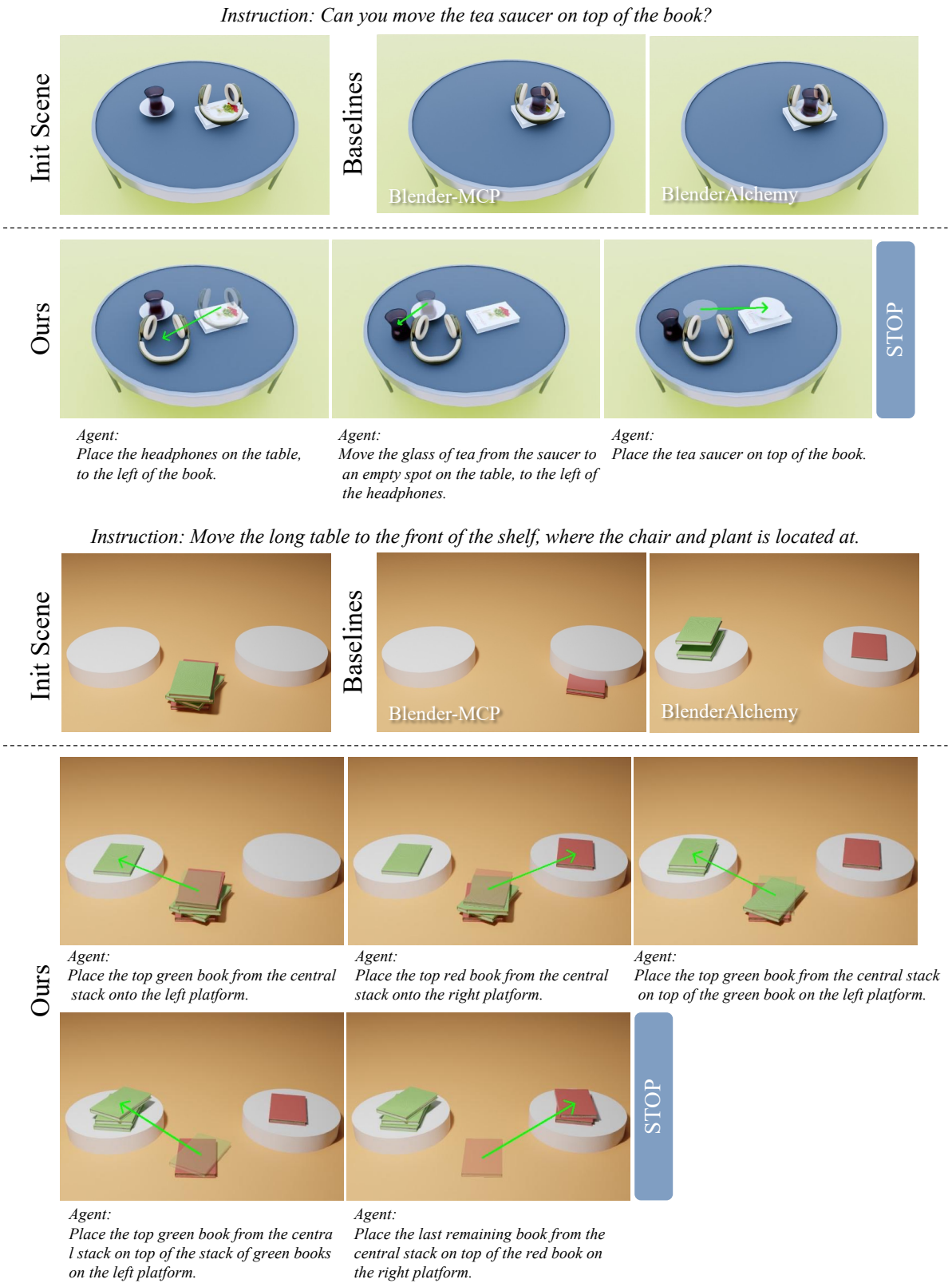


Figure 8. **Comparison on General Instruction Examples (cont'd).**

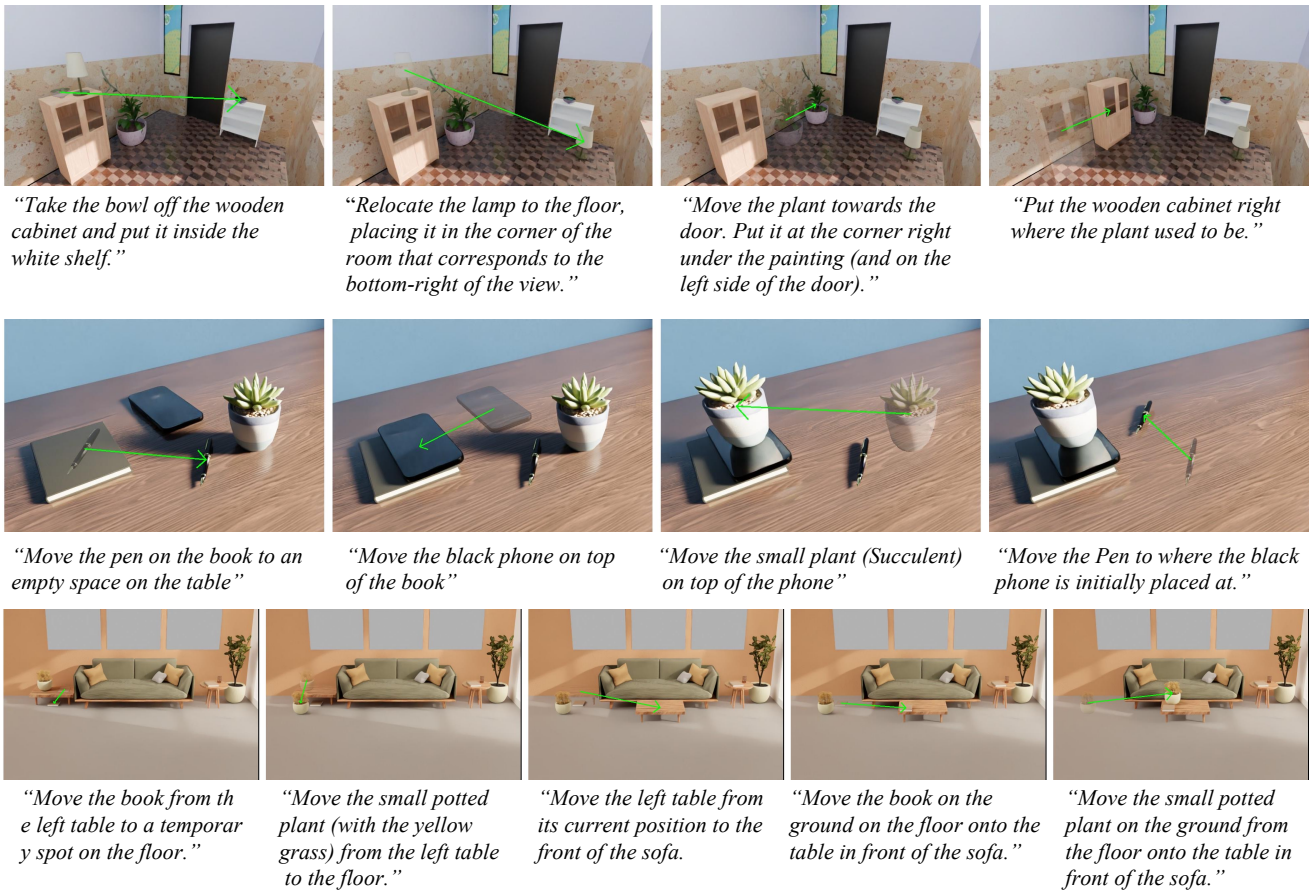


Figure 9. **Additional Results on Per-step Instruction Examples.** The input instructions are shown below the images.

