HOG-Layout: Hierarchical 3D Scene Generation, Optimization and Editing via Vision-Language Models

Supplementary Material

A. Details of the Experiment Setup

In the comparative experiment and ablation study, for all VLM and LLM usage, we use GPT-4o-2024-08-06 with the default parameters. The experiment is conducted on a single computer with the following specifications:

- **OS**: Windows 11
- **CPU**: 13th Gen Intel Core i5-13500H
- **GPU**: NVIDIA GeForce RTX 4060 (8GB)
- **Memory**: 64 GB

B. Details of HOG-Layout

B.1. Optimization Algorithm Details

We model layout optimization as a physics simulation (Algorithm 1). To maintain notational clarity, we denote the force calculation functions in the pseudocode using capitalized abbreviations (e.g., $COLH$ for Horizontal Collision), which directly correspond to the constraint details described below. In each iteration t, we calculate specific force components corresponding to different constraints:

- **Physical Constraints**:
  - **Boundary ($F^{\mathbf{h}}_{\mathbf{bnd}}, F^{\mathbf{v}}_{\mathbf{bnd}}$)**: Computed by projecting the object onto the room's axes (X, Y, Z). If the projection interval lies outside the room boundaries defined by $\mathcal{R}$, a restoring force proportional to the penetration depth is applied to push the object back.
  - **Support ($F_{\mathbf{sup}}$)**: Ensures stability by checking the contact surface area between the object and its parent. We calculate the intersection polygon of their footprints; if the ratio of the intersection area to the object's area is below a threshold, a centripetal force is applied to pull the object towards the parent's geometric center.
  - **Horizontal Collision ($F^{\mathbf{h}}_{\mathbf{col}}$)**: $F^{\text{h}}_{\text{col}}$ resolves collisions between same-level objects. The HOG-Layout framework supports two distinct approaches for resolving overlaps, both of which are viable depending on the precision requirements:
    1. *Area-based Method*: A simpler approach where the force magnitude is proportional to the overlapping area of the bounding boxes, and the direction is determined by the vector connecting the centroids.
    2. *SAT-based Method*: A more precise approach (employed in our final implementation) using the *Separating Axis Theorem (SAT)*. It calculates the *Minimum Translation Vector (MTV)* required to separate polygons, providing exact direction and magnitude for the collision force.

Algorithm 1 Hierarchical Force-Directed Optimization

Input: Objects $\mathcal{S} = \{O_1, \ldots, O_N\}$, Room $\mathcal{R}$, Params Θ
Output: Optimized $\mathcal{S}_{\text{opt}}$

$t \leftarrow 0, \quad converged \leftarrow$ False
while $t < T_{\max} \wedge \neg converged$ **do**
  Init: $F_{i,\text{plane}} \leftarrow \mathbf{0}, F_{i,\text{vert}} \leftarrow 0, \tau_i \leftarrow 0, \forall O_i \in \mathcal{S}$
  $active \leftarrow$ False
  for all $O_i \in \mathcal{S}$ **do**
    {1. Calculate Different Constraint Forces}
    $F^{\text{h}}_{\text{bnd}} \leftarrow \textsc{BndH}(O_i, \mathcal{R})$
    $F^{\text{v}}_{\text{bnd}} \leftarrow \textsc{BndV}(O_i, \mathcal{R})$
    $F_{\text{sup}} \leftarrow \textsc{Sup}(O_i, O_{\text{parent}})$
    $F^{\text{h}}_{\text{col}} \leftarrow \textsc{ColH}(O_i, \mathcal{S})$
    $F^{\text{v}}_{\text{col}} \leftarrow \textsc{ColV}(O_i, \mathcal{S})$
    $F_{\text{adj}} \leftarrow \textsc{Adj}(O_i, O_{\text{target}})$
    $F_{\text{wall}} \leftarrow \textsc{Wall}(O_i, \mathcal{R})$
    $\tau_{\text{align}} \leftarrow \textsc{Align}(O_i, O_{\text{target}})$
    $\tau_{\text{pnt}} \leftarrow \textsc{Point}(O_i, O_{\text{target}})$
    {2. Accumulate All the Forces}
    $F_{i,\text{plane}} \leftarrow F_{i,\text{plane}} + w_{\text{bnd}} F^{\text{h}}_{\text{bnd}} + w_{\text{col}} F^{\text{h}}_{\text{col}} + w_{\text{sup}} F_{\text{sup}} + w_{\text{adj}} F_{\text{adj}} + w_{\text{wall}} F_{\text{wall}}$
    $F_{i,\text{vert}} \leftarrow F_{i,\text{vert}} + w_{\text{bnd}} F^{\text{v}}_{\text{bnd}} + w_{\text{vcol}} F^{\text{v}}_{\text{col}}$
    $\tau_i \leftarrow \tau_i + w_{\text{align}} \tau_{\text{align}} + w_{\text{pnt}} \tau_{\text{pnt}}$
    {3. Check Convergence}
    $F_{\text{total}} \leftarrow \|F_{i,\text{plane}}\| + |F_{i,\text{vert}}| + |\tau_i|$
    if $F_{\text{total}} > \epsilon$ **then**
      $active \leftarrow$ True
    end if
  end for
  {4. Deadlock & Update}
  $Deadlock \leftarrow \textsc{HandleDeadlocks}(\mathcal{S}, \Theta)$
  for all $O_i \in \mathcal{S}$ **do**
    $p^{(t+1)}_{i,\text{plane}} \leftarrow p^{(t)}_{i,\text{plane}} + \eta_{\text{trans}} \cdot F_{i,\text{plane}}$
    $p^{(t+1)}_{i,\text{vert}} \leftarrow p^{(t)}_{i,\text{vert}} + \eta_{\text{vert}} \cdot F_{i,\text{vert}}$
    $\theta^{(t+1)}_i \leftarrow \theta^{(t)}_i + \eta_{\text{rot}} \cdot \tau_i$
  end for
  if $\neg active \wedge \neg Deadlock$ **then**
    $converged \leftarrow$ True
  end if
  $t \leftarrow t + 1$
end while
return $\mathcal{S}$

  - **Vertical Collision ($F^{\mathbf{v}}_{\mathbf{col}}$)**: It is used for objects at different hierarchy levels (e.g., wall-mounted pictures vs.

