Supplemental Materials

A. Additional Qualitative Results

A.1. Qualitative Floorplan Comparison

We provide a qualitative comparison with ChatHouseDiffusion (CHD) [33] under the same raster-space protocol used for the quantitative IoU evaluation in Sec. 4.1, as shown in Fig. 6. This is important because CHD formulates floor-plan generation as an image-based diffusion process, whereas our method is a training-free geometric solver that operates directly on polygons.

Figure 6. **Qualitative floorplan comparison with CHD.**

Shared raster-space protocol. To make the comparison as fair as possible, both methods are evaluated on a common 64×64 grid aligned to the same floor-plan bounding box. Since CHD produces floor-plan outputs in image form, our method is discretized into the same raster space after uniformly scaling the polygonal layout and quantizing the coordinates, following the evaluation protocol used in Sec. 4.1.

Visual style mismatch. The contour mismatch mainly arises from rendering conventions rather than layout structure. CHD visualizations typically include thick exterior contours and explicit white wall gaps between adjacent rooms, whereas our rasterization directly assigns each interior pixel to a room label. As a result, the rendered appearances may differ even when the underlying room arrangements are highly similar.

Why the rasterization comparison is fair. This rasterization introduces a small representational gap for our method, since a vectorized layout must be converted into raster form before comparison. Minor discrepancies may therefore appear near thin boundaries, corners, or narrow room connections. However, this effect is limited and systematic: both predictions and ground-truth annotations are compared after the same scaling and rasterization procedure, so the quantization error is limited and does not materially affect the IoU comparison.

A.2. Qualitative comparison with Holodeck

We further provide a qualitative comparison with existing methods that generate both room layouts and instantiated 3D scenes from high-level semantic descriptions. Representative systems in this category include AnyHome [14] and Holodeck [47]. We focus on Holodeck, since AnyHome relies on a HousGAN++-style floorplan backend mainly tailored to residential settings, making it less compatible with our open-vocabulary, non-residential scenario.

Figure 7 highlights a key methodological difference. Holodeck follows a bottom-up paradigm that directly predicts room corners and stitches them into a floor layout. This design is effective for lightweight single-floor scene synthesis, but it does not explicitly model building contours, topology-preserving floor partitioning, or cross-floor structural consistency. In contrast, MANSION adopts a top-down formulation, where each floor is generated as a constrained partition under contour, topology, and vertical-core constraints. This makes our method better suited for multi-floor buildings and large-scale non-residential spaces.

As shown in Fig. 7, this top-down design provides five practical advantages: **contour control**, **topology control**, **vertical alignment**, **realistic placement**, and **style consistency**. The first three stem from our constrained floorplan formulation, while the latter two come from our scene-instantiation design for large, non-residential spaces and building-level room-card style propagation.

The goal of this comparison is to contrast generation paradigms rather than claim strict metric superiority under a shared benchmark. Since Holodeck only produces corner-based layouts and does not explicitly target contour or topology controllability, the most faithful comparison at the layout-generation level is qualitative.

A.3. Structural Flexibility and Physical Fidelity

Fig. 8 highlights two properties of MANSION that are important for physical realism. First, MANSION is *contour-controllable* rather than restricted to identical cross-floor outlines. The consistent contours used in some main-paper experiments are an evaluation simplification, not an algo-

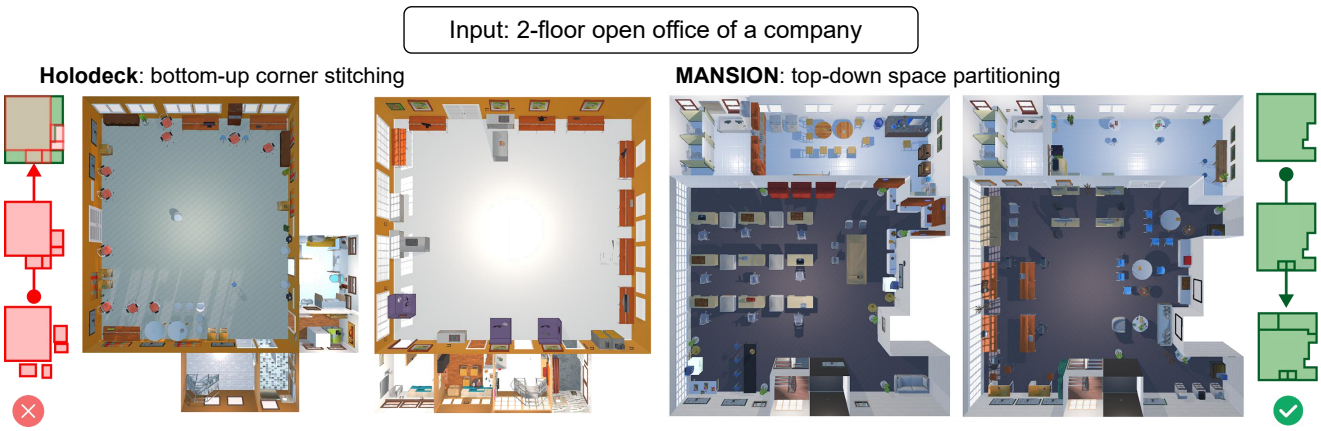


Figure 7. Qualitative comparison between Holodeck and MANSION under high-level semantic building prompts.

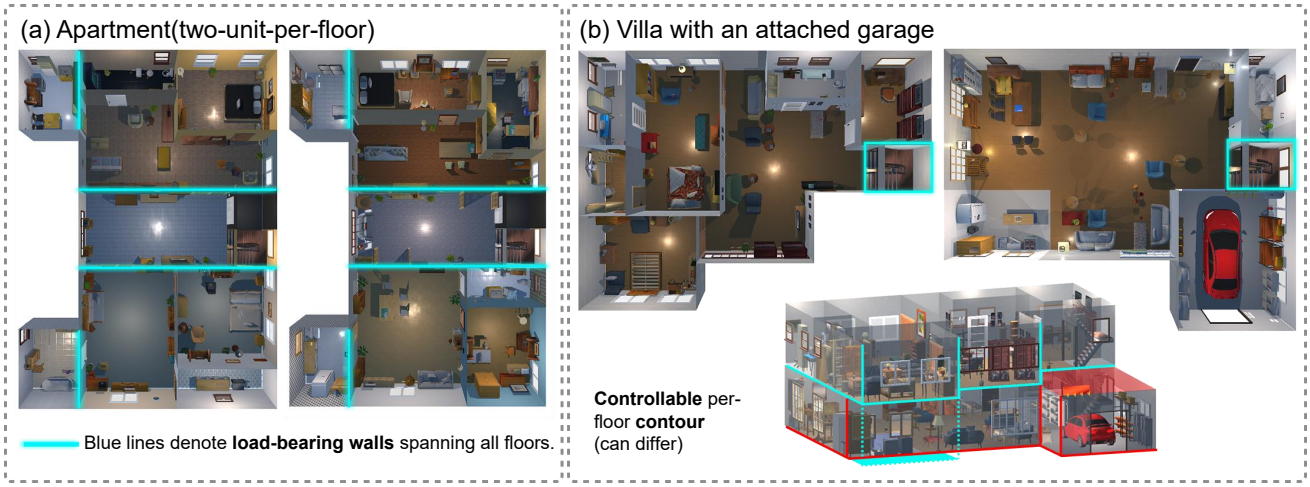


Figure 8. Structural realism in MANSION. (a) Cross-floor load-bearing walls (blue) are preserved to maintain vertical consistency. (b) A villa with a protruding room and garage shows controllable per-floor outer contours.

rithmic requirement: floor-wise footprints may differ across floors when specified by the building program.

Second, our notion of vertical consistency goes beyond stairs and elevators. During recursive partitioning, load-bearing walls and other fixed vertical structures are preserved as geometric constraints for subsequent floor splits, enabling cross-floor structural coherence in buildings with more realistic and complex organization. As a result, MANSION supports better apartment-, office-, and villa-style structures that are closer to real-world buildings, rather than simple floor-by-floor stacking.

B. MansionWorld Dataset Details

B.1. Physical Scale and Functional Composition

To build a benchmark environment that both supports high-performance simulation and enables a comprehensive evaluation of embodied AI across tasks of varying complexity, MANSIONWORLD is carefully designed along two dimensions: physical scale and functional scene composition.

Considering the physics load of AI2-THOR when handling dense rigid-body interactions, we cap the effective area of each floor at about $500\,\mathrm{m}^2$ to maintain stable frame rates in complex interaction scenes. Leveraging the dynamic floor loading mechanism of the MANSION framework, we adopt a *single-floor constrained, vertically open* spatial strategy: while the area of each individual floor is kept within a controlled range for simulation efficiency, the total number of floors in a building can be extended up to ten.