Instruction: Switch the position of the small round sofa and the wooden table.

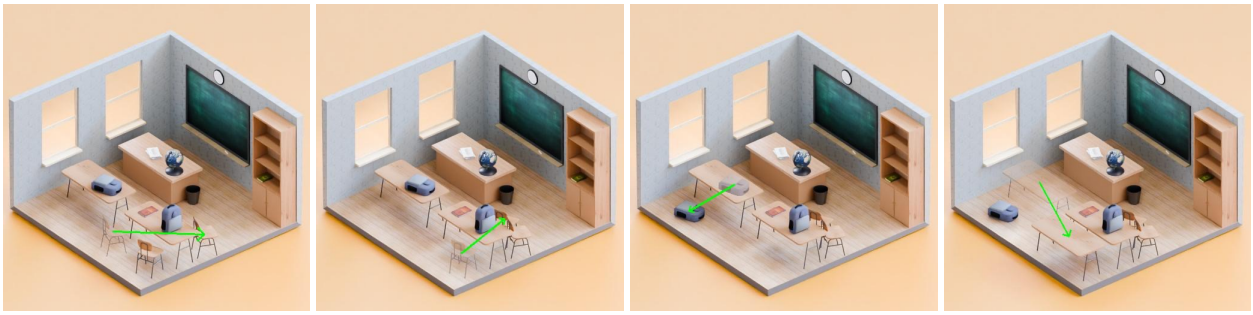
Agent:
Move the vase from the table and place it on the floor to the left of the tables.

Agent:
Move the small round sofa to a temporary position in the empty space on the floor in the bottom-left area.

Agent:
Move the larger of the two wooden tables to the empty space in the bottom right corner of the room, where the small round sofa was originally located.

Agent:
Move the small round sofa to the now-empty spot where the larger wooden table was originally located, next to the smaller wooden table.

Instruction: Move the two chair to the empty space in front of the shelf, then move the table on the upper left (the one without the book on it) to where the two chairs are originally located at.



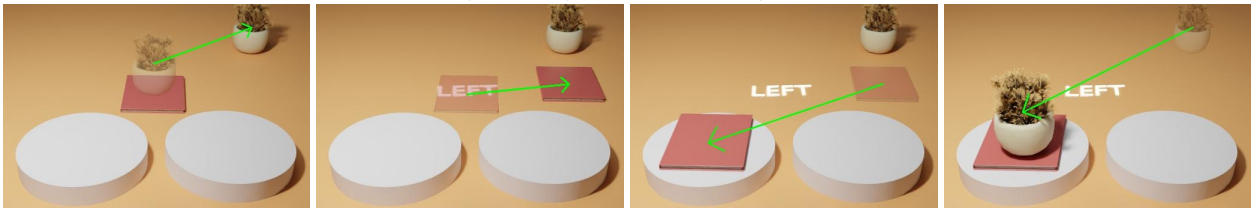
Agent:
Move the leftmost chair to the empty space on the right, in front of the bookshelf.

Agent:
Move the remaining chair to the empty space in front of the shelf, placing it next to the chair that was already moved.

Agent:
Move the printer from the desk and place it on the empty floor space to the left of the desk.

Agent:
Move the empty desk on the left to the empty space behind the other desk, centering it in the area where the two chairs were originally located.

Instruction: Move the book and the plant to elsewhere to reveal the text below, then place the two objects to the cylinder as the text on the floor instructed.



Agent:
Move the plant to a temporary empty space on the floor.

Agent:
Move the red square object to an empty space in the top-right area of the floor to reveal the text underneath it.

Agent:
Place the red square object on top of the left cylinder.

Agent:
Place the plant on top of the red square.

