InfiniBench: Infinite Benchmarking for Visual Spatial Reasoning with Customizable Scene Complexity

Haoming Wang, Qiyao Xue and Wei Gao
University of Pittsburgh
{hw.wang, qiyao_xue, weigao}@pitt.edu

A. In-Context Learning for Scene Constraint Generation

In our design of InfiniBench, the LLM takes the following inputs: the user's scene description in natural language, the given descriptions about the list of available assets and constraint APIs (determined by the specific procedural generation framework being used). Besides, to ensure the generated scene constraints are machine-readable and physically plausible, in-context examples are also provided to the LLM to demonstrate the specific syntax of the underlying constraint language used in the procedural generation framework. In practice, we use the default home constraints which are manually crafted as the in-context examples. Some more specific examples, with respect to the constraint APIs and syntax being used in the Infinigen [1] framework, are provided below.

A.1. Example 1: Dining Rooms

The following code block demonstrates how to define scene constraints for a dining room, including furniture relationships (chairs around tables) and clutter distribution (dishware and utensils).

```
# region DININGROOMS

diningtables = furniture[Semantics.Table][tables.TableDiningFactory]
diningchairs = furniture[Semantics.Chair][seating.ChairFactory]
constraints["dining_chairs"] = rooms.all(
    lambda r: (
        diningtables.related_to(r).all(
            lambda t: (
                diningchairs.related_to(r)
                .related_to(t, cu.front_against)
                .count()
                .in_range(3, 6)
            )
        )
    )
)

score_terms["dining_chairs"] = rooms.all(
    lambda r: (
        diningchairs.related_to(r).count().maximize(weight=5)
        + diningchairs.related_to(r)
        .mean(lambda t: t.distance(diningchairs.related_to(r)))
        .maximize(weight=3)
        # cl.reflectional_asymmetry(diningchairs.related_to(r), diningtables.related_to(r)).minimize(weight=1)
        # cl.rotational_asymmetry(diningchairs.related_to(r)).minimize(weight=1)
    )
)

constraints["dining_table_objects"] = rooms.all(
    lambda r: (
        diningtables.related_to(r).all(
            lambda t: (
                obj[Semantics.TableDisplayItem]
                .related_to(t, cu.ontop)
                .count()
                .in_range(0, 2)
                * (obj[Semantics.Utensils].related_to(t, cu.ontop).count() >= 0)
                * (
                    obj[Semantics.Dishware]
                    .related_to(t, cu.ontop)
                    .count()
                    .in_range(0, 2)
                )
            )
        )
    )
)

score_terms["dining_table_objects"] = rooms.mean(
    lambda r: (
        cl.center_stable_surface_dist(
            obj[Semantics.TableDisplayItem].related_to(
                diningtables.related_to(r), cu.ontop
            )
        ).minimize(weight=1)
    )
)

diningrooms = rooms[Semantics.DiningRoom].excludes(cu.room_types)
constraints["diningroom"] = diningrooms.all(
    lambda r: (
        (diningtables.related_to(r).count() == 1)
        * storage.related_to(r).all(
            lambda t: (
                (obj[Semantics.Dishware].related_to(t, cu.on).count() >= 0)
                * (
                    obj[Semantics.OfficeShelfItem]
                    .related_to(t, cu.on)
                    .count()
                    .in_range(0, 5)
                )
            )
        )
    )
)
score_terms["diningroom"] = diningrooms.mean(
    lambda r: (
        diningtables.related_to(r).distance(r, cu.walltags).maximize(weight=10)
        + cl.angle_alignment_cost(
            diningtables.related_to(r), r, cu.walltags
        ).minimize(weight=10)
        + cl.center_stable_surface_dist(diningtables.related_to(r)).minimize(
```

```
            weight=1
        )
    )
)
# endregion
```

Listing 1. An in-context example for Dining Room constraints provided to the LLM agent.