For functional composition, instead of uniformly sampling scene types, we follow the major application domains of current real-world robots and construct a three-way mixture of *residential* (50%), *office* (30%), and *public* (20%)

buildings. This mixture is intended to cover home service robots, intra-building delivery and inspection robots, as well as robots operating in public spaces such as shopping malls, hospitals, and campuses. While residential scenes form roughly half of the corpus in order to support an easy-to-hard curriculum grounded in everyday household tasks, a key novelty of MANSIONWORLD compared to prior home-centric benchmarks lies in its substantial share of non-residential office and public buildings at the building scale. These non-residential environments are where most of our long-horizon, building-scale evaluations are conducted, and they underpin the "non-residential" emphasis in the main paper.

On top of this, we deliberately impose a *difficulty curriculum* where simple scenes are more frequent while complex scenes form a long tail. Residential buildings serve as the basic testbed: a large number of *Studio & Small Flat* units, although compact in size (typically 60–90 m^2), are populated with high object density and intentionally irregular layouts, in order to stress-test agents' fine-grained manipulation, short-range navigation, and robustness to clutter (e.g., avoiding toys in a messy living room to find a TV remote). In contrast, multi-floor *Family Apartment* and *Duplex & Townhouse* units introduce vertical connections via internal staircases and elevators, enabling cross-floor tasks in domestic environments. Agents must explicitly model the abstract notion of "floor" to accomplish compound household tasks that depend on spatial memory and state tracking, such as *"collect dirty clothes from the bedroom on the second-floor and bring them to the laundry room on the first-floor."*

Office and public buildings further emphasize semantic reasoning and socially aware navigation. The office subset often exploits large, nearly 500 m^2 floor plates with long corridors and repetitive workstation patterns, posing challenges for robust self-localization in highly similar local structures and for long-range intra-building delivery (e.g., distributing documents or parcels across an eight-floor building). The public subset (e.g., shopping malls, hospitals, schools) highlights explicit functional zoning and semantic priors: agents cannot rely on geometry alone, but must leverage commonsense knowledge such as "pharmacies are not located in cafeterias" or "fresh produce sections tend to be adjacent to cold-chain facilities" to build high-quality semantic maps and perform efficient target search. This addresses a gap in existing datasets, which mostly focus on homes and single-floor dwellings. Overall, MANSIONWORLD contains a larger number of low-rise, small-to-medium scale scenes that are convenient for day-to-day algorithm development and rapid evaluation, while still reserving a non-trivial proportion of high-rise, large-scale office and public buildings to stress-test the generality and upper-limit performance of embodied systems.

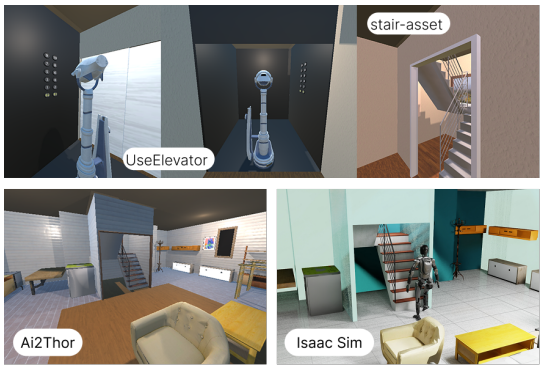


Figure 9. Additional details of the MansionWorld ecosystem.

B.2. Qualitative Examples of MansionWorld Scenes

To complement the statistics in Fig. 3, we further visualize several representative buildings from MANSIONWORLD and the egocentric observations perceived by an embodied agent operating inside these buildings. Each example pairs a 3D view of a multi-floor building with a first-person view from a highlighted room, see Fig. 10 and Fig. 11.

B.3. Cross-Floor Mobility and Simulator Transfer

Fig. 9 shows several of our extended cross-floor assets and skills, which support floor-to-floor interaction in MansionWorld, as well as an example of transferring an AI2-THOR scene to NVIDIA Isaac Sim.

C. Multi-floor Generation Pipeline Details

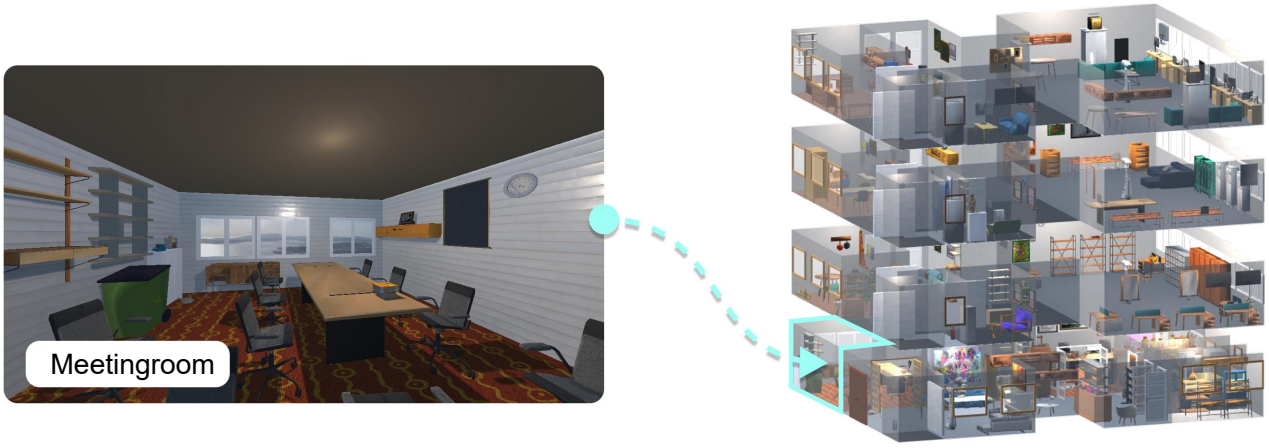
C.1. Input specification and global orchestration

The user (or a higher-level generator) provides a natural-language building description D together with a (possibly partial) set of numerical constraints. In practice, most of these constraints can be inferred by a large language model from D, and only the geometric footprint is synthesized by the planner. For clarity, we write them explicitly as

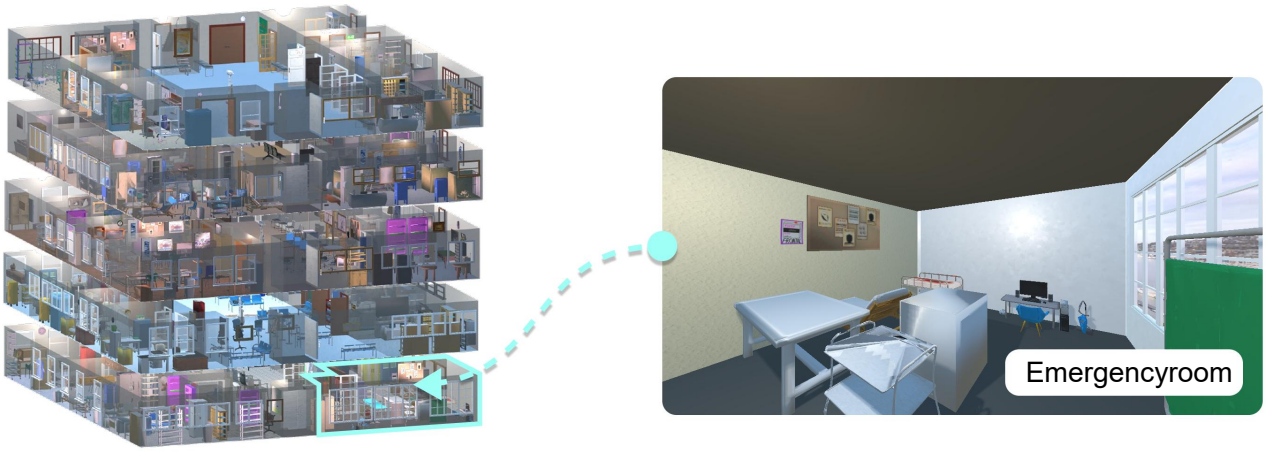
- **Target floor count** F_{target} (optional): a desired number of floors. When not explicitly specified, it is inferred from D (e.g., "two-storey townhouse" or "high-rise office").
- **Target floor area** A_{target} (optional): a desired gross floor area (per building or per floor). If not given, it is similarly inferred from D under simulator constraints (e.g., a per-floor area cap for stable physics).
- **Footprint constraint** P_{env} (optional): an outer polygon of the building envelope. In our main experiments, this footprint is *not* provided by the user; instead, the planner samples a feasible outline consistent with $(F_{\text{target}}, A_{\text{target}})$ and engine limits, and we denote the resulting footprint as P_{env}.