Algorithm 2 Deadlock Detection and Evasion

Input: Object O_i, History $\mathcal{H}_i$, Contributions Φ_i, Params Θ
Output: Boolean $is\_deadlocked$

$is\_deadlocked \leftarrow$ False
{1. Check Active Evasion Timer}
if $O_i.timer > 0$ **then**
  $F_{i,\text{plane}} \leftarrow F_{i,\text{plane}} + F_{i,\text{evade}}$
  $O_i.timer \leftarrow O_i.timer - 1$
  return True
end if
{2. Horizontal Deadlock}
if $\sum \mathcal{H}_i[t-W:t] < \epsilon_{\text{disp}} \wedge |\Phi_i| \geq 2$ **then**
  Select pair $(\mathbf{v}_a, \mathbf{v}_b) \in \Phi_i$ with **Max Mag Sum** s.t. $\angle(\mathbf{v}_a, \mathbf{v}_b) \approx 180°$
  if such pair exists **then**
    $\mathbf{v}_{\text{axis}} \leftarrow \frac{\mathbf{v}_a}{\|\mathbf{v}_a\|}$ ▷ Normalized Axis
    $\mathbf{v}_\perp \leftarrow (-\mathbf{v}_{\text{axis},y}, \mathbf{v}_{\text{axis},x})$ ▷ Orthogonal
    $F_{i,\text{evade}} \leftarrow \lambda_{\text{evade}} \cdot \mathbf{v}_\perp$
    $O_i.timer \leftarrow T_{\text{deadlock}}$
    $F_{i,\text{plane}} \leftarrow F_{i,\text{plane}} + F_{i,\text{evade}}$
    $is\_deadlocked \leftarrow$ True
  end if
end if
{3. Vertical Deadlock}
if Vertical displacement small $\wedge F^{\text{up}}_{\text{vert}} > 0, F^{\text{down}}_{\text{vert}} < 0$ **then**
  Calc available gap h_{gap}
  if $h_{\text{gap}} < h_i$ **then**
    $s_i \leftarrow s_i \cdot (h_{\text{gap}}/h_i)$ ▷ Adjust Scale
    $is\_deadlocked \leftarrow$ True
  end if
end if
return $is\_deadlocked$

objects on floor). It is heuristic: for objects between different levels, we first check if the 2D bounding boxes overlap. If they do, we calculate the overlap on the Z-axis. Repulsive forces are then applied strictly along the Z-axis—pushing the higher object upwards and the lower object downwards—to separate them.

- **Semantic Constraints**:
  - **Adjacent** (F_{adj}):This is modeled as a damped spring system. We compute the Euclidean distance between the nearest points of two objects. The force magnitude is proportional to the difference $(d_{current} - d_{target})$, acting along the vector connecting these points to either attract or repel the object.
  - **Against Wall** (F_{wall}): It identifies the nearest wall specified by the instruction (e.g., "left", "back"). A linear force penalizes the perpendicular distance from the object's bounding box edge to the wall plane if it exceeds a tolerance threshold.

Table 1. Optimized Parameters for the Hierarchical Force-Directed Optimizer.

Symbol	Description	Value
w_{col}	Collision weight (Horizontal)	1.134
w_{vcol}	Collision weight (Vertical)	2.850
w_{bnd}	Boundary weight	2.857
w_{sup}	Support weight	0.525
w_{adj}	Adjacent weight	0.833
w_{wall}	Against-wall weight	2.977
w_{pnt}	Point-towards weight	4.688
w_{align}	Alignment weight	5.037
λ_{evade}	Deadlock evasion strength	0.161
T_{deadlock}	Deadlock evasion duration (steps)	17
η_{trans}	Translation step size	0.208
η_{rot}	Rotation step size (degrees/step)	8.569
$T_{\max}$	Maximum iterations	300

  - **Alignment & Pointing** ($\tau_{\text{align}}, \tau_{\text{pnt}}$): Unlike linear forces, these generate rotational torque. We calculate the shortest angular difference $\Delta\theta$ between the object's current yaw and the target vector (derived from a fixed angle or the relative position of another object). The torque is proportional to $\Delta\theta$ to iteratively correct orientation.

These forces are aggregated into planar ($F_{i,\text{plane}}$) and vertical ($F_{i,\text{vert}}$) totals, and the system state is updated via Explicit Euler integration.

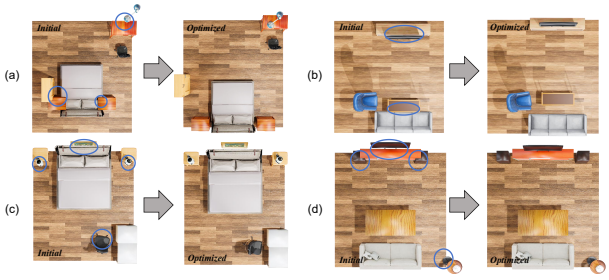


Figure 1. Optimization examples.

Gradient-free force-directed methods are prone to local minima. To robustly detect deadlocks, we maintain two state variables for each object: a displacement history window $\mathcal{H}_i$ (tracking recent movement magnitudes) and a set of active force contribution vectors Φ_i. Algorithm 2 addresses this via two mechanisms:

- **Horizontal Deadlock**: A horizontal deadlock is flagged when the cumulative displacement over window W (iteration 20 times) is negligible ($\sum \mathcal{H}_i < \epsilon_{\text{disp}}$), yet the object is subject to multiple competing forces ($|\Phi_i| \geq 2$). If these conditions are met, the algorithm searches Φ_i for a

pair of forces $(\mathbf{v}_a, \mathbf{v}_b)$ that are structurally opposing (angle $\approx 180°$) and have the maximum combined magnitude. It then applies a temporary orthogonal evasion force ($\mathbf{v}_\perp$) scaled by λ_{evade} for a duration T_{deadlock} to break the symmetry.

- **Vertical Deadlock**: Addresses squeezing where an object cannot fit vertically. If the available vertical gap h_{gap} is smaller than the object height h_i, the object scale s_i is adjusted.

B.2. Implementation Details & Parameters

To ensure stability and convergence of the physics-based simulation, appropriate simulation parameters (Θ) are critical. We **treat them as hyperparameters** for the purpose of optimization and employ automated tuning techniques to find their optimal values.

- **Necessity of Hyper-parameter Search:** Our iterative optimization process mimics gradient descent, where the various "forces" act as gradients, and their corresponding weights function similarly to learning rates. If the weight values are too small, objects move extremely slowly (analogous to vanishing gradients), failing to reach a stable state within the maximum iteration limit. Conversely, if the weights are too large, objects experience severe positional oscillations in each step (analogous to exploding gradients) and can never settle into a balanced position. Therefore, a hyper-parameter search is essential to find a reasonable set of weights that allows the scene layout to converge more quickly and smoothly to a stable state.
- **Search Objective and Dataset:** The objective of this search is to minimize a composite penalty. We optimize the parameters on a custom dataset consisting of 50 diverse initial scene layouts generated by our pipeline with GPT-4o (distinct from the SceneEval benchmark). For each scene, we calculate the sum of constraint violations and the residual force magnitudes (which indicate the severity of the violations) at the end of the optimization. The search aims to minimize the average of this composite score across all scenes.
- **Generality:** Although the hyper-parameters were not tuned on the SceneEval scenes, they perform effectively on the SceneEval benchmark, demonstrating their strong generalizability. Furthermore, the hyper-parameters only need to fall within a reasonable range to ensure convergence; the ultimate quality of the layout relies more heavily on the reasoning capabilities of the VLM and the core design of the hierarchical algorithm.
- **Stability Across Random Seeds:** We utilize the *Optuna* framework[1] with a Tree-structured Parzen Estimator (TPE[2]) sampler to conduct the search. When repeating the search process with multiple different random seeds, although the internal exploration paths may vary, the final optimal parameters consistently converge to a similar, stable range, proving the robustness of our searched weights.

To enhance search efficiency, we employ the Hyperband pruner[3] to terminate unpromising trials based on intermediate performance across scenes. The tuning process runs for 32 hours. The final optimized parameters used in our experiments are listed in Tab. 1.