Figure 10. **Additional Results on Per-Step Instruction Examples.** The input instructions are shown above the images, while the plans generated by the *Planner* agent are shown below.

```
def ray_probe_for_plane(coord_list):
"""
Casts rays from specified camera-view coordinates and reports the first object intersected by
each ray. It also extract the plane from the hitting point.

Args:
- coord_list: A list of normalized 2D coordinates (x, y) (from 0.0 to 1.0) from which to cast
  rays. Example: [[0.25, 0.25], [0.75, 0.75]]

Returns:
- hit_info_list: A list of hit details, one for each input coordinate. Each item in the list
  will be: A tuple containing (hit_object_name, hit_location_3d, hit_normal_vector) if an
  object was intersected. None if the ray did not intersect with any object.

"""

def render_with_highlighted_object(output_path, object_name_list, object_color_list=None):
"""
Renders an image with specified objects highlighted by semi-transparent colored masks. This is
useful for visually identifying objects in the rendered output.

Args:
- output_path: The file path to save the rendered image.
- object_name_list: A list of up to 10 object names to highlight. Example: ["Cube", "Sphere"].

Returns:
- image: The rendered Image object.
- object_color_dict: A dictionary mapping each object name to its corresponding highlight color
  in the image. Please note that the highlight color does not indicate the original color of
  the object.

"""

def list_all_object_in_view(interested_area="[0,0,1,1]"):
"""
Lists all objects visible within a specified rectangular area of the camera's view.

Args:
- interested_area: A list of four normalized coordinates x1,y1,x2,y2 (from 0.0 to 1.0) defining
  the rectangular area to check. Example: [0.25, 0.25, 0.75, 0.75]

Returns:
- object_list: A list of names of the objects found within the specified area.

"""
```

Figure 11. **Description of Visual API functions.** Note that function names may differ from those in the main text.

[[Task Description]]
You are an AI 3D object arrangement planner to solve a multi-step arrangement task.
Given a user's instruction and a history of edit images from your predecessors,
plan the (current) arrangement step.

Inputs:
1. General Instructions: The instruction from the user to define the task.
1. Previous Edits: Description of edits made by predecessors.
2. Image History: A sequence of images: [initial scene, result_of_step_1, ..., result_of_latest_step, current scene]. Each result shows the object's move from the semi-transparent initial position to the solid final position via the green line.
3. Maximum Allowed Steps: The maximum number of steps allowed for the task.

Note:
* All directions (e.g., "left," "top") are relative to the 2D image.
* All 2D coordinates are in normalized (x, y) format.
* Make your answer as concise as possible.
* Only one object can be operated on per step. Avoid creating plans that arrange multiple objects.
* After each arrangement, no object in the scene should be colliding with others or floating (except the already floating ones).
* To move an object with other objects on it, you should first remove the stacked objects to prevent them floating after the movement.

Your Tasks:
1. Summarize History:
Briefly describe the initial scene and all previous edits based on the History Images.

2. Predict Future Steps:
Plan the future arrangement (placement and rotation) steps based on the complete image history and the user's general instruction for global planning.
If you believe the goal is already achieved, output a <finished> token and skip all remaining steps.
If you believe the goal is impossible to reach within the maximum allowed steps (including those already taken), output an <impossible> token and skip all remaining steps.
Otherwise, briefly describe your future plan.

3. Updated Instruction:
Provide an [updated_instruction] block. Output your local instruction for the current step, It should reference only the latest_scene (e.g. remove all "initial position" prompts). Add details in the instruction so it is unambiguous.
Make sure all metric information (e.g., 0.5m, 20cm), object name information (e.g. Chair_000), and orientation information (e.g. facing towards another object) in the original instruction is retained and unchanged.

Figure 12. **Prompt for the *Planner* with General Instructions (Part 1).**

4. Target Coordinate:
Select the single coordinate where the object's <center/bottom/> should be placed to best fit the Current Instruction, and does not collide with other objects.
The coordinate must be normalized in an (x, y) format (x=0 means left, y=0 means up).
Output the coordinate information in a [coordinate] block.