A.2. Example 2: Bathrooms

The following example illustrates the logical constraints for bathroom layouts, specifically enforcing the presence of essential hardware (toilets, sinks) and their spatial accessibility.

```
# region BATHROOMS
bathrooms = rooms[Semantics.Bathroom].excludes(cu.room_types)

toilet = wallfurn[bathroom.ToiletFactory]
bathtub = wallfurn[bathroom.BathtubFactory]
sink = wallfurn[bathroom.StandingSinkFactory]

hardware = obj[bathroom.HardwareFactory].related_to(bathrooms, cu.against_wall)

constraints["bathroom"] = bathrooms.all(
    lambda r: (
        mirror.related_to(r).related_to(r, cu.flush_wall).count().equals(1)
        * sink.related_to(r).count().equals(1)
        * toilet.related_to(r).count().equals(1)
        * storage.related_to(r).all(
            lambda t: (
                obj[Semantics.BathroomItem].related_to(t, cu.on).count() >= 0
            )
        )
    )
)

score_terms["toilet"] = rooms.all(
    lambda r: (
        toilet.distance(doors).maximize(weight=1)
        + toilet.distance(furniture).maximize(weight=1)
        + toilet.distance(sink).maximize(weight=1)
        + cl.accessibility_cost(toilet, furniture, dist=2).minimize(weight=10)
    )
)

constraints["bathtub"] = bathrooms.all(
    lambda r: (
        bathtub.related_to(r).count().in_range(0, 1)
        * hardware.related_to(r).count().in_range(1, 4)
    )
)
score_terms["bathtub"] = bathrooms.all(
    lambda r: (
        bathtub.mean(lambda t: t.distance(hardware)).minimize(weight=0.2)
        + sink.mean(lambda t: t.distance(hardware)).minimize(weight=0.2)
        + hardware.mean(
            lambda t: (
                t.distance(rooms, cu.floortags).hinge(0.5, 1).minimize(weight=15)
            )
        )
    )
)

score_terms["bathroom"] = (
    mirror.related_to(bathrooms).distance(sink).minimize(weight=3)
) + cl.accessibility_cost(mirror, furniture, cu.down_dir).maximize(weight=3)
# endregion
```

Listing 2. An in-context example for Bathroom constraints provided to the LLM agent.

B. CoT Reasoning for Refinement of Scene Constraints

This section details the iterative feedback loop employed to resolve the conflicts in the generation of scene constraints. The refinement process operates as follows:

1. **Execution:** The layout optimizer attempts to generate the scene based on the current set of procedural constraints.
2. **Error Reporting:** Upon the completion of a fixed number of optimization steps, a structured error report is synthesized from the optimization metadata. This report comprises:
   - *Textual Summary:* A list of constraints that remained unsatisfied during the optimization process.
   - *Visual Metadata:* A semantic 2D BEV (Bird's-Eye View) segmentation map annotated with object labels to visualize spatial conflicts.
3. **Refinement:** Based on the user's original scene description and the generated error report, the LLM evaluates whether the user's requirements have been fully met. If unsatisfied, the LLM employs Chain-of-Thought (CoT) reasoning to analyze the failure and output revised constraints. In practice, we use the thinking mode of Gemini-2.5-Pro to enable CoT reasoning.

A detailed end-to-end example of such CoT reasoning, with the constraint APIs and syntax provided by the Infinigen framework [1], is provided in Figure 1.

C. Details of Cluster-Based Optimization

To address the limitations of the "rigid hierarchy" that is prevalent in traditional procedural generation frameworks, in InfiniBench we implemented a robust cluster-based optimization strategy.

C.1. Cluster Definition and Identification

We define the **root** of a cluster as an object situated at the highest level of the hierarchy (typically large furniture, such as tables or desks). Objects at lower hierarchy levels are designated as **child** objects if they maintain a "stable against" spatial relationship with a parent object. Information regarding these clusters is dynamically retrieved and updated from the global "scene state" throughout the optimization process.