B.3. Object Retrieval Implementation Details

3D Asset Database. Our system utilizes the 3D asset library created by Holodeck [4], which consists of over 51,000 diverse high-quality assets sourced from Objaverse [5]. These assets are specifically curated for indoor scenes and augmented with detailed metadata—including textual descriptions, bounding box dimensions, and canonical views—generated by GPT-4-Vision. Furthermore, the assets undergo optimization processes such as mesh reduction and collider generation to ensure efficient rendering and interaction within embodied AI simulators.

Retrieval Algorithm. To retrieve the most suitable 3D asset O_i from the database given a textual query T_q and a target geometric bounding box $B_{target} \in \mathbb{R}^3$, we employ a multi-stage scoring mechanism. The final relevance score $Score_{Final}(i)$ is a weighted sum of semantic, visual, and geometric similarities:

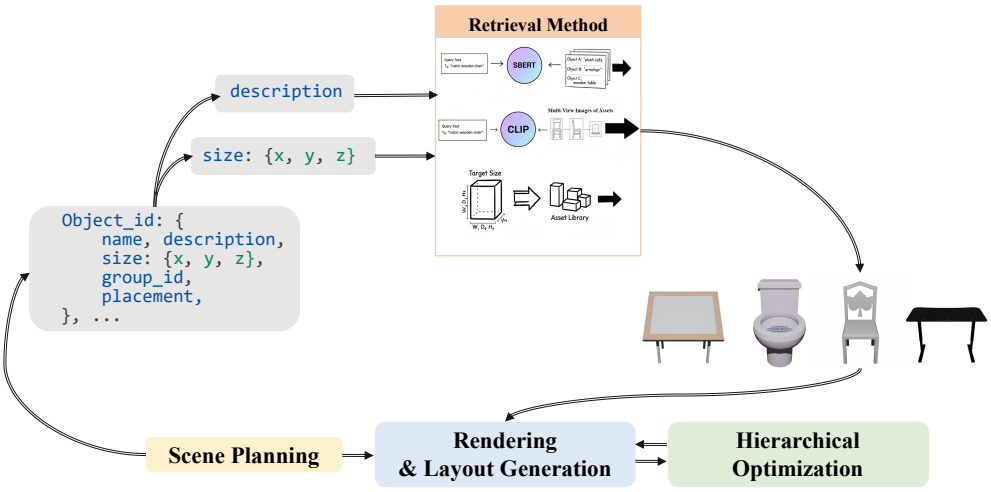
$$\begin{aligned} Score_{Final}(i) = w_1 \cdot S_{sbert}(i) + w_2 \cdot S_{clip}(i) \\ + w_3 \cdot S_{size}(i) \end{aligned} \tag{1}$$

where $S_{sbert}(i)$ denotes the semantic similarity derived from SBERT embeddings (converted from L_2 distance in a coarse retrieval stage), and $S_{clip}(i)$ represents the visual similarity calculated via OpenCLIP. Specifically, $S_{clip}(i) = \max_{v \in \mathcal{V}_i}(\text{sim}(f_{txt}(T_q), f_{img}(v)))$, where $\mathcal{V}_i$ is the set of multi-view renderings of object i, taking the maximum similarity to handle viewpoint variations.

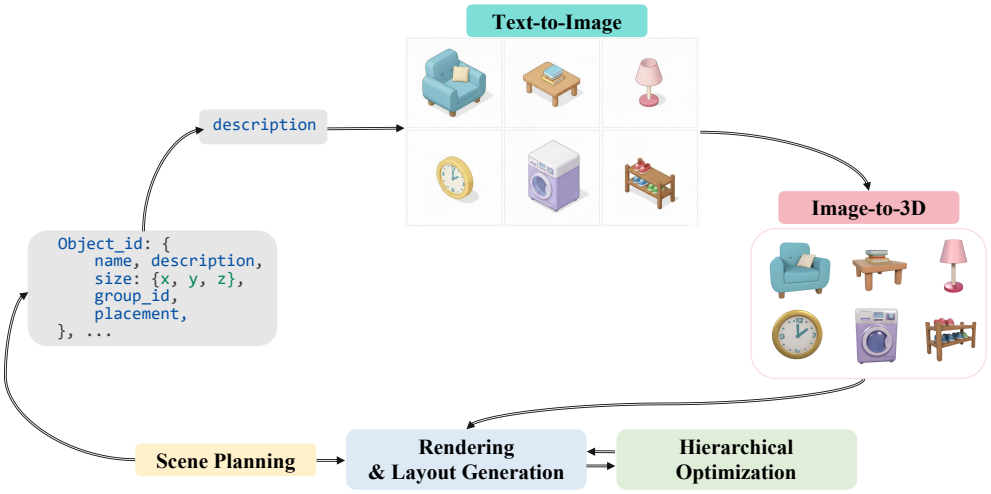
Geometric Alignment (S_{size}). To ensure the retrieved asset fits the planned layout not just in scale but in aspect ratio, we introduce a shape-aware geometric score. Direct comparison of dimensions is sensitive to absolute scale; therefore, we normalize the bounding box dimensions to focus on the object's proportions.

Let the target dimensions be $R_{target} = (w_t, d_t, h_t)$ and the candidate asset dimensions be $R^{(i)}_{asset} = (w_a, d_a, h_a)$. We first normalize these vectors by their respective maximum dimension to obtain scale-invariant representations $\hat{R}$:

$$\hat{R}_{target} = \frac{R_{target}}{\|R_{target}\|_\infty}, \quad \hat{R}^{(i)}_{asset} = \frac{R^{(i)}_{asset}}{\|R^{(i)}_{asset}\|_\infty} \tag{2}$$



(a) Pipeline with retrieved objects



(b) Pipeline with generative 3D objects

Figure 2. **Optional object acquisition methods.** (a) The retrieval-based pipeline matches query text and geometry against a fixed database using semantic and visual similarity. (b) The generative pipeline replaces retrieval with generative objects: generating an image from text (e.g., via DALL-E) and then converting it to a 3D model (e.g., via Hunyuan 3D).

where $\|\cdot\|_\infty$ denotes the infinity norm (maximum component). We then compute the geometric discrepancy D_{size} as the mean absolute difference between the normalized vectors:

$$D_{size}(i) = \frac{1}{3} \sum_{k \in \{w,d,h\}} \left| \hat{R}^{(k)}_{target} - \hat{R}^{(i,k)}_{asset} \right| \quad (3)$$

Finally, this discrepancy is converted into a similarity score $S_{size} \in (0, 1]$ using an exponential decay function with a sensitivity hyperparameter k:

$$S_{size}(i) = \exp(-k \cdot D_{size}(i)) \quad (4)$$

This formulation ensures that objects with similar aspect ratios to the planned placeholder receive significantly higher scores, promoting physical plausibility in the generated scene.

Parameter Settings. Based on our empirical validation, we set the retrieval weights to $w_1 = 5$, $w_2 = 85$, and $w_3 = 10$ in the final re-ranking stage, as the coarse stage already filters semantically irrelevant candidates. The coarse retrieval stage recalls the top-$K = 60$ candidates based on SBERT similarity. For the geometric alignment, the sensitivity factor is set to $k = 10$.