Any of the scalar constraints F_{target} and A_{target} may be omitted in the input; in that case, the planner first invokes an

(a) A high school building

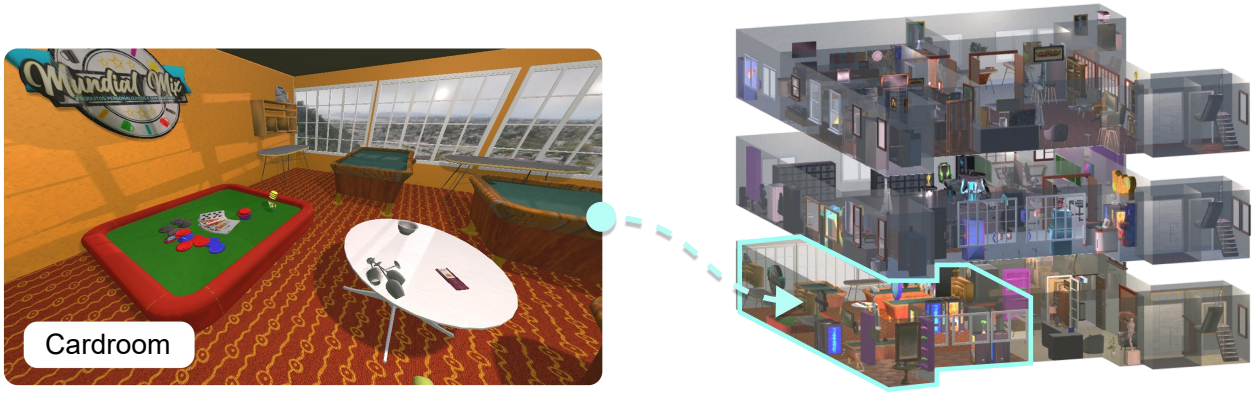


(b) A four story office building

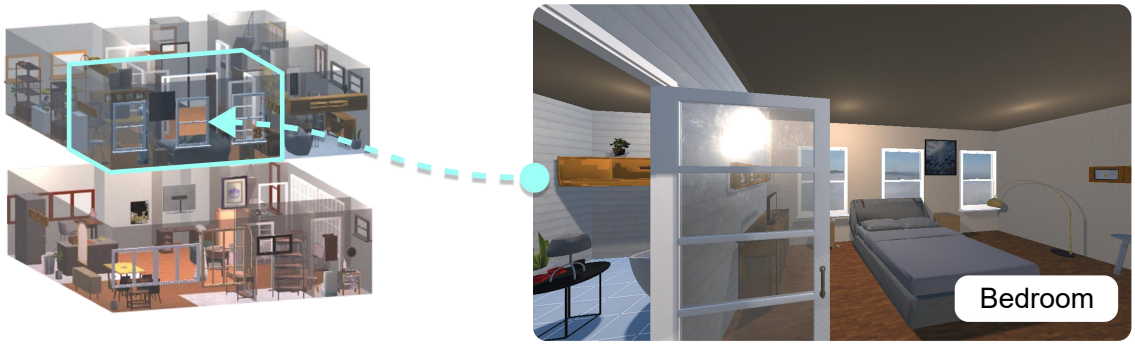


(c) A large-scale hospital

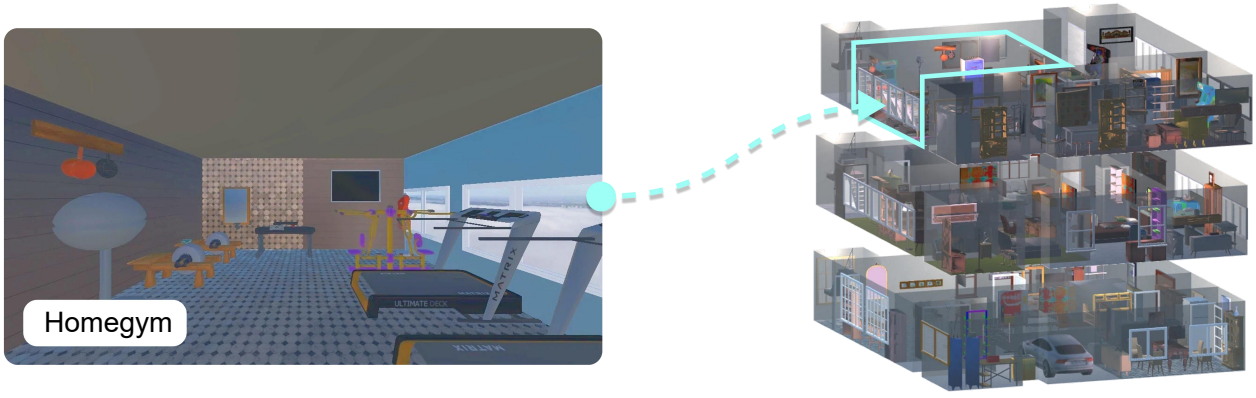
Figure 10. Qualitative examples of non-residential buildings in MANSIONWORLD.



(a) A entertainment complex



(b) A compact apartment designed for two people



(c) A three-story luxury villa equipped with entertainment and exercise facilities

Figure 11. Qualitative examples of entertainment and residential buildings in MANSIONWORLD.

Algorithm 1 Global multi-floor generation pipeline

Require: Description D; optional F_{target}, A_{target}, P_{env}
Ensure: Per-floor scenes $\{S_i\}_{i=1}^{F}$

$(\hat{F}, \hat{A}) \leftarrow \textsc{ResolveNumericConstraints}(D, \backslash F_{\text{target}}, A_{\text{target}})$
$B_{\text{plan}} \leftarrow \textsc{PlanBuildingProgram}(D, \hat{F}, \hat{A}, P_{\text{env}})$
$F \leftarrow \textsc{NumFloors}(B_{\text{plan}})$
$S \leftarrow \emptyset$
for $i = 1$ to F **do**
  $G_i \leftarrow \textsc{GenerateFloorTopology}(B_{\text{plan}}, i)$
  $L_i \leftarrow \textsc{SolveFloorLayout}(G_i, B_{\text{plan}}, i)$
  $X_i \leftarrow \textsc{ApplyFloorStructure}(L_i)$
  $Y_i \leftarrow \textsc{ApplyWallsAndOpenings}(X_i)$
  for each room $r \in \textsc{Rooms}(L_i)$ **do**
    $Y_i \leftarrow \textsc{PlaceLargeObjects}(Y_i, r)$
    $Y_i \leftarrow \textsc{PlaceSmallObjects}(Y_i, r)$
  end for
  $Y_i \leftarrow \textsc{AddLighting}(Y_i)$
  $Y_i \leftarrow \textsc{AddSkybox}(Y_i)$
  $S_i \leftarrow \textsc{PlaceAgentSpawn}(Y_i, B_{\text{plan}}, i)$
  $S \leftarrow S \cup \{S_i\}$
end for
return S

LLM to parse D and derive reasonable default values. The footprint P_{env} is typically synthesized (or, in dataset-driven settings, supplied by the benchmark) and is never manually drawn by the user in our pipeline. This makes the system applicable both when the user prescribes an approximate scale ("three small floors") and when global dimensions are left entirely to the generator.

Given $(D, F_{\text{target}}, A_{\text{target}}, P_{\text{env}})$, the multi-floor controller first synthesizes a global building program B_{plan} as above. It then proceeds floor by floor. For each floor index $i \in \{1, \ldots, F\}$, it generates a symbolic room topology $G_i = (R_i, E_i)$ consistent with the cross-floor skeleton, and calls the single-floor solver in Algorithm 2 to turn G_i into a geometric layout L_i. Internally, this solver constructs a cut schedule $\mathcal{R}_i$ and applies the topology-aware cutting node in Algorithm 3 round by round. Once the 2D floorplan L_i is fixed, a deterministic instantiation pipeline applies floor surfaces, walls and openings, then iterates over rooms to populate large and small objects, and finally adds lighting, skybox and agent spawns to obtain an executable 3D scene. The whole process is summarized below.

C.2. Single-floor topology-driven floorplan solver

Fig. 12 illustrates the single-floor solver: given a symbolic room topology graph, we construct cut rounds over the free region and finally instantiate the resulting layout as an executable 3D scene. We reuse the notation from Sec 3.1: for floor f we write Ω_f for the free region after removing vertical cores and $G_f = (R_f, E_f)$ for the room graph with target areas $\{a_r\}_{r \in R_f}$. The solver is a constructive procedure that approximately optimizes the layout objective in Sec. 3.1 under the hard topological constraint $\text{Topo}(L, G_f)$.

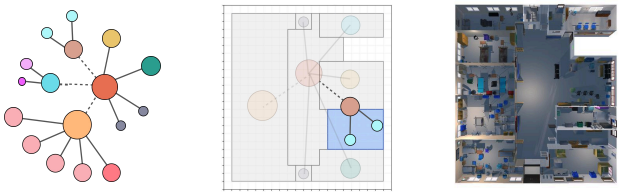


Figure 12. Illustration of the single-floor, topology-driven pipeline. (a) Input room topology graph. (b) Cut-round construction and hierarchical splitting over the free region. (c) Final 3D scene instantiation in AI2-THOR after applying structure, objects, and lighting.

