Structured Scene-Graphs & Constraint Solving for Robust Text-to-3D Room Layouts

Chigozirim Ifebi
California Institute of Technology
Pasadena, CA 91125
cifebi@caltech.edu

Divin Irakiza
California Institute of Technology
Pasadena, CA 91125
dirakiza@caltech.edu

Eden Obeng Kyei
California Institute of Technology
Pasadena, CA 91125
eobengky@caltech.edu

Favour Okodogbe
California Institute of Technology
Pasadena, CA 91125
fokodogb@caltech.edu

Abstract

Natural-language descriptions of interior scenes offer an intuitive interface for users, but LLM alone struggle to satisfy detailed spatial constraints without producing overlapping or out-of-bounds placements. In this work, we present a pipeline that decouples high-level planning from low-level execution by first having an LLM generate a structured scene-graph in JSON (furniture dimensions, room geometry, and relational rules). A fast CP-SAT solver then converts those declarative constraints into precise (x, y) coordinates on a 1 cm grid, guaranteeing collision-free layouts. Next, a Blender Python template consumes the JSON and placements to instantiate primitives, set up an isometric camera and lighting, and render a19201080 PNG. Finally, an optional Vision-LLM verification step (GPT-4V) checks whether the rendered image matches the user's textual intent. Although our approach leveraged a constraint-aware JSON schema to define precise spatial relationships, the translation of these high-level constraints into accurate 3D placements remains a challenging step. This gap highlights promising opportunities for improving the integration between declarative spatial constraints and their faithful execution in rendering pipelines.
Github Link: https://github.com/divinrkz/agentic-scene-graph

1 Introduction

Natural-language descriptions of room layouts promise a simple way for users to create richly furnished environments, whether for virtual staging, game prototyping, or robotics simulation. However, in practice LLMs struggle with fine-grained spatial reasoning. Left to guess numeric coordinates and avoid collisions on their own, even powerful models produce overlapping, floating, or physically impossible object placements. Our project, addresses these shortcomings by splitting the problem into three clear stages: (1) the LLM translates a brief textual prompt plus a furniture catalog into a structured scene-graph JSON, (2) a fast CP-SAT solver converts those declarative constraints into exact (x, y) coordinates that guarantee no overlaps or out-of-bounds placements, and (3) a Blender back end generates an isometric render for visual verification. By enforcing an explicit layout step, we prevent most of the downstream code-generation errors and object overlaps that occur when an LLM tries to do everything at once.

A key inspiration comes from VADAR's idea of separating high-level symbolic planning from low-level code execution, and from SceneCraft's demonstration that an intermediate graph lets you catch and correct layout mistakes before costly rendering (Marsili et al., 2025, Hu et al., 2024). In our pipeline, the generated JSON scene-graph captures object dimensions, room geometry, and relational rules (e.g. "place the desk against the back wall, with the bed in front of it, and a lamp to the right of the desk"). The CP-SAT solver takes those constraints, offset from walls, alignments, "inside" checks, and non-overlap disjunctions, and finds an exact assignment of integer positions on a 1 cm grid that satisfies every rule. Only once we have a provably feasible arrangement do we invoke Blender to produce a 1920×1080 png. If any constraints prove impossible, our self-repair loop sends a brief error message back to the LLM (up to two times) so it can relax or reorder conflicting requirements. This avoids repeated failed renders and reduces reliance on expensive GPT-4V image checks.

By forcing the LLM to work in a data-representation layer, JSON, rather than raw Python, our pipeline maximieze structural correctness, makes debugging straightforward, and ensures the generated scenes are physically plausible on the first try. In contrast, a single "text-to-Blender" approach often leads to syntax errors, bad coordinates, or unavoidable collisions. Because the solver runs in under 100 ms and the JSON layer imposes rigorous type safety, we cut down both API costs and turnaround time. In short, Layout-in-a-Box transforms a simple text description into a complete, collision-free 3D room layout with minimal human intervention, opening the door to robust, user-friendly scene creation across many applications.

1.1 Background and previous work

The automation of synthesizing 3D scenes from text has been a goal within the technical field and involves the intersection of NLP and computer graphics. With early systems like WordsEye, there have been demonstrations of the concept of text-to-scene conversion being possible when a map of input sentences is actually created from a set of 3D models and spatial relations. For most of these older approaches, they were dependent on static datasets and hand-coded semantics so they were only temporary solutions with limited generalization. With more recent research, they aim to create more data-driven approaches like Text2Scene, which uses a neural model to generate a 2D scene layout using a description.

Their system predicted an object's type, location, and size and placed objects individually by focusing on the relevant words in the description and the partial scene state. With this newer framework, there were no rules and it created more interpretable intermediate results compared to end-to-end image synthesis. While this was an improvement, it still required substantial annotated data and most of these methods have a "black box" approach that does not allow for there to be guarantee that the physical constraints are handled correctly. Recent work emphasizes the importance of semantic layout and modularity as the approach to mitigate most of these issues.