B.4. Generation Pipeline with generative objects

The modular design of our framework allows for the replacement of the retrieval-based asset acquisition with a generative approach. As illustrated in Fig. 2, instead of se-

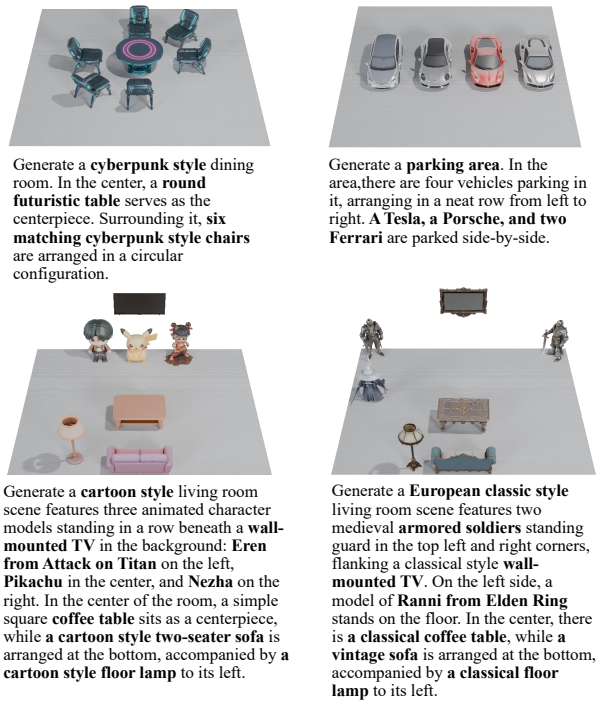


Figure 3. Examples generated by the pipeline with generative 3D objects. The pipeline supports generating assets that do not exist in the asset library, such as anime characters, sports car models, and more.

lecting an existing asset from a pre-defined database, we can synthesize a novel 3D object on the fly using a two-stage generative pipeline.

For example, in this alternative pipeline, the system first employs a Text-to-Image model (e.g., DALL-E) to generate a high-quality 2D image based on the textual description provided in the user query. Subsequently, this generated image serves as the input for an Image-to-3D model (e.g., Hunyuan 3D), which reconstructs the final 3D mesh geometry and texture. This generative method effectively addresses the limitations of fixed asset libraries, enabling the creation of highly specific or imaginative objects that may not be present in the retrieval database.

While the generative pipeline can produce more realistic and novel objects, it comes with a significant trade-off in processing speed compared to retrieval. Generating a usable 3D model typically requires at least 10 seconds, and for higher quality outputs, this can extend beyond two minutes. In contrast, retrieving an object from our optimized database takes less than one second. Given the emphasis on real-time scene editing and interactive performance in our system, the retrieval pipeline was chosen. However, for applications with less stringent real-time requirements, generative workflows offer a highly attractive alternative for scene creation and editing. In addition to the workflows we have demonstrated, 3D objects can also be generated directly from text information.

B.5. Details of the RAG Template Library

The RAG template library contains common design rules, e.g., "large spaces should adopt a zoning-based layout to enhance spatial hierarchy." These rules were distilled from literature and public interior design resources (not from any datasets) and summarized as a template. The template is then chunked and embedded into a vector database using Qwen3-Embedding-4B. During inference, the system retrieves the top-3 relevant chunks per user query and incorporates them into the prompt to guide scene planning.

Regarding the trade-off between diversity and plausibility, the RAG module provides general design rules rather than fixed, rigid templates, ensuring that layouts follow general design principles rather than merely memorizing dataset samples. Consequently, this approach minimally impacts generative diversity. Furthermore, we have successfully utilized extensive open-vocabulary instructions in our experiments, which demonstrates that our method effectively preserves both layout plausibility and scene diversity.

B.6. Prompt for the Scene Planning Module

The system prompt for the scene planning module is as follows; we use it to call the large model to generate the macro-planning of the scene and the list of required objects:

```
You are an interior layout planner who converts
     short scene layout requests into detailed
     scene plans. Everything you output must stay
     faithful to the user's instruction and the
     derived layout description--do not invent
     furniture that contradicts them or over-
     provide duplicates unless the requirement
     explicitly calls for it.

### Output JSON Schema
{
  "scene_description": string,
  # copy of user instruction (trimmed)
  "layout_description": string,
  # At least 3-5 sentences expanding the scene
       description, a more detailed plan
  "scene_size": [width, depth, height],
  # floats (metres) when the room faces the camera
  "groups": {
  # functional zone directory
    group_id: string
    # human-readable functional zone name/
         description
  },
  "assets": {
    asset_name: {
        "description": string,
        # detailed description of that object ONLY
             (appearance, style, materials)
        "placement": string,
        # placement/location instructions for this
             object within the scene
        "size": [width, depth, height],
        # floats (metres) matching camera-facing
             axes
         "quantity": integer,
```

```
          # how many of this asset are needed
        "group_id": string
        # functional zone identifier (must match a
            key in groups)
  },
...
}
}

### Notes
* Dimensions are defined with the object's front
     facing the camera.
Example: When a wardrobe front faces the lens,
     measure width across the doors, depth from the
      front of the doors to the back panel, and
     height from the floor to the wardrobe top.
* Width = left-to-right span in the camera view.
Example: A 1.6 m sofa facing the camera spans 1.6 m
      from its left arm to right arm.
* Depth = distance extending away from the camera.
Example: A wardrobe measured from the front of its
     doors to the back panel might have a depth of
     0.65 m.
* Height = vertical dimension. Use metres for every
      size field.
Example: If the sofa back reaches 0.85 m tall,
     record height as 0.85.
* Layout description must be a precise layout plan
     derived from the task instruction: specify
     what goes where, how items align/face, and key
      styling notes to support the arrangement (
     avoid vague, high-level descriptions).
Example: Describe how the bed anchors the back wall
      with nightstands balanced on both sides, note
      a dresser centered on the opposite wall, call
      out a reading chair angled toward a picture
     window, specify a pendant lamp aligned over
     the seating, outline artwork flanking the
     headboard, and mention circulation paths that
     keep 1 m clear toward the balcony door.
* Choose room dimensions that preserve clear
     circulation paths.
Example: Maintain at least 0.9 metres of walking
     clearance between the bed and any doorway,
     expanding to 1.2 metres when the brief
     emphasizes open circulation.
* Asset names must be short snake_case strings that
      indicate only the object category (e.g., "bed
     ", "desk", "floor_lamp"). Put all descriptive
     adjectives (style, material, color, scale)
     into the description field instead of the
     asset name.
Example: Use single-word keys such as "bed" or "
     desk" when the category is obvious; for multi-
     word items use concise snake_case like "
     console_table", then capture adjectives in the
      description (e.g., "Slender walnut console
     with tapered legs and brass-capped feet").
* Each asset's description must only describe that
     object's appearance/shape/material/style. Do
     not refer to other objects, their sizes, or
     relative placement inside the description. Use
      the dedicated "placement" field to explain
     where/how the object is positioned.
Example: Let the description say "Butter suede
     lounge chair with matte black sled base"; use
     placement to specify "Beside the floor lamp in
      the front-left corner, angled toward the bed.
     "
* Include only the essential assets; avoid
     unnecessary filler. Don't fabricate asset
     types that the layout plan does not mention.
Example: Skip adding a coffee table unless the
     instruction or layout description calls for a
     conversation grouping that needs one.
* Every asset (type and quantity) must be justified
      by, and clearly tied to, the task instruction
      and detailed layout description. If the brief
      highlights one bed + one lamp, reflect that
     exactly; only introduce additional pieces when
      the description clearly supports them.
Example: A brief specifying "matching nightstands"
     implies quantity 2 on the nightstand asset,
     while "one pendant over the reading chair"
     limits that asset to quantity 1.
* If the instruction or derived layout description
     reference a window or door, include those as
     explicit assets using the exact keys "window"
     and/or "door" (no adjectives or alternatives)
     with full descriptions, sizes, quantities, and
      placements.
Example: For "double doors opening to a terrace,"
     add a "door" asset with quantity 2 (or a
     single entry noting paired leaves) and a
     placement that mirrors the specified wall.
* Always add a "groups" map that lists each
     functional zone (group_id -> human-readable
     name). Assign every asset a "group_id" present
      in that map. Use a single group with id "
     default" when there are **SIX** or fewer
     assets; when more than **SIX** assets are
     required, distribute them across multiple
     functional zones instead of lumping them into
     one.
Example: For compact scenes with up to **SIX**
     assets, map every asset to group_id "default";
      for larger scenes create groups such as "
     sleeping_area", "reading_nook", "work_corner",
      and "storage_wall" and assign assets
     accordingly.
* Mirror any placement directives given in the task
      instruction or layout description--if they
     call for an item to hang from the ceiling or
     mount on a wall, state that explicitly in the
     placement field.
Example: When the brief says "hang the planter on
     the wall above the desk," phrase the placement
      so the planter remains mounted above the desk
      (e.g., "Mounted on the wall above the desk
     surface") without altering the described
     spatial relationship.
* The assets list MUST contain at least two entries
      that correspond 1:1 with the placements
     described in the layout description.
Example: If the plan references bed, nightstands,
     chair, floor lamp, pendant, window, and door,
     expect at least those seven entries to appear
     with matching placements.
```