Algorithm 2 Single-floor topology-driven floorplan solver

Require: Floor index f; free region Ω_f; room graph $G_f = (R_f, E_f)$; target areas $\{a_r\}_{r \in R_f}$; vertical cores V
Ensure: Floorplan layout L_f partitioning Ω_f

$main \leftarrow \textsc{SelectHubNode}(G_f)$
$\mathcal{R}_f \leftarrow \textsc{BuildCutRounds}(G_f, main, V)$
$L_f \leftarrow \textsc{InitLayout}(\Omega_f, V, main)$
for each $(p_t, C_t) \in \mathcal{R}_f$ **do**
  $L_f \leftarrow \textsc{CutNode}\big(L_f, p_t, C_t, G_f, \{a_r\}\big)$
end for
return L_f

The algorithm proceeds by hierarchical splitting. A hub node $main \in R_f$ is chosen as the root; a cut-planning routine constructs a sequence of rounds $\mathcal{R}_f = \{(p_t, C_t)\}_{t=1}^{T}$ with parents $p_t \in R_f$ and non-empty child sets $C_t \subseteq R_f$; and a generic cutting node successively refines the layout in each round.

Cut-round construction. $\textsc{BuildCutRounds}$ performs a breadth-first traversal on G_f rooted at $main$, assigns a depth to each node, and groups non-vertical rooms by depth and parent. Vertical-core nodes are excluded from the parent set. For each non-vertical parent p and its non-empty child cluster $C \subseteq R_f$, it emits a round (p, C). The resulting $\mathcal{R}_f$ induces an order that respects the graph structure (no child is instantiated before its parent) and expands from the hub to the periphery.

Topology-aware cutting node and adaptive growth. For a fixed round (p_t, C_t) and current layout L_f, the cutting node first extracts the parent region $\Omega_f(p_t) \subseteq \Omega_f$ and renders a top-down preview in which the polygon of p_t is highlighted against the rest of the floorplan. This image, to-

gether with L_f, G_f, p_t, C_t and $\{a_r\}$, is given to an MLLM that outputs a seed plan $\sigma_t = \{(r, c_r, \alpha_r) \mid r \in C_t\}$, where c_r is a continuous seed (approximate centroid) in $\Omega_f(p_t)$ and α_r is a target area fraction consistent with a_r and $|\Omega_f(p_t)|$.

Conditioned on σ_t, the node runs an adaptive sampling procedure with $N_{\text{retry}} = 10$ retries and batch size $B = 100$ local candidates per retry. For each child $r \in C_t$ it computes an initial radius $R_r^{(0)} = r_{\text{base}} + k \cdot a_r / |\Omega_f(p_t)|$ (with fixed $r_{\text{base}} = 2$ in grid units and scaling factor k) and at retry j uses a scaled radius $R_r^{(j)} = \gamma_j R_r^{(0)}$ for a monotonically increasing sequence $(\gamma_j)_j$. Intuitively, $R_r^{(j)}$ is the adaptive perturbation radius around the seed for room r in retry j, controlling how far candidate seeds may move away from the MLLM-proposed centroid. In retry j, it samples B seed perturbations inside the discs of radius $R_r^{(j)}$ (with a minimum separation constraint between seeds), grows B local candidate partitions of p_t, filters them by the predicate $\mathrm{Topo}(\cdot, G_f)$, and scores the survivors with the score function $\mathrm{Score}(L; \mathbf{w})$ described below. If at least one candidate survives in retry j, the best-scoring one is accepted and the retry loop terminates. If all N_{retry} retries fail, the node falls back to a Monte Carlo seeding strategy: seeds are sampled uniformly in $\Omega_f(p_t)$ in decreasing order of target area, subject to repulsion, and the same growth, topology filtering and scoring pipeline is applied.

The cutting node is summarized in the following skeleton.

Algorithm 3 Topology-aware cutting node with MLLM-guided seeds (skeleton)

Require: Layout L_f; parent p_t; children C_t; room graph G_f; target areas $\{a_r\}$
Ensure: Updated layout L'_f where p_t is split into C_t

$\Omega_f(p_t) \leftarrow \textsc{LocalFootprint}(L_f, p_t)$
$I_t \leftarrow \textsc{RenderHighlightPreview}(L_f, \Omega_f(p_t), p_t)$
$\sigma_t \leftarrow \textsc{PlanSeedsWithMLLM}(I_t, L_f, G_f, p_t, C_t, \{a_r\})$
$\{R_r^{(0)}\} \leftarrow \textsc{ComputeBaseRadii}(\Omega_f(p_t), C_t, \{a_r\})$
$best \leftarrow \textsc{None}$
for $j = 0$ to $N_{\text{retry}} - 1$ **do**
  $\tilde{\Sigma}_j \leftarrow \textsc{SampleSeedBatch}(\sigma_t, \{R_r^{(0)}\}, j)$
  $\mathcal{L}_j \leftarrow \textsc{GrowCandidates}(\Omega_f(p_t), p_t, C_t, \tilde{\Sigma}_j)$
  $\mathcal{L}_j \leftarrow \textsc{FilterByTopology}(\mathcal{L}_j, L_f, G_f)$
  $cand \leftarrow \textsc{SelectBestByScore}(\mathcal{L}_j)$
  if $cand \neq \textsc{None}$ **then**
    $best \leftarrow cand$; **break**
  end if
end for
if $best = \textsc{None}$ **then**
  $best \leftarrow \textsc{FallbackMonteCarlo}(\Omega_f(p_t), p_t, C_t, \{a_r\}, L_f, G_f)$
end if
$L'_f \leftarrow \textsc{MergeLocalPartition}(L_f, p_t, best)$
return L'_f

Energy-based scoring and weight selection. We now detail the energy-based $\mathrm{Score}(L; \mathbf{w})$ objective introduced in Eq. 3.1. For each local candidate layout L of the parent region, we compute a per-room energy and aggregate into a total energy $E(L; \mathbf{w})$, from which the score is obtained by negation.

For every child room $r \in C_t$ with realized polygon P_r, target area a_r, and seed c_r, we extract four raw features:

- $f_{\text{ratio}}(r)$ (`ratio`): relative area error $|\operatorname{area}(P_r) - a_r|/a_r$;
- $f_{\text{seed}}(r)$ (`seed_dist`): Euclidean distance between the centroid of P_r and the input seed c_r;
- $f_{\text{wall}}(r)$ (`wall_contact`): absolute length of the boundary intersection between P_r and the envelope $\partial\Omega_f(p_t)$ of the parent region, i.e. $|\partial P_r \cap \partial\Omega_f(p_t)|$;
- $f_{\text{corner}}(r)$ (`extra_corners`): $\max(0, n_{\text{int}}(r) - 4)$, where $n_{\text{int}}(r)$ is the number of non-collinear corners of P_r that do *not* lie on $\partial\Omega_f(p_t)$.

Among these, f_{ratio}, f_{seed}, and f_{corner} are *penalty* terms (smaller $\Rightarrow$ better), while f_{wall} is a *reward* term (larger $\Rightarrow$ better, since more envelope contact yields more regular rooms).

To balance heterogeneous scales, we apply a *mixed normalization* strategy. Only f_{seed} undergoes min–max normalization across the room set within a single candidate:

$$z_{\text{seed}}(r) \;=\; \mathrm{clamp}_{[0,1]}\!\left(\frac{f_{\text{seed}}(r) - f_{\text{seed}}^{\min}}{f_{\text{seed}}^{\max} - f_{\text{seed}}^{\min} + \varepsilon}\right),$$

with $f_{\text{seed}}^{\min} = \min_r f_{\text{seed}}(r)$ and $f_{\text{seed}}^{\max} = \max_r f_{\text{seed}}(r)$. For f_{ratio} and f_{corner}, we found that using raw values directly provides more stable value differences across candidates with varying room counts and boundary complexities, because min–max normalization can compress informative differences when the candidate set is homogeneous. Similarly, f_{wall} enters as a raw value clamped to $[0, 1]$.

The per-room energy contribution is

$$\begin{aligned} e(r) \;=\; & w_{\text{ratio}}\, f_{\text{ratio}}(r) \;+\; w_{\text{seed}}\, z_{\text{seed}}(r) \\ & +\; w_{\text{corner}}\, f_{\text{corner}}(r) \\ & -\; w_{\text{wall}}\, \mathrm{clamp}_{[0,1]}\big(f_{\text{wall}}(r)\big), \end{aligned}$$

The dominant w_{ratio} strongly penalizes area mismatch; w_{corner} discourages complex room shapes; w_{wall} encourages rooms to align with the building envelope; and w_{seed} provides a mild bias toward the MLLM-proposed centroid. where penalty terms enter with positive signs (higher $\Rightarrow$ worse) and the wall reward enters with a negative sign (more contact $\Rightarrow$ lower energy). The total energy sums over

all child rooms and relates to the search objective (Eq. 3.1) by negation:

$$E(L; \mathbf{w}) = \sum_{r \in C_t} e(r),$$
$$\mathrm{Score}(L; \mathbf{w}) = -E(L; \mathbf{w}).$$

Among all candidates in a round that satisfy $\mathrm{Topo}(\cdot, G_f)$, the cutting node retains the layout with the lowest energy (equivalently, the highest Score).