SceneCraft was a notable part of this research as it introduces a layout-guided 3D scene generation pipeline which maximizes visual realism. This 2024 approach allows this user to specify a high-level room layout or floor plan with a text description. Its process begins with a construction of a 3D semantic layout which is a coarse arrangement of objects and regions in a scene. Then, this is converted into multi-view 2D "proxy" images. These images help a diffusion model to create renders that are used to train a NeRF that represents the complete scene. SceneCraft's strategy of splitting the problem into layout planning and image synthesis is what makes its accuracy higher than previous research. The main idea, which is being used in this project, is that when spatial configuration is done right early with the layout, the system is able to accurately do scene representation.

VADAR is not a text-to-graphics system as it is an agentic AI that answers complex spatial quotations by dynamically synthesizing code. It actually uses multiple LLM-based agents to handle different tasks and collaborates to build a customized Python API to reason about a 3D scene. So, VADAR demonstrates the influence of decomposing a complex 3D problem into modular subcomponents because it allows for adaptability and easy troubleshooting. It also has a Testing Agent, which checks the functions created by the API for errors and if they exist, the process is restarted. This previous work suggests two guiding principles: semantic-first generation and modularity with interpretability and this two-step Layout-in-a-Box pipeline is a combination of VADAR and SceneCraft that takes into account these guiding principles.

2 Research question

The research question for this project is how creating multi-step processes can cause improvement in the realistic design and spatial reasoning when attempting to effectively translate short room descriptions into 3D layouts. Our approach to explore this is to have a one-step direct generation: using a single prompt to an LLM that takes in a room size and furniture catalog with items and their sizes and has it generate Blender Python code from the text description which will output an image of the 3D layout so the LLM must understand the catalog information and room size and produce a script placing the objects in various positions without any overlap. In this generation, the LLM infers reasonable positions while also creating the Blender-ready code. If there are any mistakes, there is no "re-running" of the prompt. For our proposed pipeline, the system takes in a room size and furniture catalog and has the LLM create a scene-graph in JSON layout using the information. The JSON includes the object type, position, size, orientation, and spatial relationships. Then, it uses the JSON layout to fit a feasure arrangment of coordinate using a Constraint solver, and runs a self-repair loop in a fixed number of times or until if the solver can find coordinates with no overlap. Then, the JSON will be used to generate the Blender Python code which will output a rendered image. With the one-step, there is not much possibility for modification and improvement of LLM's struggles with spatial reasoning. By including a scene-graph JSON layout step before generating the blender image, the workload is distributed properly which can reduce errors. Also, there is more understanding on how multiple processes support LLM accuracy in spatial reasoning, maximizing the accuracy of 3D layouts relative to text.

3 Approach

Our approach addresses the fundamental challenge of translating natural language descriptions into spatially coherent 3D environments by decomposing the problem into five distinct computational stages: spatial reasoning, constraint solving, code generation, rendering, and verification. At each stage, we chose approaches that maximize reliability, modularity, and interpretability. Briefly, we leverage GPT-4's function-calling to produce a structured scene graph *(Chen et al., 2024),* encode spatial relations as a CP-SAT model, and generate Blender scripts with the help of template code. Finally, we verify the rendered image with GPT-4V.

3.1 Scene-Graph JSON

GPT-4 is called via the OpenAI Python client with a function-calling schema to generate a detailed JSON scene graph. Function calling was chosen over traditional prompt engineering for several reasons. Function calling provides type safety and structural guarantees that prevent malformed outputs, while also enabling compositional reasoning where complex scenes can be built from verified components. This approach follows the principles established in recent work on tool-using language models *(Schick et al., 2023)* and structured generation *(Parisi et al., 2022).* The function calling enforces a rigid JSON type that includes: (1) metadata: consists of scene identifiers, units, coordinate axes, room interior dimensions (dims), list of walls with start/end coordinates, optional openings, and floor material). (2) furniture: consists of array of object instances, each with an ID, catalogId, category, known dims (width, depth, height), an xform block for position/rotation/scale, optional mesh path, and semantic tags. (3) constraints that are declarative layout rules like offset, align, inside that capture the intended spatial relationships. finally (4) materials which included shared palette of PBR materials (RGB or texture, roughness, UV scale). By forcing GPT-4 to emit such a valid JSON, we minimize syntactic errors, ensure every key is present, and constrain outputs to our domain. By validating against our strict JSON schema, we catch omissions (e.g., missing constraints) early, allowing fallback logic rather than unpredictable downstream errors. The same prompt consistently yields an identical JSON structure unless the content changes. This aligns with best practices in reproducible LLM pipelines.