Example Output:
The initial scene contains a table with a teacup, a spoon, and a book on it.
There're two previous arrangements in total.
arrangement 1: The teacup was moved on top of the book.
arrangement 2: The spoon was moved inside of the teacup.

(If the goal is already achieved)
The goal is already achived. <finished>

(If the goal is not achieved and require more steps)
The goal is to place the pen to the apple's original location, and facing to the banana. From the current state, this requires a 2-step plan:
1. move the apple to an empty space
2. Place the pen to apple's original position (from image 1, $\approx(0.52, 0.56)$), and facing to the banana.

(If allowed steps are sufficient)
The maximum allowed steps is 5. Since this plan only requires 2 more steps (4 steps in total), it is valid.

[updated_instruction]
Place the pen to the the table, and facing the banana.
[end_of_updated_instruction]

[coordinate]
The center of the object should be around [0.52, 0.56].
[end_of_coordinate]

(If allowed steps are insufficient)
The maximum allowed steps is 3. Since this plan requires 2 more steps (4 steps in total), it is impossible to reach the goal.
<impossible>

(the input instructions and images)

Figure 13. **Prompt for the *Planner* with General Instructions (Part 2).**

[[Task Description]]
You are an AI 3D object placement planner.
Given a list of instructions and a history of scene images,
plan the (current) placement step.

Inputs:
1. Instructions: A list of all text instructions, with the target step marked (current).
2. Image History: A sequence of images: [initial scene, result_of_step_1, ..., result_of_latest_step, current scene].
Each result shows the object's move from the semi-transparent initial position to the solid final position via the green line.

Note:
* All directions (e.g., "left," "top") are relative to the 2D image.
* All 2D coordinates are in normalized (x, y) format.
* Make your answer as concise as possible.

Your Tasks:
1. Summarize History:
Briefly describe the initial scene and all previous edits based on the History Images and Previous Instructions.

2. Reasoning:
Plan your placement for the (current) instruction with both the all historic images and future instructions for global planning.

3. Updated Instruction:
Provide an [updated_instruction] block. Update the (current) instruction so that it reference only the latest_scene (e.g. remove all "initial position" prompts). Add details in the updated instruction so it is unambiguous. Make sure all metric information (e.g., 0.5m, 20cm) and object name information (e.g. Chair_000) in the original instruction is retained and unchanged.

4. Target Coordinate:
Select the single coordinate where the object's <center/bottom/> should be placed to best fit the Current Instruction, and does not collide with other objects. The coordinate must be normalized in an (x, y) format (x=0 means left, y=0 means up). Output the coordinate information in a [coordinate] block.

Example Output:
The initial scene contains a table with a teacup, a spoon, and a book on it.
Placement 1: The teacup was moved on top of the book.
Placement 2: The spoon was moved inside of the teacup.

Reasoning: Current instruction targets the pen's original location (from image 1, $\approx(0.52, 0.56)$).
A future instruction ("place a banana next to the apple") requires leaving space.
I will place the apple at the original pen's spot, ensuring room for the banana.

[updated_instruction]
Place the apple on the table to the left of the teacup, and leave a small gap for a banana to be placed next to it.
[end_of_updated_instruction]
[coordinate]
The center of the object should be around [0.52, 0.56].
[end_of_coordinate]

(the input instructions and images)

Figure 14. **Prompt for the *Planner* with Per-step Instructions.**

[[Task Description]]
As a Virtual Set Dresser AI, your task is to plausibly place a specified object within a 3D Blender scene based on a user's prompt.

[Inputs]
1. Target Prompt: The objective for the scene (e.g., "Place the vase on the desk").
2. Initial Image: The view of the initial scene.
3. Blender Tools: A suite of functions for scene interaction.

[General Principles]
* Strictly follow the steps, do not skip any of them unless it's explicitly mentioned in the prompt.
* Please be efficient with your function calls. Consolidate function usage where possible.
* Stop immediately after providing the final confirmation message. DO NOT evaluate your own result.
* Log your full thoughts, the reasoning for every tool call.