C.2. Augmented Action Space

In standard rigid-hierarchy optimization, the action space for individual objects comprises addition, resampling, deletion, translation, rotation, relation plane changes, relation

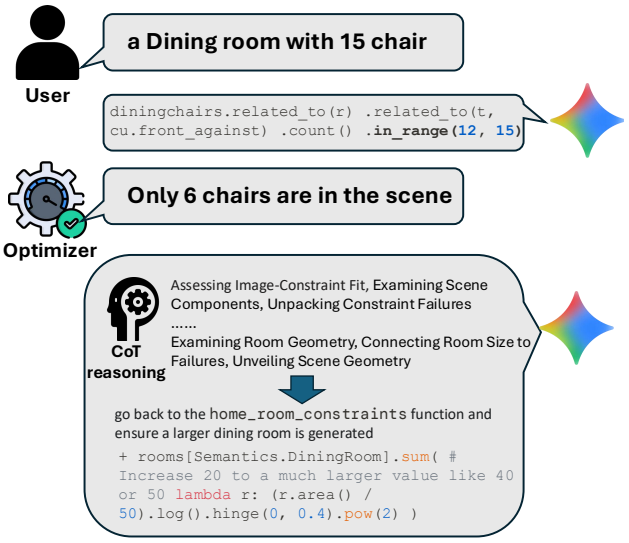

Figure 1. An example of scene constraint refinement via CoT reasoning

target changes, and swapping. To facilitate cluster-based optimization, we augment this space by adding the following cluster-specific operations:

- `resample_cluster`: Resamples the parameters for the cluster's root object, triggering a cascade update for child objects.
- `translation_cluster`: Translates the entire cluster as a single entity while strictly preserving the relative positions of all constituent objects.
- `rotation_cluster`: Rotates the cluster around the geometric center of the root object, maintaining the relative orientation and position of all child objects.

C.3. Collision Checking

Following each operation, the optimizer performs a collision detection and checking routine. If a collision is detected, the operation is immediately reverted to maintain physical plausibility. For cluster-level operations, collision checks are performed using the cluster's collective bounding box (or the aggregate of all component meshes) to ensure the entire group is placed validly without intersecting other scene elements.

D. Gallery of Generated 3D Scenes

We provide additional examples of the 3D scenes generated by InfiniBench, demonstrating its versatility across various room types and levels of scene complexity. The scenes with different levels of compositional complexity are shown in Figures 2-4, and the scenes with different levels of relational complexity are shown in Figures 5-7. In addition, in Figure 8 and Figure 9, we also provided examples of the optimized camera trajectories, along with the scene frames at the selected waypoints, with different levels of observational complexities. Note that, in addition to InfiniBench's native scene-generation pipeline, we also leverage the room-shape generation module from Infinigen to produce more realistic renderings of these example scenes.

E. Details of the Camera Trajectory Optimization Algorithm

To ensure that the generated camera trajectory simulates a natural, human-like perspective while maximizing the visual coverage of task-related objects, we constrain the optimization problem to a 4-DOF space. We fix the camera's vertical height (z) to approximate a standard eye level and restrict the roll angle (θ_x) to zero to maintain view stability. Consequently, the optimization is performed over the remaining four degrees of freedom: the horizontal position coordinates (x, y) and the rotational angles for yaw (θ_y) and pitch (θ_z). This configuration allows the camera to freely navigate accessible areas on the horizontal plane and adjust its viewing angle to effectively frame relevant spatial features.

Optimization Workflow. As shown in Algorithm 1, our method follows an iterative workflow that progressively builds a path to view every task-related object in the scene. The process is as follows:

0. Scene Preprocessing: Before optimization, the generated scene file (e.g., a `.blend` file) is preprocessed to extract the necessary geometric and semantic metadata. Using Blender's `Bmesh` API, we extract the 3D mesh of every object. These meshes are then triangulated and converted into `Trimesh` objects, which facilitates efficient geometric queries. A 2D floor plan is also extracted to define the navigable area for the camera.

1. Target Selection (find the closest target object): The algorithm begins at a predefined starting point, typically a door. It maintains a list of "unvisited" objects. In each iteration, it identifies the object from this list that is closest to the camera's current position, designating it as the next target.

2. Viewpoint sampling: A set of candidate viewpoints is randomly sampled in the vicinity of the current target object. Each viewpoint is a 4-DOF pose that is oriented towards the object.