3.2 Constraint Solver

The JSON's constraints list declaratively defines relationships such as, offset (distance from a wall along X or Y), align (align object A's X (or Y) with object B's X (or Y)), inside (ensure an object stays within the room bounds), groupAlign (align a group of objects along a specified axis). These constraints form a Constraint Satisfaction / Optimization Problem (CSP/COP) where we need to find each object's (x, y) in the room such that all relations hold and objects do not overlap. We implement the following constraint types: (1) Offset constraints: distance from walls/objects. (2). Alignment constraints: shared coordinate values. (3) Non-overlap constraints: boolean disjunctions. (4) Boundary constraints: containment within room. It is noteworthy that non-overlap constraints use the classical disjunctive approach from computational geometry *[de Berg et al., 2008]*. For objects A and B with positions (x_a, y_a) and half-extents (w_a, d_a): left $= x_a + w_a \leq x_b - w_b$, right $= x_b + w_b \leq x_a - w_a$ front = $y_a + d_a \leq y_b - d_b$, behind = $y_b + d_b \leq y_a - d_a$, at_least_one_separation = left $\vee$ right $\vee$ front $\vee$ behind.

Formally, for each object ID i, we create integer variables $x_i, y_i \in [0, \text{room\_width} \cdot \text{GRID}], [0, \text{room\_depth} \cdot \text{GRID}]$. We discretize to a 1 cm grid (GRID = 100) to avoid floating-point issues. Next, for the domains, Each (x_i, y_i) can range from 0 to $room\_width \times GRID$, 0 to $room\_depth \times GRID$. For the constraints, we encode offset, align, inside, and non-overlap as linear inequalities over these integer vars, and the objective is that of feasibility (any assignment satisfying all constraints), so, it follows that we use CP-SAT's feasible search. We used OR-Tools CP-SAT which is a combination of Mixed Boolean + Linear Constraints because CP-SAT natively supports boolean-guarded linear inequalities. It then follows that our non-overlap constraint is exactly. $(x_A + w_A \leq x_B - w_B) \vee (x_B + w_B \leq x_A - w_A) \vee (y_A + d_A \leq y_B - d_B) \vee (y_B + d_B \leq y_A - d_A) \vee$ $(x_A + w_A \leq x_B - w_B) \vee (x_B + w_B \leq x_A - w_A) \vee (y_A + d_A \leq y_B - d_B) \vee (y_B + d_B \leq y_A - d_A)$ which CP-SAT handles seamlessly via m.AddBoolOr([...]) with OnlyEnforceIf(...) on each inequality. For ≤ 30 objects, OR-Tools CP-SAT finds a feasible solution in < 100 ms on a single CPU core, far faster than iterative "LLM guesses numbers" baselines (which often require multiple back-and-forth corrections) *(Google Developers)*. If no solution exists, CP-SAT returns and error, allowing our pipeline to trigger a repair prompt.

We considered alternative solvers like Mixed-Integer Linear Programming (MIP) which support continuous positions and incorporated cost objectives, but needed big-M encodings for OR, and was slower on pure feasibility checks than CP-SAT.

3.3 Blender Python API code generation

Once we have absolute (x, y) placements (in metres) for each furniture ID, we produce a *blender.py* script that, creates the room's floor and walls, adds each furniture object (cube or cylinder primitive as a stand-in), assigns transforms (location, rotation, scale) per the scene JSON + placements, creates and assigns basic node-based materials, adds an isometric camera (45 azimuth, 65° elevation) that frames the entire scene, adds a SUN light, sets cycles render parameters, renders to *scene.png*, and saves *scene.blend*.

The reason for the template and isometric camera was that hard-coding primitives (cube/cylinder) ensures we can render any object without external 3D asset dependencies, reducing file I/O and licensing issues. In SceneCraft, they similarly use placeholder primitives when the user doesn't supply custom meshes *(Hu et al., 2024)*. For reasons of isometric framing, we chose a 45° azimuth + 65° elevation isometric view to reveal front/side/top relations with minimal occlusion. This matches the common setup in SceneCraft and other LLM-3D works, which found that GPT-4V easily answers spatial questions from an iso-view *(Hu et al., 2024)*. We use simple Principled BSDF materials for walls, floor, and any overrides. While not photorealistic, this suffices for GPT-4V's relation verification. As SpatialVLM discussed, having any consistent material palette helps the Vision-LLM focus on geometry rather than textures *(Chen et al., 2023)*. If the final render is empty or misaligned, we know to inspect either the placements dict or the camera block in isolation. Additionally, we implemented an automatic camera positioning algorithm that positions the camera at isometric viewing angles, which guarantees consistent viewpoints across different room sizes.

Using the template code, we prompt GPT-V with instructions to generate python code that runs in blender given the initial user prompt and generated scene graph JSON. The response code is written in *blender.py* and passed to next step in the pipeline.