[Task Specific Principles]
* All directions are relative to the image. For example, 'left' means the left side of the image as you see it.
* DO NOT make assumptions about the Blender's coordinate system (e.g. X-axis means left/right; Z=0 means the floor plane).
* All 2d coordinates (e.g. [0.43, 0.62]) are in the (x, y) format.
* DO NOT rely on the object's full name to determine its property, instead exam more on the image visually.
* DO NOT use object's name as planes in the constraint parameters. Always use bounding box planes or created planes.
* DO NOT use type of constraint that not supported by the description list.

[Constraint Description]
The following constraints are supported:
* ObjectName("<name_of_object>"): The target object. Make sure it is called only once.
* Contact("down/side/up", "<name_of_surface>"): The bottom/side/top of the target object contacts the surface.
* NoOverhang("down", sur, mode): 'full_only' means the object's bottom is completely within the surface, while 'center' means the center of the bottom is on the surface.
* CloseToPix("center"/"down", [x, y]): The object's center/bottom is close to the 2D pixel coordinate in the camera view.
* Distance("<name_of_other_object>", <distance>): The distance (in meters) between the center of the target object and another object. Use it when the instruction contains distance information.
* FaceTo("<name_of_object>/<name_of_surface>/camera"): The object should face to the center of an object / the normal of the surface / camera.
* BackTo("<name_of_object>/<name_of_surface>/camera"): The object should back to the center of an object / the normal of the surface / camera.
* Rotate(90/180/270): Rotate the object by 90/180/270 degrees. This constraint will be overridden by FaceTo/BackTo.

There're two types of plane names allowed:
* <object_name>_<face_id>: Planes created by ray_probe_for_plane calls.
* <object_name>_<up>: The top surface of the object. Use this plane for stacking objects on top of each other.

Figure 15. **Prompt for the *Executor* (Part 1).**

[Workflow]
IMPORTANT: You must finish the task by following these steps in strict order.

Step 1: Visual Scene Analysis
Analyze the prompt and initial image to identify the target object, its destination, and key reference objects.
* Hint 1: Use the ray_probe tool on likely image coordinates to determine precise 3D coordinates and identify reference objects. You can probe multiple points for comprehensive data.
* Hint 2: Use list_all_object_in_view on interested area to retrieve relevant objects for placement in a certain area.
* Hint 3: (Only when the previous hints are not sufficient) use render_with_highlighted_object to visually identify and confirm relevant objects for placement. DO NOT use this function when the input instruction contains color information (e.g. red book).

Step 2: Plane Selection
Find the plane that the object should be placed on (or hanged to). Use the ray_probe_for_plane tool on likely image coordinates to extract the plane's name located at the coordinate.
In addition to the suggested coordinates, you are encouraged to explore other pixels to find the right plane to place the object. Please carefully look at the image and find best candidates for plane selection.

Step 3: Design constraints for the placement solver
Based on your visual analysis, and the description of the constraint list, design a set of constraints that helps the solver to solve the right position for placement. Output the constraints in the [constraints] block and end with a <completion> token. For NoOverHang constraints, use "full_only" for large surfaces (table, floor) and "center" for small surfaces (book, block).

Constraint List Requirements:
* Strictly follow the constraint syntax from the description.
* Must include at least one CloseToPix constraint.
* All included plane names must be valid (from Step 2 or the object's bounding box).

Example Output:
[constraints]
["ObjectName", "<Name_of_the_moved_object>"],
["CloseToPix", "center", [0.5, 0.5]],
["Contact", "down", "<Name_of_surface_from_step_2>"],
["NoOverhang", "down", "<Name_of_surface_from_step_2>", "full_only"],
<rotation_constraint_example>
[end of constraints]
<completion>

[[End of Task Description]]

(the input instructions and images)

Figure 16. **Prompt for the *Executor* (Part 2).**

[[Task Description]]

You are a Virtual Set Dresser AI. Your task is to evaluate the plausibility and correctness of an object's final placement.