3. Viewpoint selection: Each candidate viewpoint is evaluated against a series of physical constraints to determine its validity. The best viewpoint is one that satisfies all the following criteria:

- *Accessibility*: The camera's (x, y) location must be within the accessible area of the 2D floor plan (i.e., inside the room but outside the bounding boxes of all objects).
- *Field of View (FOV)*: The target object must be fully visible within the camera's viewing frustum from the candidate pose.

Figure 2. Examples of generated **living rooms** with varying levels of **compositional complexity**, measured by the number of objects. 'Low' indicates seating for fewer than five people; 'Medium' indicates seating for five to ten people; and 'High' indicates seating for more than ten people. The user's prompt to the LLM agent is: "Generate a living room with seating for ** people."

Figure 3. Examples of generated **bedrooms** with varying levels of **compositional complexity**, measured by the number of objects. 'Low' indicates only one shelf; 'Medium' indicates 2-8 shelves; and 'High' indicates seating for more than 10 shelves. The user's prompt to the LLM agent is: "Generate a bedroom with ** shelves."

- *Occlusion Check*: To ensure a clear line of sight, we perform an occlusion check. A segmentation map is rendered from the viewpoint using a fast off-screen renderer (`pyrenderer`). By projecting the 3D mesh of the target object and comparing it against the segmentation map's depth buffer, we calculate the percentage of the object that is occluded by other geometry. A viewpoint is only considered valid if the occlusion is below a predefined threshold.

The candidate that best satisfies these conditions (e.g., has the lowest occlusion) is selected as the next keyframe in the trajectory.

4. Path planning: Once a valid viewpoint is selected, we use Dijkstra's algorithm to compute the shortest, collision-

Figure 4. Examples of generated **dining rooms** with varying levels of **compositional complexity**, measured by the number of objects. 'Low' indicates seating for fewer than five people; 'Medium' indicates seating for five to ten people; and 'High' indicates seating for more than ten people. The user's prompt to the LLM agent is: "Generate a dining room with seating for ** people."

Figure 5. Examples of generated **living rooms** with varying levels of **relational complexity**, measured by the room occupancy ratio. 'Low' indicates an occupancy rate below 10%; 'Medium' indicates 10–50%; and 'High' indicates an occupancy rate above 50%. The user's prompt to the LLM agent is: "Generate a living room with with occupancy rate of **."

free path on the 2D floor plan from the camera's current position to the new target viewpoint.

5. Iteration: The camera is moved along the planned path to the new viewpoint, the target object is marked as "visited," and the process repeats from Step 1. This loop continues until all objects in the scene have been successfully captured by at least one viewpoint. The final output is a complete, smooth camera trajectory composed of a sequence of optimal viewpoints connected by navigable paths, ensuring a comprehensive visual record of the scene.

F. Details of InfiniBench Implementation

As illustrated in Figure 10, we employ the Blender Cycles engine, which utilizes path tracing, to achieve photoreal-

Figure 6. Examples of generated **bedrooms** with varying levels of **relational complexity**, measured by the room occupancy ratio. 'Low' indicates an occupancy rate below 10%; 'Medium' indicates 10–50%; and 'High' indicates an occupancy rate above 50%. The user's prompt to the LLM agent is: "Generate a bedroom with with occupancy rate of **."

Figure 7. Examples of generated **dining rooms** with varying levels of **relational complexity**, measured by the room occupancy ratio. 'Low' indicates an occupancy rate below 10%; 'Medium' indicates 10–50%; and 'High' indicates an occupancy rate above 50%. The user's prompt to the LLM agent is: "Generate a dining room with with occupancy rate of **."

istic frame rendering. A critical parameter in this process is the number of path-tracing samples. While 8192 samples are typically recommended to produce noise-free images (as shown in Figure 11), this setting is computationally expensive, requiring approximately 5 minutes per frame. To optimize for efficiency without compromising quality, we adjust the sample count to 32 and enable denoising. As shown in Figure 10, after reducing the number of samples and enabling such denoising, the rendered frames appear nearly identical to the originals with 8192 samples, with only some fine textures slightly blurred. The specific rendering configuration is detailed below (unlisted parameters remain at default values).