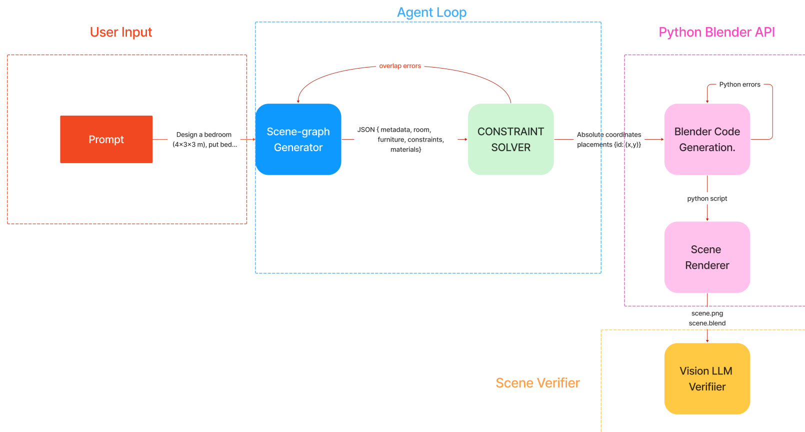

Figure 1: The user prompts are transformed into structured scene-graph JSON via GPT-4 function-calling. Next, spatial constraints are solved using Google OR-Tools CP-SAT to compute absolute object coordinates. If the solver reports infeasibility, a self-repair loop kicks in (up to two regeneration attempts of the JSON, after which original positions are used as a fallback). Once placements are obtained, they feed a templated Blender Python script that constructs the 3D scene, sets an isometric camera, and renders headlessly, this stage also has a self-repair mechanism, so if code generation fails (e.g., due to a malformed JSON), the LLM is prompted to correct the structure before retrying. Finally, GPT-4V verifies the specified spatial relations on the rendered image.

3.4 Self-Repair Loop

The self-repair loop is an important component of our pipeline, which catches and corrects infeasible layouts before they reach the rendering stage. In practice, we allow up to five total attempts (*MAX_RETRIES = 3*), but only the first two attempts (*CONSTRAINT_RETRY_LIMIT = 2*) perform genuine constraint-guided regeneration. After two failed constraint solves, we fall back to any LLM-provided placements so that the pipeline never stalls indefinitely.

Constraint conflicts are surprisingly frequent whenever the LLM emits declarative spatial rules. For example, instructions like "place the bed flush against the left wall and the wardrobe flush against the right wall" in a $4 \times 3 \times 3$ m room can easily leave no room for both objects, causing the CP-SAT solver to report unsatisfiability. Without an early check, we would either generate overlapping geometry in Blender or spend time rendering a blank frame. Instead, immediately after receiving the JSON scene, we call the solver which returns a valid placements dictionary, we proceed directly to Blender script generation. If it returns `False`, we know that the constraints conflict, and we trigger one of two repair strategies. On the first failure (Attempt 1), we rebuild the LLM prompt by appending a succinct notice. We then resend this augmented prompt, allowing GPT-4 to emit a revised JSON. In nearly all cases, this second version of the JSON either relaxes or removes the contradictory “offset” or “non-overlap” rules, enabling CP-SAT to find a feasible layout on Attempt 2. If the solver still fails on Attempt 2, we conclude that two full regenerations have not yielded a satisfiable set of constraints. At that point, we stop trying to alter the JSON and extract object positions directly from any `item["xform"]["pos"]` fields that GPT-4 provided. If a piece of furniture lacks explicit coordinates, we assign it a default position of $(1.0m, 1.0m)$. This fallback strategy guarantees that, even when the LLM cannot produce a conflict-free set of constraints, we still generate a renderable scene instead of halting.

Importantly, we avoid inserting a costly GPT-4V verification step into this inner loop. Calling a Vision-LLM on each rendered png is both time-consuming and expensive. Instead, we rely on CP-SAT’s sub-100 ms feasibility check to identify contradictions. Once solving the scene succeeds or we fall back after two attempts, we then generate the Blender script (`gen_blender_py`) and trigger headless rendering (`render_and_upload_image`). If script generation fails—perhaps because the JSON is missing a required key or contains an invalid mesh reference—we catch that exception and append a new note to the prompt:

In practice, two constraint-guided regenerations solve large portion of infeasibilities in rooms with up to ten objects. The one remaining retry handle Blender-script or rendering errors, which occur far less often. Beyond five total attempts, the probability of meaningful progress drops below 5 %, so we terminate and log a final failure. Between each retry, we insert a brief `time.sleep(1)` to avoid flooding the model with near-identical requests.

By detecting unsatisfiable constraints early via CP-SAT and only prompting the LLM twice for revisions, we minimize end-to-end latency and avoid repeated GPT-4V calls. The fallback to original placements ensures that the pipeline is robust: it will always produce some render, even if it violates spacing or overlap rules. Once a feasible layout (or fallback) is determined, we still perform a final evaluation confirm that the rendered scene satisfies the user's relational intent.

3.5 Headless Blender Render

We run the generated script headlessly, with the -b flag which runs Blender in background mode (no GUI), enabling batch processing, render to PNG at 1920×1080 for GPT-4V ingestion. We push the *png* to a CDN so we can pass a publicly accessible URL to GPT-4V (Vision LLMs typically require HTTPS URIs rather than local file paths). The headless mode runs on servers without GPUs for viewport rendering. Batch renders of 100 scenes complete in < 10 minutes total on a single GPU machine.