We set $\mathbf{w}$ by random search on a held-out subset of the RPLAN dataset: candidate weight vectors are sampled from a low-dimensional simplex, the solver is run on RPLAN-style instances, and configurations that yield good area agreement and regular room shapes are retained; one such $\mathbf{w}$ is fixed for all experiments. We intentionally use this hand-crafted, interpretable energy rather than a learned scoring network, since RPLAN is dominated by residential layouts and a learned scorer trained on it would be strongly domain-specific. In contrast, the feature-based energy can be reweighted to accommodate different building types without retraining.

Spur removal and hole filling. After growth, room polygons may contain thin protrusions (spurs)—single cells connected to the room body by at most one edge. A spur cell is identified as any occupied cell whose same-room 4-neighbor count is at most one while having at least one neighbor belonging to a different room or lying outside the interior. Spur removal proceeds iteratively: in each pass, all detected spur cells are set to empty; passes repeat until no spurs remain, yielding a spur-free grid. Subsequently, each remaining empty connected component within the interior is filled by the room sharing the longest boundary with that component. This fill-then-clean cycle repeats up to 20 iterations to ensure that hole filling does not re-introduce spur artifacts.

A practical limitation is that when $\Omega_f(p_t)$ is highly constrained and G_f is complex, the number of candidates satisfying the hard topological constraints can be very small. In this scenario, the solver is feasibility-driven, where the influence of $\mathrm{Score}(L; \mathbf{w})$ is reduced.

D. Task-Semantic Scene Editing Agent

D.1. System Architecture

We illustrate the detailed architecture of our Task-Semantic Scene Editing Agent in Fig. 13. Operating as a high-level **neuro-symbolic scheduler**, our system decouples semantic reasoning from physical simulation, achieving both computational efficiency and physical plausibility. The architecture comprises three core subsystems:

The ReAct Controller (Cognitive Layer). The agent's core is a Large Language Model (LLM) utilizing a ReAct (Reason+Act) protocol. It processes natural language instructions and conversation history, outputting structured JSON actions to invoke specific tools. This abstraction enables the agent to plan over long horizons by manipulating scene semantics rather than raw pixels or low-level motor commands.

Hybrid State Management (Dual-Backend). A key innovation is the separation of static data from dynamic simulation. This dual-backend approach ensures optimal resource utilization:

- **Static Semantic State (JSON + Asset DB):** The primary "source of truth" is a lightweight Holodeck-compatible JSON file. All logical checks (e.g., path connectivity) and geometric planning (e.g., surface area calculation) are executed directly against this JSON structure and an external Asset Metadata Database (Asset DB). This avoids rendering overhead, enabling rapid "mental simulation" and topological reasoning.
- **On-Demand Physics Engine (Unity):** Calling physics simulation is an expensive, on-demand resource. An AI2-THOR controller is temporarily instantiated only for actions requiring physical validation (specifically, `PlaceInContainer` or `PlaceOnSurface`). It executes atomic physics-based actions (e.g., `SpawnAsset`, `OpenObject`, `PlaceObjectAtPoint`) to resolve collisions and gravity. The object's final valid pose is then synchronized back to the static JSON, and after that, the simulator instance is stopped.

The Tool Invoker. Serving as the execution bridge, this component parses the JSON requests from the ReAct controller and redirects function calls to the appropriate backend, from querying the static semantic state for fast perception tasks to triggering the on-demand physics engine for complex interactions, and returns the execution results as observations to the agent.

D.2. Tool Library Cards

We present the detailed specifications of the toolset used in the "Check-and-Provision" workflow. For brevity, we categorize the **Data Source** of each tool into three components:

- **JSON**: Operations on the static scene graph file (fast, geometric).
- **Asset DB**: Queries to the external Objaverse/AI2-THOR metadata library.
- **Unity**: Runtime physics simulation via AI2-THOR (atomic actions).

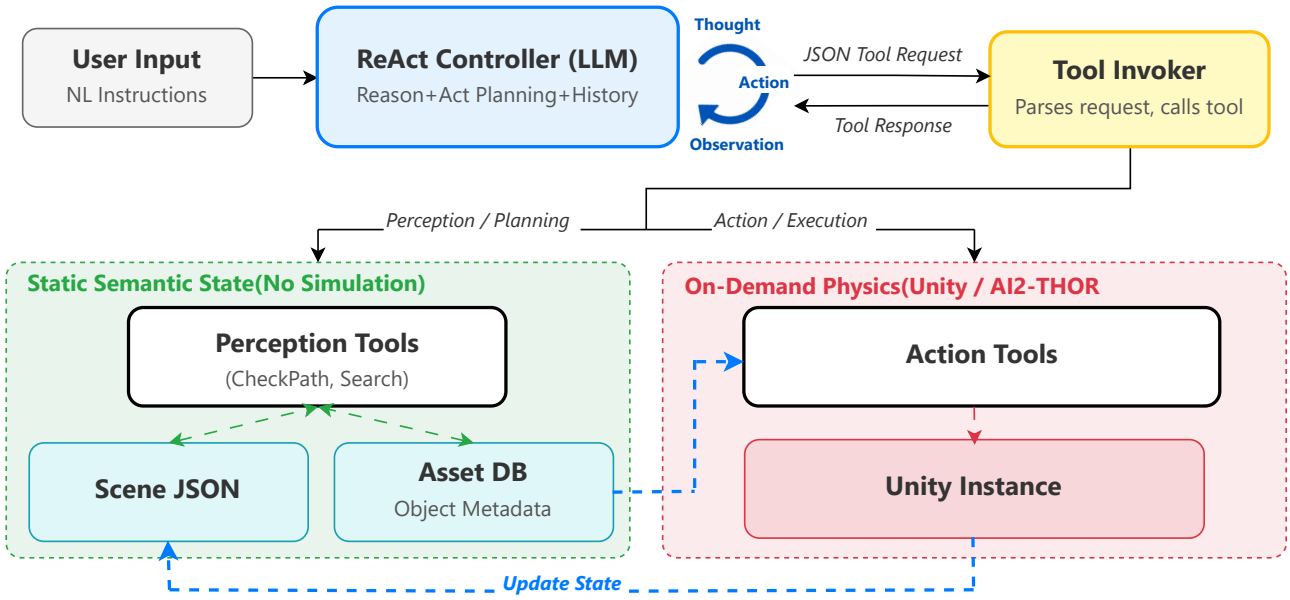


Figure 13. **System Architecture of the Task-Semantic Scene Editing Agent.** The system operates via a **ReAct Controller** (top) that iteratively plans and issues JSON tool requests. A **Tool Invoker** (middle) serves as an execution bridge, routing perception tasks to the fast **Static Semantic State** (bottom left) and action tasks to the **On-Demand Physics Engine** (bottom right). The dashed arrow highlights the **Hybrid State Management** mechanism, where physical simulation results are synchronized back to the static scene JSON to ensure consistency.

D.2.1. Perception Tools (Checking Phase)

Tool: CheckPath

Data Source JSON
Description Verifies topological connectivity and room existence to validate navigation feasibility.
Input Global Context (No specific arguments)
Logic Parses the polygon boundaries of each room and the coordinates of connecting portals (doors/stairs) from the JSON. It constructs a topological path graph to verify if a valid navigable route exists between the task's start and end locations.

Tool: SearchAssets

Data Source Asset DB
Description Retrieves new interactive objects based on natural language queries.
Input `query`, `top_k`, `properties` (optional)
Logic Utilizes the retrieval method from **Holodeck**: it employs CLIP (visual) and SBERT (semantic) embeddings to match the user's query against the asset library. It returns `assetIds` that match specific physical properties (e.g., `Pickable`).

Tool: ListObjects

Data Source JSON + Asset DB
Description Lists existing objects in the scene, supporting filtering by location and functional properties.
Input `room` (optional), `keyword/id` (optional)
Logic Iterates through the JSON to find objects. The `keyword` input supports both fuzzy name matching and exact ID lookup. It cross-references each object's `assetId` with the Asset DB to retrieve implicit functional properties (e.g., `Receptacle`, `CanOpen`) not stored in the scene file.

Tool: CheckSurface

Data Source JSON
Description Inspects what is currently placed *on top* of a specific object.
Input `keyword/id`
Logic Identifies the target object (by keyword or exact ID) and retrieves its child nodes from the JSON hierarchy. It geometrically verifies the "on-top" relationship by comparing the child's centroid height (y) against the parent object's Axis-Aligned Bounding Box (AABB) top surface.