- `num_samples`: 128

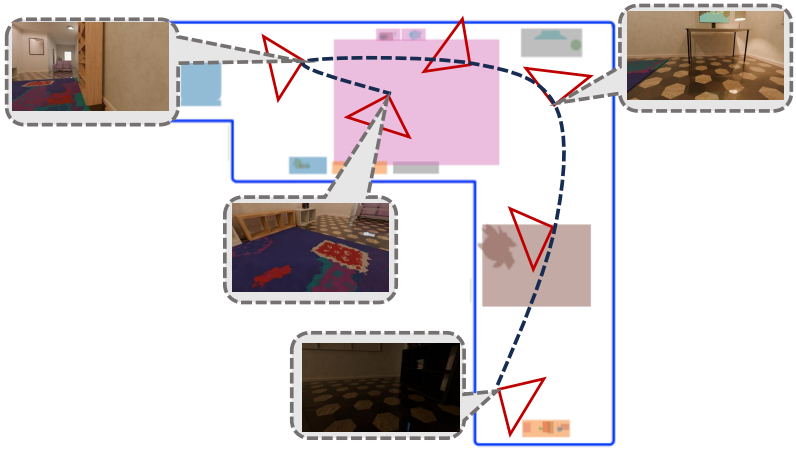

Figure 8. Visualization of the camera trajectory under a specific level of observational complexity (partial occlusion). The user's prompt to the LLM agent is: "Generate a typical living room. The camera should capture objects with partial occlusion."

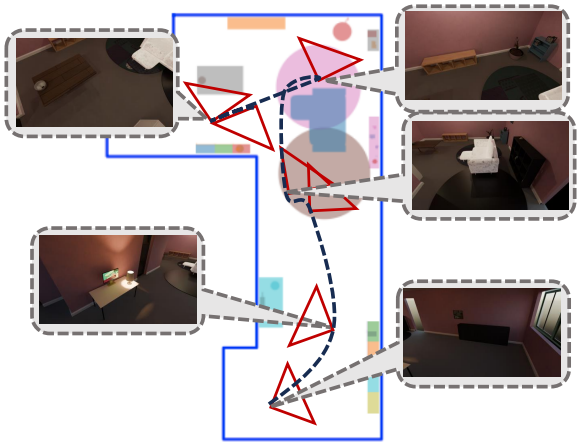

Figure 9. Visualization of the camera path under a specific level of observational complexity (bird-eye view, camera height is 2m). The user's prompt to the LLM agent is: "Generate a typical living room. The camera should capture height should be 2m"

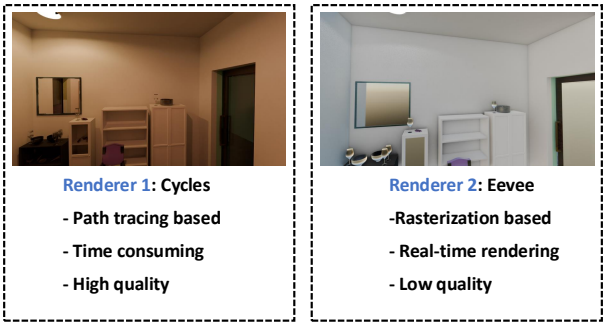

Figure 10. Comparison of image quality across different renderers.