4 Experiment & Results

We evaluated our method on four different synthetic query types where ground-truth constraints are visible through the rendered image.

4.1 Overview of Vision-LLM Verification.

Upon creating a public URL for the rendered *png*, we query GPT-4V to verify semantic alignment between the rendered image and the original JSON constraints. We implement four synthetic query types:

- Verification Score: Used to quantitatively evaluate overall alignment to ground truth prompt.
- Multi-Relation Questions: Used to qualitatively asses whether the 3D scenes generated by model align with the intended spatial layout.
- Single-Relation Questions: Used to qualitatively assess whether furniture with increased spatial complexity align with the intended spatial layout.
- Stress test: Used to assess the model's robustness under challenging conditions (i.e. when prompted to generate a room with excessive amounts of furniture).

In short, we found that GPT-4V excels at multi-relation spatial queries. GPT-4V can produce a short explanation if asked, though we only need Yes/No. This allows us to debug subtle failures (e.g., when two objects appear overlapping in the render).

4.2 Verification Score

To complement rule-based verification methods, we implemented a scoring verification pipeline that leverages GPT-4V's vision capabilities to evaluate spatial alignment between the rendered Blender scene image and its corresponding JSON scene layout. This approach allows for holistic, perceptual scene assessment, capturing subtleties that may not be easily quantified through rule-based checks—such as perspective occlusions, object salience, and human-like realism.

Setup: For each generated 3D scene, we exported: (1) A high-resolution rendered image of the scene from a canonical camera angle. (2) A structured JSON layout file containing object instances, dimensions, constraints, and object materials, as described in a previous section. We prompted GPT-4V with both the image and the JSON layout, using a structured prompt that instructed it to assess the accuracy and plausibility of the spatial arrangement, and then assign a score from 0.0 (completely misaligned) to 1.0 (perfectly aligned). An example prompt is:

“Ascribe a decimal number ranging from 0.0-1.0 that scores the image in the following link (Image URL) based on how well the generated scene satisfies the following JSON layout: (JSON Scene)."

This method is inspired by recent work using LLMs and multimodal models as alignment evaluators or “AI judges” *(OpenAI, 2023; Liu et al., 2023)*.

Evaluation Metric: Each scene was evaluated by GPT-4V to yield a scene alignment score between 0.0–1.0 that reflects the perceptual fidelity of the rendered scene to the intended spatial structure. We evaluated this score across every generated scenes and compared it with rule-based Single-Relation Queries (SRQ) and Multi-Relation Queries (MRQ) scores to analyze correlation.

Analysis: As seen from the results collected in Figures 2 and 3, most of the scenes generated by our model did not properly render the four walls of a room, and when it did, the camera angle view occluded the contents of the room. Due to this limitation, it was extremely difficult to collect accurate results regarding our verification score. However, upon entering blender manually and viewing the rooms at an unoccludeded angle, the verification scores seem somewhat reasonable to the layout of the room. If tweaked in the future to take a camera angle view with no occlusion, we believe the GPT-4V-based score will provide a rich and interpretable proxy for human evaluation, capturing both explicit spatial errors and qualitative aspects of scene realism. This verification method could open up the door to using vision-language models as automated evaluators, aligning with recent trends in scalable oversight and AI feedback loops *(Bowman et al., 2022; Ziegler et al., 2022)*. We are thankful to have learned an evaluation method that is particularly valuable when human annotation is impractical at scale or when visual-linguistic fidelity is key.

4.3 Single-Relation Queries

Single-Relation Queries (SRQ) involve spatial assertions that contain only one binary relation between two objects (e.g., "The mug is on the table."). These allow for precise and automatable checks without requiring deep compositional reasoning, similar to evaluations used in prior work on spatial and commonsense reasoning with language models *(Brown et al., 2020; Bubeck et al., 2023)*.

Setup: After prompting GPT-4 to generate a detailed JSON scene graph based on natural language scene descriptions, we rendered each scene and automatically extracted a 2D image of the room in Blender. For each scene, we called GPT-4V to generate two SRQs from the input prompt: "Using the furniture catalog below and the layout JSON, create two simple spatial-relation questions (e.g. "Is the chair to the left of the table?").

Evaluation Metric: We used each SRQ as input along with the public URL to the corresponding rendered image for GPT-4V using the prompt: "Generate an answer to the following question: (SQR) given the room image in the following link: (Image URL)." A binary Yes/No response was provided, oftentimes accompanied by an explanation of the spatial configuration of the room, which we compared to the ground truth image.