Tool: SearchContents

Data Source JSON
Description Inspects what is currently stored *inside* a container.
Input `keyword/id`
Logic Identifies the target container and retrieves its child nodes. Unlike surface checks, it verifies the "inside" relationship by checking if the child's centroid is strictly contained within the vertical range (y_{min}, y_{max}) of the parent's AABB.

D.2.2. Action Tools (Provisioning Phase)

Action tools modify the scene. These tools automatically handle collision avoidance via an internal geometric solver before invoking native AI2-THOR actions for physical consistency.

Tool: PlaceInContainer

Data Source JSON + Asset DB + Unity
Description Physically instantiates an asset inside an openable container (e.g., placing a cola inside a fridge).
Input `asset_id`, `container_id`
Logic The system first calculates valid non-overlapping (x, z) coordinates using an internal geometric solver. It then triggers an on-demand AI2-THOR instance and executes a sequence of **native atomic actions**:

1. **`OpenObject`**: Fully opens the container to ensure accessibility.
2. **`SpawnAsset`**: Instantiates the asset (loaded from Asset DB) at the planned coordinates.
3. **`PlaceObjectAtPoint`**: Uses the physics engine to verify collisions and settle the object.

Finally, the stable pose is captured and written back to the static JSON.

Tool: RemoveObject

Data Source JSON
Description Deletes specific objects from the scene.
Input `object_id` (optional), `receptacle_id` (optional)
Logic Directly edits the JSON scene graph. If `object_id` is provided, it removes that specific node. If `receptacle_id` is provided, it recursively deletes all child nodes associated with that receptacle, effectively clearing the surface.

Tool: PlaceOnSurface

Data Source JSON + Asset DB + Unity
Description Physically instantiates an asset on an open surface (e.g., table, sofa).
Input `asset_id`, `receptacle_id`
Logic Similar to container placement, this tool uses an internal 2D bin-packing algorithm to determine the planar position. It then invokes the following atomic actions in AI2-THOR:

1. **`SpawnAsset`**: Instantiates the asset from the Asset DB.
2. **`PlaceObjectAtPoint`**: Allows the object to settle naturally under gravity.

This process ensures the object rests naturally on uneven surfaces (e.g., sofa cushions) without floating or clipping before updating the JSON.

E. Embodied Algorithms in Mansion

Here, we briefly introduce BUMBLE [39], COME-robot [50], and a variant of BUMBLE with text augmentation. These algorithms are representative embodied mobile robot systems for long-horizon navigation and manipulation tasks. BUMBLE is a whole-building framework with a VLM-driven reasoning core, and an open-world perception system, integrating parameterized navigation and manipulation skills guided by dual-layer memory for long-horizon planning and recovery [39]. COME-robot operates similarly as a closed-loop, open-vocabulary system, exposing perception and execution APIs and using GPT-4V to refine code-level plans from visual feedback iteratively, but without long-term memory [50]. We enable the global perception map of COME-robot by providing rich object information in the scene when prompting the VLM planner.

Within MANSION, we adapt the skill libraries and decision modules from both systems to our multi-floor experimental setting and evaluate their performance in terms of success rate and robustness to complex layouts. We omit intricate real-world robotic manipulation components, such as dexterous grasping, localization, and low-level motor control, and instead focus on evaluating high-level sequential decision-making for task completion. To enable richer scene interaction, we extend the systems with new skills built upon the atomic actions provided by AI2-THOR [21], allowing the agents to operate effectively in multi-floor environments. Furthermore, to enhance exploration capabilities, we introduce a rotation skill that enables the agent to reorient itself and continue searching when the target object is not initially in sight. For consistency and to balance API query time with model performance, we adopt GPT-4.1 as the VLM backbone [39]. However, a key limitation arises from the VLM's reduced object identification accuracy in

simulated environments. Although the agent can generate coherent action sequences for the retrieval and delivery of the task, it frequently misidentifies the target object, leading to task failure. To mitigate this issue, we introduce a variant of BUMBLE that exposes the object type only during skill selection, providing the agent with just enough semantic guidance to better interpret its surroundings. Importantly, the agent does not receive object-type information when executing the skills.

In single-floor tasks, the agent is required to locate an object (e.g. basketball, laptop) and deliver it to another room. In two-floor tasks, the agent is asked to first get a cloth from the first floor and deliver it to the second floor. The example is shown in Fig. 14. In the four-story setting, the agent starts on the first floor, collects an orange toolbox on the third floor, and delivers it to the fourth floor.

F. Skills in MANSION

F.1. Skill library expansion in MANSION

To better support the baseline algorithms in MANSION, we extend the original AI2-THOR skill library with three essential atomic skills required for multi-floor, long-horizon tasks: *CallElevator*, *UseElevator*, and *TakeStairs*. The detailed descriptions of these skills can be found in the following skill cards.

Skill: UseElevator

Description Selects the target floor and performs a floor transition using the elevator if a valid floor number is provided. State information transition will be processed internally.

Prerequisite

- The robot must already be positioned at the elevator entrance.
- CallElevator skill is used.

Input `target_floor`

Internal process The parameter `target_floor` is converted into a floor offset: `floor_delta = target_floor - current_floor`, which is passed to `ThorGym.update_floor(floor_delta)`. `update_floor(floor_delta)` internally:

1. Removes the currently held object from the current-floor JSON and inserts it into the target-floor JSON.
2. Calls `controller.reset(scene=...)` to reload the target floor's scene.
3. Reconstructs room landmarks and the room graph.
4. Teleports the robot to a standardized starting pose outside the elevator on the new floor, oriented outward from the elevator.

Skill: CallElevator

Description Call the elevator to the robot's current floor and open the elevator door.

Prerequisite

- The robot must be located near the elevator entrance.

Input None

Internal process Invokes the environment's elevator-calling mechanism:

1. Sends a call request to bring the elevator to `current_floor`.
2. Open the elevator door for the next step operation.

Skill: TakeStairs

Description Move the robot between adjacent floors using the staircase. This skill serves as an alternative to elevator-based floor transitions. The VLM determines whether the robot should go Up or Down based on the stair-view image.

Prerequisite

- The robot must be positioned at the staircase entrance.
- The direction must correspond to an available adjacent floor.

Input `direction` $\in \{\text{Up}, \text{Down}\}$

Internal process The parameter `direction` is mapped to a floor offset: `floor_delta` $= \begin{cases} +1 & \text{if direction = Up,} \\ -1 & \text{if direction = Down.} \end{cases}$ This `floor_delta` is passed to `ThorGym.update_floor(floor_delta)`.

F.2. Progress Score

We decompose task completion into two components: correct object retrieval and successful navigation, as described in Section 4.2. This separation reflects a key limitation of current VLMs: they struggle to reliably identify and retrieve small objects, even though they possess a stronger, more global understanding of room layout and spatial context. Therefore, in addition to reporting overall success rates, we also evaluate performance using a progress score that captures partial task completion.

F.3. Task details

We now provide the detailed prompts that we used in the Table 6.

Some sample test environments that we used can be found in the following Fig. 15–17.

F.4. Algorithms implementation details

In integrating the embodied algorithms into MANSION, we introduce several key adaptations to better reflect the robot's

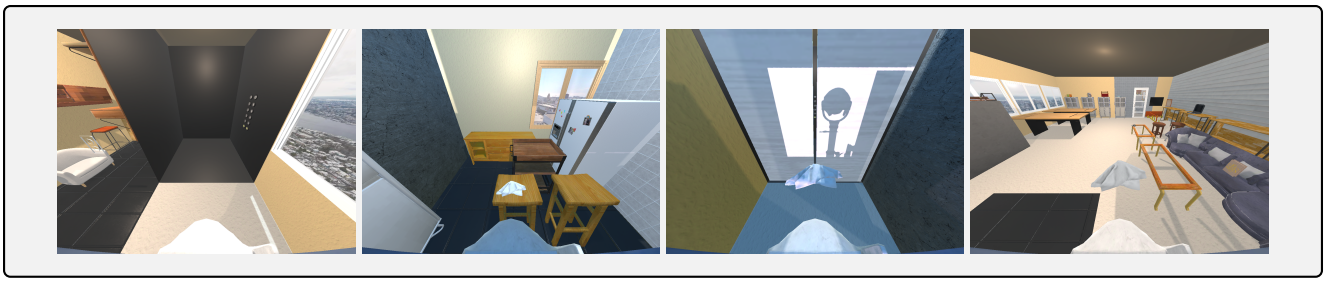


Figure 14. Sample screenshots from a task execution. The robot begins on the second floor, takes the elevator to the first floor to retrieve a cloth, and then returns to the sofa. *Task query: I want to clean my sofa. Go get a cloth from the first floor and come back near the sofa.*

Table 6. Task Settings and Prompts

Environment	Prompt
Single-floor apartment	Find a box on the bed and bring it to the bathroom.
Double-floor office	Find a cellphone on a blue couch on the first floor and bring it to the round table on the second floor.
Four-floor office	Go to the third floor, find a laptop on the desk in the meeting room on the third floor, and take it to the restroom on the fourth floor.