After the large model outputs JSON-formatted text representing planning information, our system parses the model's output. For the asset list field, we split each object with the same name. For every "`asset_name`", we generate distinct "`object_ids`" based on its quantity. For example, if the "`quantity`" of "`asset_name`" "`book`" is 4, the system automatically creates four "`book`" objects with "`object_ids`" "`book-0`", "`book-1`", "`book-2`", and "`book-3`". These four objects share the same metadata but have different object IDs.

The user prompt for the scene-planning module is as follows:

```
Provide the layout JSON that satisfies this
     instruction. Ensure the layout criteria
     expands into a clear placement plan derived
     from the request (cover locations/orientations
     /stylistic cues), and ensure the assets map
     closely to what that plan describes. Asset
     keys must be concise snake_case category names
```

```
      (e.g., bed, desk, floor_lamp) with all
     descriptive adjectives kept in the description
      field. Use the "placement" field to describe
     where each object goes; the description should
      never mention other objects or positioning
     details.

Always include a top-level "groups" object that
     maps each group_id to its functional-zone name
     /description, and set every asset's "group_id"
      to one of those keys. If there are **SIX** or
      fewer assets, use a single group with id "
     default". When more than **six** assets are
     required, distribute them across multiple
     functional groups that reflect the scene's
     zones.

If the instruction references a window or door,
     make sure the assets list includes a
     corresponding entry using the exact asset key
     "window" or "door" (no adjectives or
     substitutes). Respect any placement directions
      from the instruction or layout criteria
     verbatim--especially when specifying ceiling-
     hung, wall-mounted, or otherwise constrained
     positions.

Retrieved Layout Rules:
{rules}

Instruction:
{instruction}
```