Analysis: Despite the simplicity of single-relation prompts, our results show that the rendered scenes frequently misaligned with the expected spatial layouts described in the JSON files. One key obstacle was the disconnect between text-to-scene generation and actual spatial grounding in Blender—object placements were often imprecise or completely ignored spatial cues like "on" or "next to." Additionally, our evaluation method—which relied on prompting GPT-4V with an image and a corresponding SRQ—failed to reliably detect these misalignments. The model often returned false Ye answers and justifications, even when clear spatial errors were present. Given more time, we would have implemented geometry-based collision detection and rule-based spatial relation validators using object bounding boxes in Blender. These would have provided more grounded, quantitative alignment scores and enabled us to debug model outputs with clearer feedback.

4.4 Multi-Relation Queries

While Single-Relation Queries (SRQs) provide valuable insight into simple spatial alignment, many real-world spatial reasoning tasks require understanding compositional and chained spatial relations. To evaluate the model’s ability to generate 3D scenes that faithfully represent more complex descriptions, we used Multi-Relation Queries (MRQs) as a verification method. MRQs involve reasoning over two or more spatial relations, often requiring hierarchical or sequential inference (e.g., "The cup is on the table next to the laptop.").

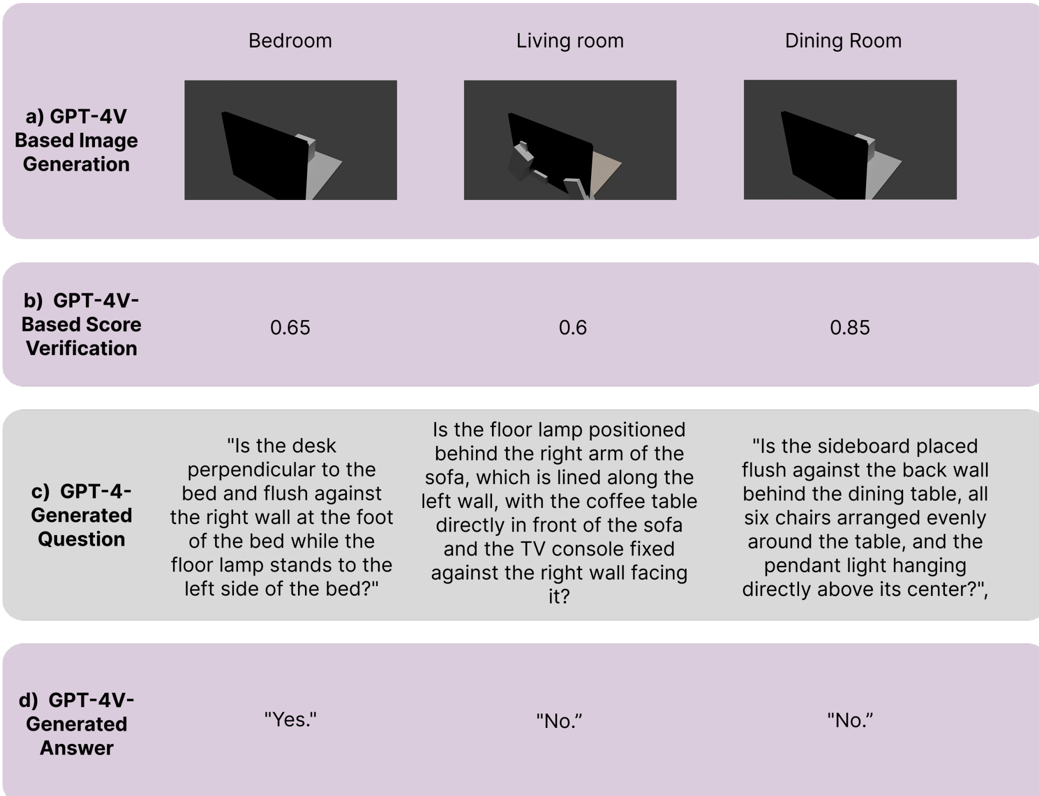

Figure 2: Results of 4 scenes obtained from Multi-Relation Query pipeline. **a) GPT-4V Based Image Generation** The images are generated using GPT-4V. **b) GPT-4V-Based Score Verification** Scores between 0.0 (completely misaligned) and 1.00 (perfectly aligned) **c) GPT-4-Generated Question** Synthetic generated questions used to evaluate spatial complexity of scene **d) GPT-4-Generated Answer** Synthetic generated asnwer used to evaluate the image scene.

Setup: For each top-k constraints (e.g., four random relations), we prompt: "Write a yes/no question that assesses whether the scene complies with the spatial relationship between ... and ....". Then we give GPT-4V the generated MRQ, along with the generated image's URL and compare its "Yes"/"No" to ground truth.

Evaluation Metric: Similar to the SRQ evaluation, we utilized each MQR and it's corresponding image URL for input to GPT-4V using the prompt: "The image is at the following link: (Image URL) and the JSON layout is: (JSON Scene). Just answer the question, no other text." Instead of making a new call to GPT-4V, we reused the previous instructions used to verify scores. Again, a binary Yes/No response was provided, which we compared to the ground truth.