Figure 15. Single-floor apartment layout.

actual capabilities within the simulated environment.

GoToLandmark: The success of embodied navigation algorithms depends heavily on the VLM's ability to obtain reliable visual observations of different rooms. The robot can only plan a route to the correct destination if the VLM correctly identifies the room type. However, in the original BUMBLE implementation, each room is represented by a single image. When that image happens to capture a featureless or uninformative part of the room, the robot's failure rate increases significantly. To address this, we provide panorama views for each room, giving the robot a more complete and informative visual representation of the environment and improving its ability to recognize the correct room. An example can be found in Fig. 18. To balance image granularity with the input constraints of VLMs, each room's panorama is constructed by concatenating three images captured at yaw angles of 0°, 120°, and 240°.

Figure 16. Two-floor office layout.

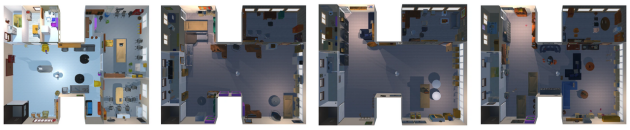


Figure 17. Four-floor building layout.

Figure 18. Panorama view of the service shaft room.

UseElevator: The agent is informed of its current floor and given a visual observation that includes the elevator button panel. It must identify the valid floor numbers from the panel and select the target floor it intends to reach.

TakeStairs: The agent is told of its floor and provided with the corresponding visual observation. To prevent invalid decisions, such as attempting to go downstairs from

the first floor, we overlay valid directional arrows onto the visual input, ensuring the agent is guided toward only feasible movement options.

F.5. Failure Case Analysis

In this subsection, we analyze several representative task failure cases and their underlying causes.

Failure Case Analysis 1: Two-floor task. A typical failure pattern is as follows: the agent navigates into a corner and, even after attempting to backtrack and rotate, still cannot escape from the corner, eventually exhausting the step budget and failing the task. See in Fig. 19

Figure 19. Failure case 1

Failure Case Analysis 2: Four-floor task. The most prominent issue is that `goto_landmark` needs to stitch all landmarks into a single long image as input to the VLM. However, in the four-floor building, there are too many landmarks, so the stitched image must be heavily downsampled when resized to the VLM input resolution, causing severe information loss and making it difficult for `goto_landmark` to function effectively. See in Fig. 20

Figure 20. Failure case 2

G. Object Placement

For complex rooms, the key challenge lies not only in accommodating a larger number of objects, but also in preserving regular global distribution under dense and repeated furniture patterns. A purely instance-level object placement algorithm tends to over-emphasize local relations, which can overfill one part of a large room while leaving other feasible regions unused. We shift our planning focus from individual objects to structured groups before solving the geometry.

We require the LLM to output object-level constraints for each item, including three types of constraints: global placement constraints (e.g., `edge`, `middle`, or `unconstrained`), structural constraints (e.g., `single`, `matrix`, or `paired`), and optional relative position constraints (e.g., `near`, `far`, etc.). The `matrix` primitive compactly represents repeated rows such as desk rows or bookshelf blocks, while `paired` expresses one-to-one accessory relations such as desk–chair pairs. This representation reduces the burden on the LLM, since in large spaces such as classrooms, libraries, or offices, it no longer needs to output dozens of nearly identical instance-level constraints. We then normalize these constraints into groups $G = (a, M)$, where a denotes the anchor object and M denotes the member set. The anchor carries the global spatial role of the group, while the remaining members are placed in the anchor's local frame.

Algorithm 4 Priority-aware group-based object placement

Require: Room polygon Ω; normalized object groups $\mathcal{G}$; placement constraints $\mathcal{C}$
Ensure: Placement set $\mathcal{P}$

Sort $\mathcal{G}$ by the constraints of $a(G)$: `edge+matrix` $\succ$ `edge` $\succ$ `matrix` $\succ$ `middle` $\succ$ `free`
$\mathcal{P} \leftarrow \emptyset$
for each group G in $\mathcal{G}$ **do**
  if $a(G)$ has a `matrix` constraint **then**
    $(r, c) \leftarrow$ requested matrix size of $a(G)$
    $\mathcal{Q} \leftarrow \emptyset$
    while $r \geq 1$ and $c \geq 1$ and $\mathcal{Q} = \emptyset$ **do**
      $\hat{o} \leftarrow$ BUILDMACROOBJECT(G, r, c)
      $\mathcal{Q} \leftarrow$ FINDFEASIBLEPLACEMENT$(\hat{o}, \Omega, \mathcal{P}, \mathcal{C})$
      if $\mathcal{Q} = \emptyset$ **then**
        $(r, c) \leftarrow$ DOWNGRADEMATRIX(r, c)
      end if
    end while
    if $\mathcal{Q} \neq \emptyset$ **then**
      $\mathcal{P} \leftarrow \mathcal{P} \cup \mathcal{Q}$
    end if
  else
    for each object o in G **do**
      $\mathcal{Q} \leftarrow$ FINDFEASIBLEPLACEMENT$(o, \Omega, \mathcal{P}, \mathcal{C})$
      if $\mathcal{Q} \neq \emptyset$ **then**
        $\mathcal{P} \leftarrow \mathcal{P} \cup \mathcal{Q}$
      end if
    end for
  end if
end for
return $\mathcal{P}$

We present our object placement algorithm in Algorithm 4. Our algorithm follows a priority-aware constructive search. Groups are sorted according to the constraints of their anchor object, yielding a strict priority order. Groups that are both wall-dependent and highly structured are processed first, since they occupy the most constrained regions of the room and strongly affect later circulation. For a matrix group, the solver first places the whole pattern as a macro object; if no feasible placement is found, it progressively downgrades the matrix size and retries. For a non-matrix group, objects are processed sequentially within the group, starting from the anchor object. For each object, the solver samples candidate positions and filters them by hard constraints including collision checking,

constraint consistency, and incremental reachability. Objects that do not admit a feasible placement are discarded, while the solver continues with the remaining objects.

Reachability is evaluated on the remaining free space when searching for feasible positions, ensuring that the room entrance remains connected to the required circulation areas and to accessible interaction zones around the placed objects. Candidates that block passages or destroy walkable structures are discarded immediately. As shown in the Fig. 21, our method maintains full reachability while preserving a high object placement count.

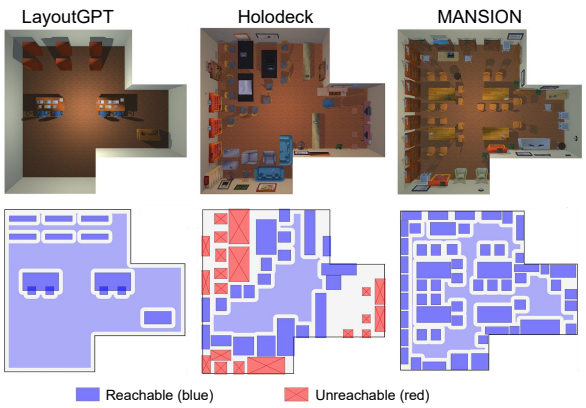


Figure 21. Reachability visualization in a library scene.

H. User Study

To understand how real-person users perceive our generated scenes compared to other methods, we conducted a comprehensive user study with 52 participants from different backgrounds. The full list of participating institutions can be found in Section 5, and we thank them again for their input. For each scene type, we randomly sample two cases from the 10 generated results with the same prompt for subjective evaluation. In each scene setting, participants are presented with one set of images from the three methods and are asked to select the best method in terms of realism, diversity, and overall layout quality. To prevent bias, we made sure that the recruited participants had no prior experience or exposure to 3D scene generation. Furthermore, we kept the names of the corresponding algorithms hidden from them throughout the survey. A sample of the user study image and form can be seen in Fig. 22.

We also provide the metric instructions that we used in the survey to guide the users to rank the different methods below.

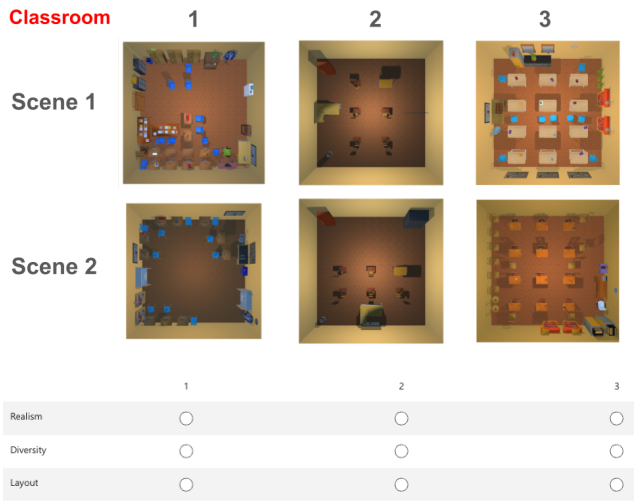


Figure 22. Sample from the user study for the classroom scene category.