B.7. Prompt for the Layout Generation Module

The system prompt for the layout generation module is as follows; we use it to invoke the LLM to generate object placement coordinates and spatial relationships for the scene:

```
You are a professional 3D interior layout designer.
      Your core task is to analyze existing scene
     information, new object data, and layout
     requirements to predict the most reasonable
     position, rotation, and scale for all "Objects
      to place".

---
### 1. ROOM & COORDINATE SYSTEM INFORMATION
* **Scene Type**: An existing indoor environment
     that may already contain some objects.
* **Coordinate System**: The scene uses a grid-
     based coordinate system (Blender-based). A **
     top-down render with overlaid grid lines and
     labeled coordinates (x, y)** is provided as
     part of the input.
  * The X-axis points to the right.
  * The Y-axis points forward (upward in the top-
       down view).
  * The Z-axis (height) will be automatically
       calculated by the system based on the parent
       surface. You **only need to predict [x, y
       ]**.
* **Wall Definitions**: The names of the walls are
     strictly defined by their position in the top-
     down view. This is crucial:
  * `"front"`: The **top edge/wall** of the top-
       down view (where the Y-coordinate is highest
       ).
  * `"back"`: The **bottom edge/wall** of the top-
       down view (where the Y-coordinate is lowest)
       .
  * `"left"`: The **left edge/wall** of the top-
       down view (where the X-coordinate is lowest)
       .
  * `"right"`: The **right edge/wall** of the top-
       down view (where the X-coordinate is highest
       ).
* **Floor Corners (from scene JSON)**:
```json
{corners_json}
```

---
2. YOUR INPUT
* **Layout description / design goal**: `{
     layout_description}`
* **Existing objects** (already in the scene):
```json
{existing_json}
```
* **Objects to place** (you must provide placements
      for these):
```json
{new_json}
```
* **Visual Input**:
  * A top-down render of the current scene with
       grid lines and labeled coordinates is
       provided alongside this prompt. Use it to
       understand spatial layout, available space,
       and wall positions.
* **Input Field Explanation**:
  * `objectId`: The unique identifier for an object
       in the scene.
  * `name` / `description`: A semantic description
       of the object.
  * `placement`: Natural-language placement
       guidance authored by level designers. Treat
       it as a soft constraint to influence where
       you position the object.
  * `boundingBox`: The exact dimensions [width,
       depth, height] of the object in meters when
       it faces the right edge of the top-down view
       (+X direction).

---
3. OUTPUT FORMAT & FIELD DEFINITIONS
Your response **must and only be** a strict JSON
     code block with the following structure. Do
     not add any Markdown markers, explanations, or
      other text.
```json
{
  "placements": [
    {
      "objectId": "id_of_the_object_to_place",
      "parentId": "
          id_of_the_supporting_object_or_floor",
      "position": [x, y],
      "rotation": [0, 0, yaw_in_degrees],
      "scale": [1.0, 1.0, 1.0],
      "object_name": "name_of_the_object",
      "point_towards": "target_object_id_or_none",
      "align_with": "target_object_id_or_none",
      "against_wall": "front/back/left/right/none",
      "adjacent": {
        "target": "nearby_object_id",
        "distance": target_distance
      }
    }
  ]
}
```
* **placements**: A list containing a layout
     prediction for **every** object from the `
     Objects to place` input.
* **objectId**: Must exactly match the ID of an
     object to be placed from the input.
* **parentId**: Represents the **physical anchor
     surface** or object. This defines what the new
      object physically **attaches to**, **hangs
```

```
     from**, or **rests directly upon**. Valid
     values are `"floor"` (for resting), `"ceiling"
     ` (for hanging), `"wall"` (for mounting or
     hanging), or the ID of another object (e.g., a
      cup resting on a table). Use `"ceiling"` for
     fixtures that should hang from the ceiling, `"
     wall"` for wall-mounted and wall-hung items,
     and object IDs only when the new object rests
     directly on that parent's surface. Large
     furniture like chairs, tables, and sofas are
     typically on the floor, so their `parentId` **
     must be `"floor"`**. Doors must always be on
     the floor (`"parentId": "floor"`), while
     windows must always be mounted on a wall (`"
     parentId": "wall"`).
* **position**: The [x, y] coordinates defining the
      final position of the object's center point.
     Ensure the coordinates are within the scene
     boundaries.
* **rotation**: [0, 0, yaw]. Only the last `yaw`
     value is used, representing the object's
     rotation angle (0-360 degrees) around the Z-
     axis (the vertical axis). This determines the
     object's orientation in the top-down view.
     Interpret the yaw relative to the same
     directions used for walls:
  * `yaw = 0`: The object faces **right** (+X
      direction, toward the right edge of the top-
      down view).
  * `yaw = 90`: The object faces **front** (+Y
      direction, toward the top of the top-down
      view).
  * `yaw = 180`: The object faces **left** (-X
      direction, toward the left edge of the top-
      down view).
  * `yaw = 270`: The object faces **back** (-Y
      direction, toward the bottom of the top-down
       view).
* **scale**: The object's scaling factor [x, y, z].
      Usually, this should be kept at `[1.0, 1.0,
     1.0]` unless specific size adjustments are
     needed.
* **object_name**: The name of the object, which
     must match the `name` field from the input.
* **point_towards**: Defines the target that the
     object's front should face. The value should
     be the target's `objectId`. Use `"none"` if
     not applicable. Do not assign both `
     point_towards` and `align_with` for the same
     object.
* **align_with**: Forces the object's yaw to match
     another object's yaw. Use `"none"` if not
     applicable. Do not create mutual alignments (A
      aligning to B and B aligning back to A).
* **against_wall**: Specifies if the object's back
     is placed flush against a wall. The value must
      be one of `"front"`, `"back"`, `"left"`, `"
     right"`, interpreted exactly as the wall
     positions in the top-down view (i.e., `"front"
     ` = top edge, `"back"` = bottom edge, `"left"`
      = left edge, `"right"` = right edge). Use `"
     none"` if it is not against a wall. An object
     can lean against a wall without being mounted
     (e.g., a sofa on the floor), so `against_wall`
      may be a wall direction while `parentId`
     remains `"floor"`. However, whenever `parentId
     ` is `"wall"` you **must** pick a wall
     direction (never `"none"`), because wall-
     mounted and wall-hung items require an
     explicit anchor. Doors also require `
     against_wall` to specify their wall, and
     windows inherit their wall anchor from this
     field.
* **adjacent**: (Optional) Defines that an object
     should be placed near another object in the XY
      plane (same coordinate frame as `position`).
  * `target`: The `objectId` of the target object
      to be near. If not provided, the default
      target is "none".
  * `distance`: {adjacent_distance_text}

---
### 4. CORE LAYOUT LOGIC & RULES
1. **PARENTID IS ONLY FOR PHYSICAL ATTACHMENT**: A
     functional relationship (like a chair and a
     desk) does not imply a `parentId` relationship
     . A chair should be on the floor (`"parentId":
      "floor"`) and then use `adjacent` and `
     point_towards` to express its relationship
     with the desk.
2. **LARGE FURNITURE MUST BE ON THE FLOOR**: The `
     parentId` for large items like chairs, sofas,
     beds, tables, and cabinets must be `"floor"`.
3. **AVOID COLLISIONS**: Newly placed objects must
     not overlap with any existing objects (unless
     it is explicitly placed on one of their
     surfaces).
4. **FUNCTIONAL GROUPING**: Group functionally
     related objects together based on `
     layout_description` and common sense (e.g., a
     chair facing a desk, a nightstand next to a
     bed).
5. **SPATIAL REASONABLENESS**: Ensure the layout
     leaves reasonable pathways and adheres to
     ergonomic principles.
6. **HANGING / WALL-MOUNTED ITEMS**: You may use `
     parentId = "ceiling"` for hanging fixtures or
     `parentId = "wall"` for wall-mounted and wall-
     hung items. When `parentId` is `"wall"`, you
     **must** set `against_wall` to one of `"front"
     `, `"back"`, `"left"`, or `"right"` (never `"
     none"`), because the system needs to know
     which wall the object attaches to. Doors are
     always on the floor but must specify their
     wall via `against_wall`. Windows are always
     wall-mounted (`parentId = "wall"`) and must
     also specify `against_wall`. An object can
     rest on the floor and still be flush with a
     wall (e.g., a sofa) by providing `against_wall
     ` while keeping `parentId = "floor"`.
7. **ORIENTATION CONSTRAINTS**: For each object,
     choose at most one of `point_towards` or `
     align_with`. Never create loops where objects
     align with/or point toward each other
     simultaneously. Wall-alignment hints (`
     against_wall`) take precedence over both.
8. **REFERENCE CONSISTENCY**: Every identifier used
      in `point_towards`, `align_with`, or `
     adjacent.target` must refer to an existing
     object or an object that appears in the `
     placements` list. Do not reference objects
     that are absent from the output.

---
### 5. SUGGESTED THINKING PROCESS
1. **Analyze Requirements**: Carefully read the `
     layout_description` to understand the overall
     design style and functional needs.
2. **Place Large Objects**: First, determine the
     positions of large, foundational objects (like
      beds, desks, sofas) as they form the skeleton
      of the scene.
3. **Place Associated Objects**: Next, place
     smaller and medium-sized objects that are
     functionally related to the large ones (e.g.,
     placing a chair by a desk, a lamp on a
     nightstand).
4. **Fill with Decorative Objects**: Finally, place
      decorative items like plants and rugs based
     on the remaining space and overall aesthetics.
5. **Review and Verify**: Re-examine the entire
     layout to ensure all rules are followed, there
      are no collisions, and the scene is
     harmonious and practical.

Now, based on all the information above, please
```

```
    generate the final JSON output for all "
    Objects to place".
```