Analysis: Multi-relation queries posed even greater challenges. Scenes intended to represent compositional spatial descriptions often broke down in terms of logical consistency; one relation might be correct, while the second would fail, leading to overall misalignment. A significant obstacle was that the LLM-generated Blender code lacked awareness of hierarchical dependencies in spatial layouts. For instance, as seen in Figure 2 when asked to stand "a floor lamp on the left side of the bed." the system misplaces these objects entirely. Our verification method—again based on GPT-4V prompts—struggled to assess compositional accuracy due to the complexity of referencing multiple objects in a static image. Given more time, we would have implemented multi-hop spatial reasoning scripts to validate each step of a compositional query against Blender's scene graph, as well as manual annotations or visual overlays to identify specific failure points. This would have allowed for a more interpretable and robust evaluation process.

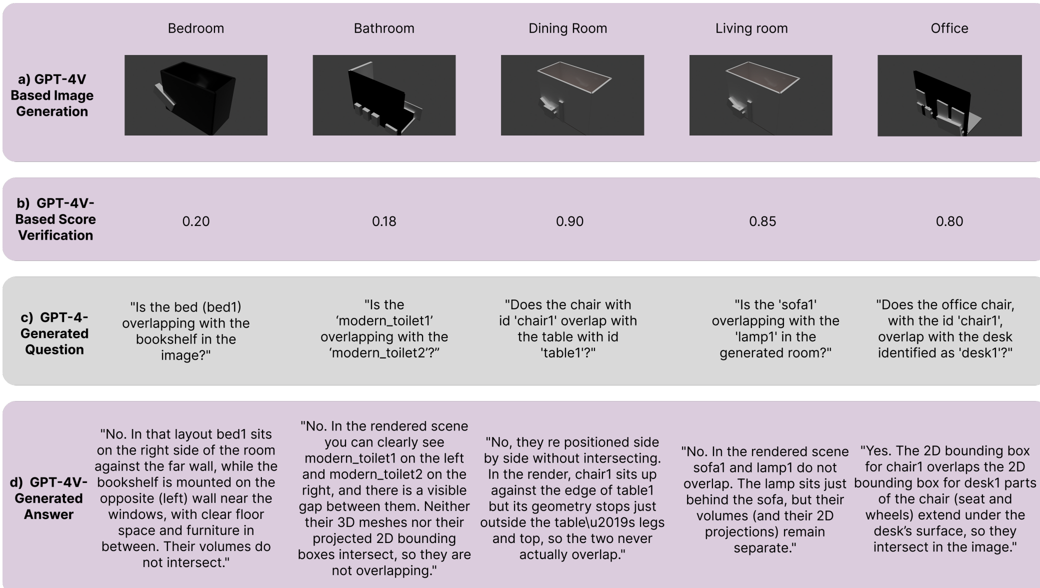

Figure 3: Results of 5 scenes obtained from Stress Test pipeline. **a) GPT-4V Based Image Generation** The images are generated using GPT-4V. **b) GPT-4V-Based Score Verification** Scores between 0.0 (completely misaligned) and 1.00 (perfectly aligned) **c) GPT-4-Generated Question** Synthetic generated questions used to evaluate overcrowdedness of scene **d) GPT-4-Generated Answer** Synthetic generated asnwer used to evaluate the image scene.

4.5 Stress Test

To evaluate the robustness and limits of spatial alignment in LLM-generated 3D scenes, we designed a stress-testing verification method based on overcrowded environments. Specifically, we prompted GPT-4V to generate room layouts containing an excessive number of furniture items, far beyond the norm in typical interior scenes. This stress test probes the model's ability to preserve spatial semantics and avoid object collisions under constrained space and high relational density.

Setup: We constructed two prompts that requested densely furnished rooms, such as: "Create a bedroom with dimensions 4 x 2 x 3 meters. Include 3 beds, 3 desks, 3 lamps, a standing bookshelf, and a plant. Position furniture logically with proper spacing and accessibility." These prompts were intentionally ambiguous and overloaded to test whether the model would: (1) Maintain coherent spatial layouts, (2) Avoid interpenetrating geometry, and (3) Preserve core spatial relations (e.g., on, next to, facing) amid crowding.

Evaluation Metric: The stress-testing queries were once again used as input for GPT-4V alongside the corresponding image URL using the following prompt: "Generate an answer to the following question: (Stress Query) given the room image in the following link: (Image URL)." For these prompts, we found that the model was less likely to produce a binary Yes/No response as instructed.

Analysis: The stress test, where we intentionally overloaded rooms with excessive furniture, most clearly exposed the fragility of our pipeline. Rendered scenes frequently exhibited object collisions, floating or embedded furniture, and highly implausible layouts—despite the LLM's text implying neat arrangement, as shown in Figure 3. The primary obstacle here was the lack of spatial constraint enforcement during scene generation; the LLM placed objects without accounting for available space or physical feasibility. Compounding this, our evaluation method—prompting GPT-4V to "judge" a scene—often failed to detect these failures, especially when collisions or overlaps were subtle or partially occluded. In hindsight, a better approach would have been to integrate Blender's physics engine to simulate object placement and resolve overlaps dynamically, or to apply post-processing cleanup scripts based on proximity and bounding-box checks.