You are given two inputs:
1. Target Prompt: The text instruction for the placement. All directions described in the text are relative to the image.
2. Edited Image: A render showing the object's move from the semi-transparent initial position to the solid final position via the green line.

Please strictly evaluate the final placement in the edited image against the target prompt. Check carefully for any physical errors (floating, clipping) or positional inaccuracies. Provide a verdict (<excellent|good|fair|bad|terrible>) with a clear justification in an [evaluation] block,

* Example:
[evaluation]
visual_exam: The agent correctly identified the support surface but failed on two counts: the object is visibly floating, and its position does not match the prompt's request to be in the corner.
verdict: fair
[end of evaluation]

[[Evaluation Rubric]]
Excellent: Perfect placement. Physically plausible (no errors) AND location exactly matches the prompt.
Good: Physically perfect placement, but the location is only an approximation of the prompt (e.g., near the corner, not on it).
Fair: Has one clear flaw:
A) Correct location, but minor physics error (slight floating/clipping).
B) Physically plausible, but wrong location on the correct surface.
Bad: Has multiple flaws OR one severe flaw:
A) Wrong location and a physics error.
B) Placed on the wrong support surface (e.g., floor instead of table).
Terrible: Physically impossible (e.g., floating in mid-air) or shows a complete misunderstanding of the prompt.

(the input instruction and images)

Figure 17. **Prompt for the *Evaluator*.**

Your task is to evaluate a sequence of AI-generated object placements based on their realism and consistency with instructions.

Here's the instruction for the previous agent (each line corresponds to one step):
(user's instruction)

The input image showing the render of the initial scene and all edited scenes for each step (from left to right).
(All input images)

Please STRICTLY evaluate each result image (starting from `"0"`) on two separate criteria using the scale: `<excellent|good|fair|bad|terrible>`.

1. **Plausibility:** How physically realistic is the placement?
2. **Correctness:** How well does the placement match the agent's instruction?

Evaluation Rubric

* **Excellent:** Perfect placement. Physically plausible AND location exactly matches the prompt.
* **Good:** Physically perfect placement, but the location is only an approximation of the prompt.
* **Fair:** Has one clear flaw:
* A) Correct location, but minor physics error (slight floating/clipping).
* B) Physically plausible, but wrong location on the correct surface.
* **Bad:** Has multiple flaws OR one severe flaw:
* A) Wrong location and a physics error.
* B) Placed on the wrong support surface (e.g., floor instead of table).
* **Terrible:** Physically impossible (e.g., floating in mid-air) or shows a complete misunderstanding of the prompt.

Output Format

Provide your ratings and justifications as a single JSON object. **DO NOT** output anything other than the JSON object.

Example:

```json
{
"0": {
"correctness": "fair",
"plausibility": "bad",
"justification": "Placed on the correct table but in the wrong spot, and the object is clearly clipping into the wall."
},
"1": {
"correctness": "excellent",
"plausibility": "excellent",
"justification": "Perfect placement, exactly matching the instruction."
}
}
```

Figure 18. **Prompt for the Benchmark Judges.**

References

[1] Blender. https://www.blender.org/, . 1

[2] Blender-mcp. https://github.com/ahujasid/blender-mcp?tab=readme-ov-file, . 1

[3] Ian Huang, Guandao Yang, and Leonidas Guibas. Blenderalchemy: Editing 3d graphics with vision-language models. In *European Conference on Computer Vision*, pages 297–314. Springer, 2024. 1

[4] Ian Huang, Yanan Bao, Karen Truong, Howard Zhou, Cordelia Schmid, Leonidas Guibas, and Alireza Fathi. Fireplace: Geometric refinements of llm common sense reasoning for 3d object placement. *arXiv preprint arXiv:2503.04919*, 2025. 3