- `adaptive_threshold`: 0.01
- `denoise`: True
- `exposure`: 1

G. Additional Qualitative Comparisons

As shown in Figure 12, we present additional examples comparing scenes generated by InfiniBench against baseline approaches. To ensure a fair comparison, all methods utilize the same 3D asset library and an identical square-shaped room plan. For baselines implemented in disparate engines, we exported the scene layouts and rendered them using Blender with identical settings to maintain visual consistency. For baselines incapable of processing natural language input directly, we employed a constraint generation method similar to Infinigen, iteratively refining the con-

Algorithm 1 Camera Trajectory Optimization

Require: Scene, MaxSamplingTimes, MaxDistance, Threshold, RequiredRange

CamParams ← GETCAMERAPARAMETERS
Objects ← EXTRACTOBJECTMESHES(*Scene*)
Targets ← GETCAPTURELIST
CurrentView ← GETDOORPOSITION
TotalPath ← [ ]
RemainingTargets ← COPY(*Targets*)
while ¬ISEMPTY(*RemainingTargets*) **do**
    target ← FINDCLOSESTTARGET(*CurrentView*) // 1. Target selection
    for $i = 1$ to *MaxSamplingTimes* **do**
        view ← SAMPLEVIEWPOINT(*target*, *MaxDistance*) // 2. Viewpoint sampling
        if ¬CHECKCOLLISION(*view*, *Targets*) **then** // 3. Viewpoint selection
            FOV ← COMPUTEFOV(*view*, *CamParams*)
            FOVPercent ← CHECKFOV(*FOV*, *target*)
            if *FOVPercent* > *Thr* **then**
                SegMap ← RENDERER(*view*, *Scene*)
                Occ ← COMPUTEOCCLUSIONRATE(*SegMap*, *target*)
                if *Occ* ∈ *RequiredRange* **then**
                    Path ← PLANPATHDIJKSTRA(*CurrentView*, *view*, *scene*) // 4. Path planning
                    ADD(*TotalPath*, *Path*)
                    CurrentView ← *view*
                    break
                end if
            end if
        end if
    end for
    REMOVE(*RemainingTargets*, *target*) // 5. Proceed to the next closest target
end while
return *TotalPath*

straints until the program became executable.

H. Mitigating the Impact of Ambiguity in Object Description

We mitigated object description ambiguity through task-specific designs. **Perspective-taking tasks** rely on counting, which is immune to instance-level ambiguity. For **measurement tasks**, we employ detailed text descriptions, including materials and spatial relationships, to uniquely identify the target object. For scenarios where multiple identical objects appear in similar contexts, we rely on the fact that identical assets inherently possess the same physical dimensions (e.g., height). Consequently, the ground truth answer is valid for any instance of that class. For **spatiotemporal tasks**, similarly, we rely on detailed descriptions to ground targets. For identical objects in the same category, we eliminate the impact of ambiguity by selecting only the first object appeared as the target, for the appearance sequence in the VQA task's ground truth.

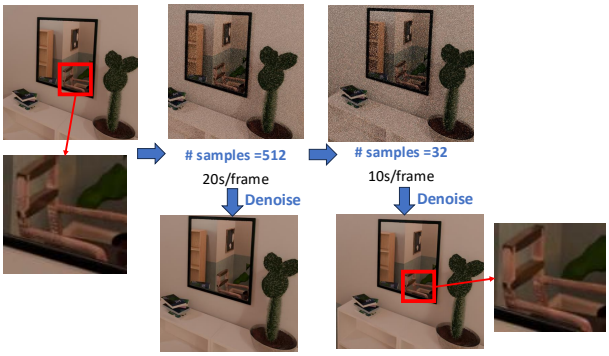

Figure 11. Impact of sample counts on image quality.

To further verify that VLM's spatial reasoning performance drops due to the scene complexity rather than object description ambiguity, we tested Gemini-3-Pro on scenes containing multiple tables with distinct items (e.g., dinnerware). As shown in Table 1 below, when scene complexity increases, the model maintained high accuracy on non-spatial identification tasks, but suffered large accuracy drop in the corresponding spatial measurement tasks.