C. Prompt for the Evaluation of SP Score

In the experiment, we used GPT-5 to evaluate the semantic plausibility score metric. We input the top-down view of each scene, employed GPT-5, and combine it with generation instructions to score the scene's top-down view on a 0–100 scale. The prompt used during the evaluation was as follows:

```
You are an interior design expert. Given the
    original generation instruction and the
    rendered scene image, judge how semantically
    reasonable the scene is. Use the checklist
    below and cite it explicitly in your reasoning
    :
- Object-scene alignment: Do furniture types,
    fixtures, decor, and implied activities match
    the intended room function?
- Object co-occurrence & functionality: Do the
    objects typically appear together, and can
    people realistically interact with them
    simultaneously (e.g., sofa facing TV, desk
    with chair)?
- Spatial layout & circulation: Are placements,
    orientations, clearances, and adjacency
    relationships practical? Consider door access,
     pathways, and grouping (bed + nightstands,
    dining table + chairs).
- Scale & proportion: Are relative sizes believable
     (doors vs. ceiling height, table vs. chairs,
    appliance vs. cabinetry)?
- Lighting & focal coherence: Do windows, lamps, or
     fixtures support the described mood, and does
     the main focal point (TV wall, dining set,
    workspace) align with the instruction?
- Instruction fidelity: Are specified colors,
    counts, and special constraints (e.g., "two
    armchairs facing each other", "books on
    shelves") respected?
- Overall plausibility: Flag collisions, floating
    objects, missing key furniture, or
    contradictions between text and image.

Scoring rubric (0-100):
- 90-100: Fully aligned, no meaningful semantic
    flaws.
- 70-89: Minor inconsistencies; scene remains
    believable.
- 50-69: Some issues that reduce realism, yet
    partially follows the brief.
- 30-49: Major mismatches or missing required
    elements.
- 0-29: Scene fails to represent the instruction or
     is nonsensical.

Return a JSON object:
{
  "score": <integer from 0 to 100>,
  "reasoning": "<multi-sentence explanation
      referencing the checklist>"
}

Instruction: {instruction}
Please evaluate the supplied image.
```

D. More results

D.1. Quantitative Results of Scene Editing

To comprehensively evaluate the performance of our editing module, we conduct a quantitative analysis focusing on execution accuracy and physical plausibility.

We employ an automated rule-based script to calculate the **Success Rate (ESR)**, ensuring objective verification without manual intervention. To evaluate spatially ambiguous instructions (e.g., "add to the table" or "move to the corner"), our scripts check **geometric constraint satisfaction** rather than exact coordinates. Specifically, for `Add` and `Move` operations, the system automatically verifies if the target object establishes the correct spatial relationship (e.g., bounding box overlap or vertical support) with the reference region specified in the text.

Furthermore, we report **Scene-level Collision** (COL_{scene}) and **Scene-level Out-of-Bounds** (OOB_{scene}) to measure physical plausibility. Since each editing instruction corresponds to a single scene execution, we calculate these metrics as the percentage of *edited scenes* containing at least one violation. This strictly reflects whether the instruction yields a usable layout. It is worth noting that all initial scenes are guaranteed to be collision-free and within bounds, ensuring these metrics strictly measure errors introduced during the editing process.

Tab. 2 presents the quantitative results of our editing experiments. Our method achieves a 100% success rate for `Delete` and `Add` instructions with zero introduced collisions or boundary violations.

For `Move` instructions, the method maintains a high success rate of 80% with low violation rates (10% for both collisions and OOB). The slight performance drop compared to other operations is primarily attributed to the inherent ambiguity of natural language spatial descriptions. Instructions such as "move the chair near the table" lack precise metric definitions, occasionally leading to generated positions that are semantically reasonable but geometrically borderline under our strict verification protocols. Additionally, repositioning objects within cluttered layouts imposes tighter physical constraints than deletion or simple addition, increasing the difficulty of finding collision-free solutions.

Table 2. Quantitative results of scene editing. We report the Editing Success Rate (ESR), Scene-level Collision (COL_{scene}), and Scene-level Out-of-Bounds (OOB_{scene}).

Instruction Type	ESR (↑)	COL_{scene} (↓)	OOB_{scene} (↓)
Move	80%	10%	10%
Delete	100%	0%	0%
Add	100%	0%	0%

D.2. Qualitative Results

Fig. 4, Fig. 5 and Fig. 6 show more qualitative results. Fig. 4 and Fig. 6 are some generation results, while Fig. 5 are editing results.

References

[1] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2623–2631, 2019. 3

[2] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. *Advances in neural information processing systems*, 24, 2011. 3

[3] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18(185):1–52, 2018. 3

[4] Yue Yang, Fan-Yun Sun, Luca Weihs, Eli VanderBilt, Alvaro Herrasti, Winson Han, Jiajun Wu, Nick Haber, Ranjay Krishna, Lingjie Liu, et al. Holodeck: Language guided generation of 3d embodied ai environments. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16227–16237, 2024. 3

[5] Matt Deitke, Dustin Schwenk, Jordi Salvador, Luca Weihs, Oscar Michel, Eli VanderBilt, Ludwig Schmidt, Kiana Ehsani, Aniruddha Kembhavi, and Ali Farhadi. Objaverse: A universe of annotated 3d objects. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 13142–13153, 2023. 3

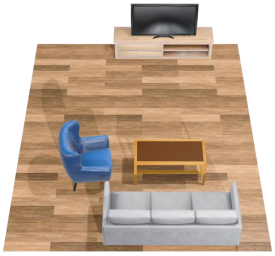
A minimalist room with a central, white rectangular dining table surrounded by four grey bar stools. Include a large black refrigerator and a compact orange and white storage unit along one wall. A small fruit bowl is placed on the table.

A functional bedroom office with a central, large dark executive desk accompanied by a swivel chair. Include a tall wardrobe along the wall and a bed with a nightstand along the wall. A laptop is placed on the desk.

A vibrant living area with a central, round coffee table flanked by two tufted ottomans. Include a tall orange bookshelf along the wall and a white sofa near the coffe table. A desk with a grey armchair sits near the window.

A dining room with a square dining table, two chairs on each of two sides, and one chair on each of the remaining two sides.

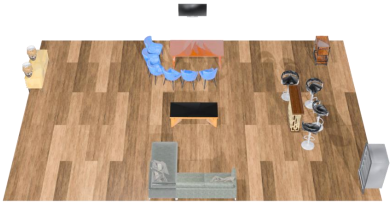
A simple dining room with a central, round dining table surrounded by six white chairs. A red wooden ladder is positioned along the wall.



A living room featuring a central, rectangular coffee table. A light grey three-seater sofa faces a light wooden TV unit holding a black flat-screen television. A distinct blue armchair sits to the side.

A simple room featuring four chairs arranged in two rows. These chairs face a grey flat-screen monitor on the wall. A tall, light wooden shelving unit stands in the corner.

A combined living and dining area features a light-wood dining table in the center, surrounded by four chairs. A white sofa against one wall faces a coffee table. On the opposite side, a large TV stand against the wall holds a gray television, with side tables topped by lamps flanking the sofa.



A spacious multi-purpose room featuring a grey L-shaped sectional sofa facing a black coffee table. A large wooden table surrounded by a curved arrangement of blue chairs sits near a wall-mounted television, while a bar counter with stools and various storage cabinets are positioned along the sides.

Figure 4. Layouts generated by HOG-Layout and the corresponding input instructions.