Instructions

- Realism: How realistic and plausible the scene is (object choices and placements make sense in the given setting; no obvious weird placements or impossible arrangements). Which one is most realistic to real life?
- Diversity: How different and varied the generated scenes are across different generated scenes of the SAME method (variety in object selection, arrangement, and overall design, while still matching the scenario)
- Layout: How well-organized and functional the spatial arrangement is (clear structure, reasonable spacing, good flow, and sensible grouping of objects).

I. Prompt Templates

We present prompt templates for three representative modules: (i) whole-building program planning, (ii) single-floor topology (bubble graph) generation, and (iii) LLM-guided seed box selection for cutting. For readability, these templates preserve the core task definition, input fields, output schema, and major constraints, while abstracting away some implementation-level details. The templates are shown in Fig. 23–25.

Prompt Template: Whole Building Planning

Module: `building_program_planner`
System Instruction: You are an experienced architect asked to program a complete multi-floor building using the first-floor plan as reference.
Inputs:
- **Visual:** *[Image Input: Base64 rasterized boundary]*
- **Geometry:** First-floor boundary JSON
- **Params:** Floors: {`floors`}, {`area_note`}
- **Requirement:** {`user_part`}

Core Placement Preferences:
- Place stairs/elevators only in corners that are bounded by two exterior walls; pick the corner whose surrounding leftover space is smallest to keep large continuous areas intact.
- Make use of tight/awkward leftover pockets.
- Quantize core boxes to integer coordinates and size exactly {`core_area`}. Coordinates must be non-negative.

What to Do:
1. Analyze the outline and area and reason about a practical building function.
2. Decide the vertical connectivity method: stair | elevator | stair_and_elevator, consistent with the rule.
3. Choose locations for vertical cores; each stair/elevator occupies an axis-aligned {`core_area`} bbox within the floor polygon. Output as x=[x1,x2], y=[y1,y2] with integer coordinates.
4. For each floor, produce a list "rooms" with ID-only entries, area estimates, and material specifications. **Do NOT include a "type" field.** The **first room** in the list MUST be the circulation hub of the floor (traffic core): choose logically based on building type. Each room must include `floor_material` and `wall_material` fields with descriptive text.
5. Room sums must not exceed the gross floor area; keep 12–25% as circulation/core reserve unless justified.
6. Ensure totals are reasonable; indicate whether plans fit within GFA.
7. If a floor layout should be exactly the same as another floor, use a shorthand: specify only {`"index": k, ``copy": j`}.

Output Schema (Strict JSON):

```
{
  "reasoning": "<Brief explanation of program and area logic>",
  "vertical_connectivity": {
    "method": "stair | elevator | stair_and_elevator",
    "cores": [ {"type": "stair", "x": [<int>, <int>], "y": [<int>, <int>]} ],
    "justification": "<Why this choice fits rules and requirements>"
  },
  "floors": [
    {
      "index": <int>,
      "requirement": "<Natural language requirement for this floor>",
      "gross_floor_area": <float>,
      "rooms": [
        {
          "id": "hub_room", "area_estimate": <float>,
          "floor_material": "<Description (e.g., warm oak hardwood, matte)>",
          "wall_material": "<Description (e.g., soft beige drywall, smooth)>",
          "notes": "circulation hub (put FIRST)"
        },
        {
          "id": "<other_room_id>", "area_estimate": <float>,
          "floor_material": "<Description>", "wall_material": "<Description>",
          "notes": "<Optional>"
        }
      ],
      "area_summary": {
        "sum_rooms": <float>, "reserve_ratio": <float>,
        "fits_within_gfa": <boolean>, "notes": "<Optional notes>"
      }
    }
  ]
}
```

Floor Layout Context: {`layout_json`}

Figure 23. Prompt template for whole-building program planning.

Prompt Template: Single-Floor Topology Generation

Module: `topology_bubble_planner`
System Instruction: You are an experienced architect designing the abstract topological connectivity of a single floor.
Inputs:
- Overall program reasoning: {`reasoning`}
- Floor context: Floor index {`idx`}, Gross floor area {`gfa`} m^2, Floor requirement {`requirement`}
- Floor polygon JSON (main space after cores removed): {`layout_json`}
- Vertical cores: {`vtext`}
- Floor hints from program: {`rooms_json`} (first item is the suggested circulation hub)
- Material selection guidance: {`material_hints_text`}

Your Task:
- Derive a minimal useful set of rooms/spaces for this floor based on the requirement and rooms list, and assign each node an estimated area (m^2); capture only abstract connectivity, not geometry.
- Include all elevator/stair connectors indicated by the floor layout as fixed nodes for this floor; do not omit them.
- Treat the provided rooms list as hints (the first item is the suggested circulation hub), but re-evaluate which space should serve as `main` if one space clearly dominates the floor.

Output Requirements:
- Return **exactly one JSON object** with two top-level fields: `nodes` and `edges`, with no extra explanatory text.
- **Required node fields:** `id`, `type`, `area`, `floor_material`, `wall_material`, `open_relation`.
- `Node types` allowed: `main`, `Entities`, `area`, `elevator`, `stair`.
- `Edge kinds` allowed: `access`, `adjacent`.
- Every node must include `floor_material` and `wall_material` using descriptive text.
- `open_relation` must be either "open" or "door". For `main`, use "open"; for `elevator` and `stair`, use "door".

Design Preferences: {`node hierarchy & branching`}, {`area vs. Entities selection`}, {`open_relation assignment`}
Output Schema:

```
{
  "nodes": [
    {"id": "lobby", "type": "main", "area": 80.0,
     "floor_material": "warm oak hardwood, matte",
     "wall_material": "soft beige drywall, smooth", "open_relation": "open"},
    {"id": "office_zone", "type": "area", "area": 30.0,
     "floor_material": "warm oak hardwood, matte",
     "wall_material": "soft beige drywall, smooth", "open_relation": "door"},
    {"id": "room_1", "type": "Entities", "area": 15.0,
     "floor_material": "carpet, neutral gray",
     "wall_material": "painted drywall, white", "open_relation": "door"}
  ],
  "edges": [
    {"source": "lobby", "target": "office_zone", "kind": "adjacent"},
    {"source": "office_zone", "target": "room_1", "kind": "adjacent"}
  ]
}
```

Figure 24. Prompt template for single-floor topology generation.

Prompt Template: Hierarchical Seed Planning

Module: `seed_guidance`

System Instruction: You are a floor plan seed planning assistant. Given the floor topology and a preview image containing the full floor outline with stairs/elevators, your task is to plan an axis-aligned rectangular bounding box for each target room within the current parent room, expressed as `x=[xmin,xmax], y=[ymin,ymax]` to indicate each room's approximate position and extent.

Context provided:

- Target room list: {`target_ids_text`}
- Parent room ID: {`parent_id`}, Type: {`parent_type`}, Area: {`parent_area`} m^2
- Adjacent room IDs (already placed): {`neighbor_ids_text`}
- Project requirement: {`requirement`}
- Topology JSON and area hints per room
- {special_instruction}

Coordinate convention: x increases to the right, y increases upward; the parent room's approximate coordinate range is $x \in [\{\texttt{minx}\}, \{\texttt{maxx}\}]$, $y \in [\{\texttt{miny}\}, \{\texttt{maxy}\}]$; your output should use values within this coordinate interval.

Output format: The output must be a JSON array where each element contains:

- `room_id`: string, the room ID (must be chosen from the candidate list only)
- `x`: array of length 2 `[xmin,xmax]`, the bounding box in the x direction, requires $\texttt{xmin} \leq \texttt{xmax}$
- `y`: array of length 2 `[ymin,ymax]`, the bounding box in the y direction, requires $\texttt{ymin} \leq \texttt{ymax}$
- `area`: float, the approximate fraction of the parent room's area this room occupies (0.0–1.0, optional, used as a downstream hint)
- `reason`: a brief one-sentence explanation of why you placed this room at this position

Output Schema (Strict JSON):

```
[
  {"room_id": "lobby", "x": [0.0, 8.0], "y": [0.0, 6.0], "area": 0.45,
   "reason": "Main lobby occupies the central area, enclosing the stair core."},
  {"room_id": "office_1", "x": [8.0, 12.0], "y": [0.0, 5.0], "area": 0.2,
   "reason": "Office placed at the east wing, away from the stair core."}
]
```

Please refer to the floor outline and existing stair/elevator positions in the preview image to provide a reasonable bounding box allocation. Note: bounding boxes should be placed as much as possible within the parent room outline; avoid large-area overlaps.

Figure 25. Prompt template for LLM-guided seed box planning for cutting.