Scene Complexity	Low	Medium	High
spatial task (measurements)	70.2	62.8	54.3
non-spatial task (descriptive)	94.1	93.5	93.6

Table 1. The variance of VLM's spatial reasoning task accuracy when different levels of scene complexity

References

[1] A. Raistrick, L. Mei, K. Kayan, D. Yan, Y. Zuo, B. Han, H. Wen, M. Parakh, S. Alexandropoulos, L. Lipson, et al. Infinigen indoors: Photorealistic indoor scenes using procedural generation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 21783–21794, 2024. 1, 2

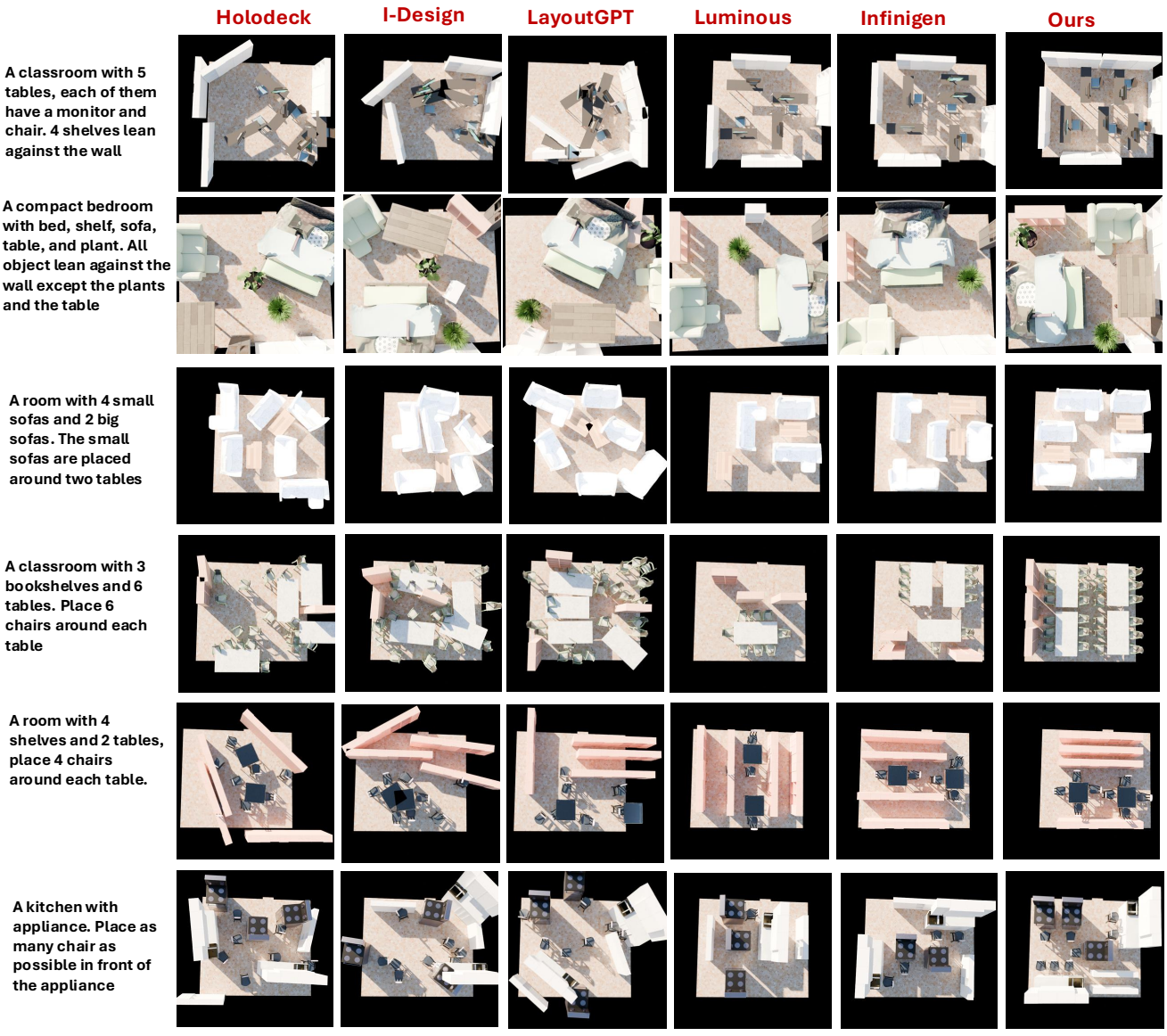

Figure 12. Comparing InfiniBench with baselines