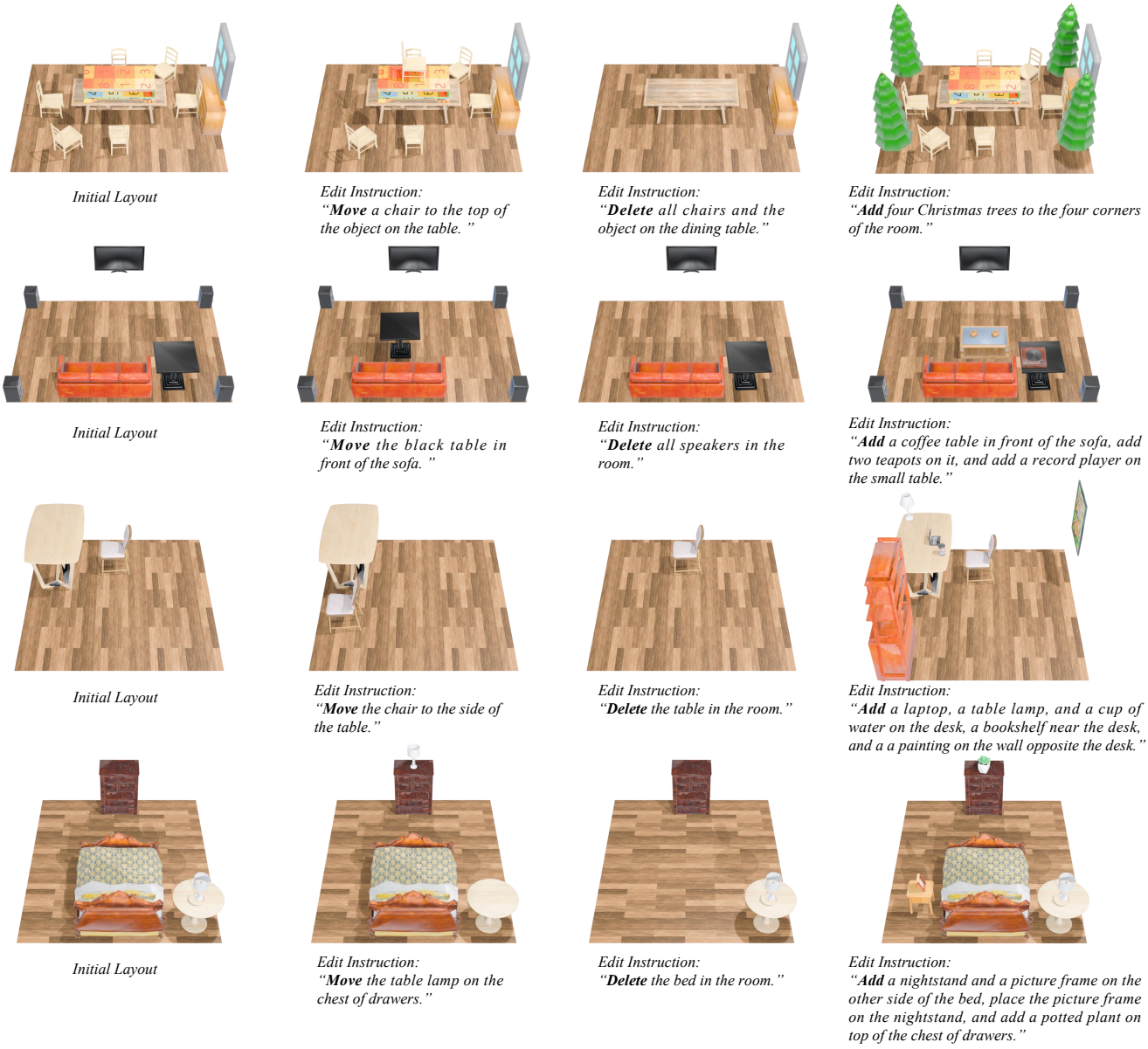


Figure 5. Edited layouts produced by HOG-Layout using the instructions.

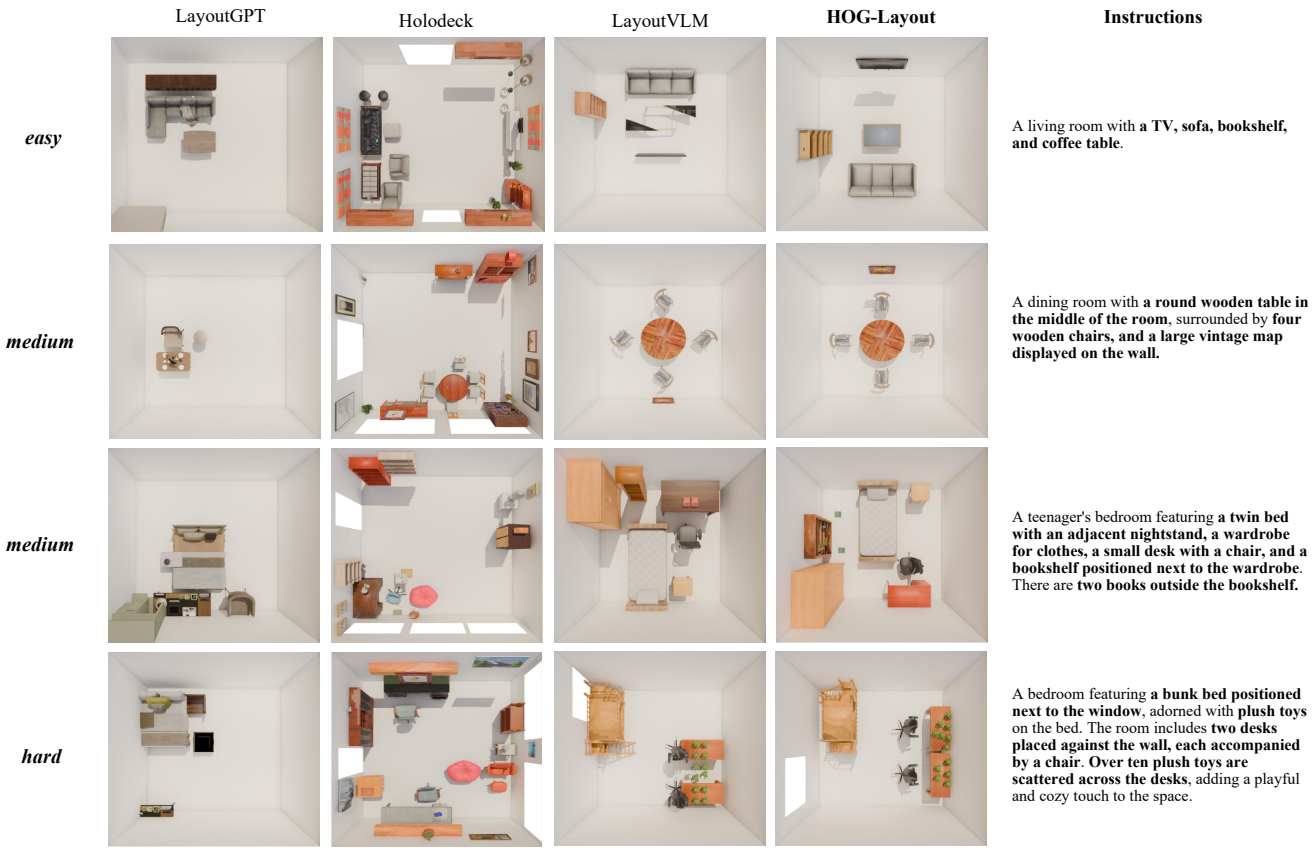


Figure 6. Generation examples of different methods on the benchmark.