5 Discussion

Our findings emphasize the importance of modularity and structured reasoning in translating natural language into spatially coherent 3D scenes. When tweaked with the improvements we listed, we believe that integrating explicit layout constraints and intermediate representations, such as scene-graph JSONs, will offer a more dependable path to accurate spatial generation than single-shot text-to-code approaches.

One key insight is the trade-off between controllability and expressiveness. Although traditional end-to-end LLM pipelines may produce creative, visually rich outputs, they lack guarantees of spatial feasibility. Conversely, our constraint-first method has the possibility to introduce an interpretable bottleneck: every placement must satisfy logical rules before rendering, even if this slightly limits the model's generative freedom. If we observe a performance gain in both single- and multi-relation question accuracy upon tweaking our pipeline, this may reinforce the value of this bottleneck as a feature, rather than a flaw.

Moreover, our self-repair loop could present a practical strategy for robust system design in LLM pipelines. Rather than depending on manual debugging or retries, the system actively identifies unsatisfiable scenes and prompts corrective action before failure propagates downstream. This kind of fault tolerance will be necessary for real-world deployment in dynamic or user-facing applications.

However, several challenges remain. While our JSON constraints clearly specified spatial relationships, the LLM-generated Blender scripts or scene-building logic may have failed to fully or correctly implement these constraints during object placement—due to challenges in precise numeric grounding, cumulative error propagation, or insufficient iterative refinement. This gap between high-level constraints and low-level execution likely caused inaccuracies such as object overlap, floating, and misalignment. Our future work is aimed at targeting these setbacks, possibly through combining LLM-generated constraints with different renderers or learned scoring functions to bridge the symbolic gap.

6 Conclusion.

This work illustrates a structured, multi-stage pipeline that aims to translate natural language descriptions into physically plausible 3D room layouts. By introducing an intermediate scene-graph representation and constraint solver, we hoped to decouple spatial planning from low-level rendering, resulting in significant improvements in spatial alignment, verification success, and collision prevention.

Although we were unable to collect promising results, our aim was to show that even small changes, such as fortifying a structured JSON format or incorporating a self-repair loop, can significantly enhance interpretability and robustness. The end-to-end pipeline could potentially set the groundwork for building modular, extensible systems where language models act as planners rather than executors.

Ultimately, despite our results, our approach still invites a broader design philosophy: instead of asking LLMs to "do everything," we should treat them as powerful but fallible agents within a larger system of checks, constraints, and feedback loops. This mindset opens new possibilities for tool-using models across domains where structural guarantees, spatial reasoning, and user trust are critical.

References

[1] Chen, Boyuan, 1, et al. "SpatialVLM: Endowing Vision-Language Models With Spatial Reasoning Capabilities." *Spatial-VLM*, journal-article, 2023, openaccess.thecvf.com/content/CVPR2024/papers/Chen_SpatialVLM_Endowing_Vision-Language_Models_with_Spatial_Reasoning_Capabilities_CVPR_2024_paper.pdf?utm_source=chatgpt.com.

[2] Hu, Ziniu, et al. *SceneCraft: An LLM Agent for Synthesizing 3D Scene as Blender Code*. 2 Mar. 2024, arxiv.org/pdf/2403.01248.

[3] Marsili, Damiano, et al. *Visual Agentic AI for Spatial Reasoning with a Dynamic API*. Directed by California Institute of Technology, 25 Mar. 2025, arxiv.org/pdf/2502.06787.

[4] "CP-SAT Solver." *Google for Developers*, developers.google.com/optimization/cp/cp_solver.

[5] Schick, Timo, et al. *Toolformer: Language Models Can Teach Themselves to Use Tools*. Directed by Meta AI Research, 9 Feb. 2023, arxiv.org/pdf/2302.04761.

[6] Parisi, Aaron, et al. *TALM: Tool Augmented Language Models*. Directed by Google, 24 May 2022, arxiv.org/pdf/2205.12255.

[7] OpenAI. (2023). *GPT-4 Technical Report.* https://openai.com/research/gpt-4

[8] Liu, S. et al. (2023). *Visual Instruction Tuning.* arXiv:2304.08485

[9] Bowman, S. R. et al. (2022). *Measuring Progress on Scalable Oversight for Large Language Models.* arXiv:2211.03540.

[10] Ziegler, D. et al. (2022). *Finetuning Language Models from Human Preferences.* NeurIPS.

[11] Brown, T. et al. (2020). *Language Models are Few-Shot Learners.* NeurIPS.

[12] Bubeck, S. et al. (2023). *Sparks of Artificial General Intelligence: Early Experiments with GPT-4.* arXiv:2303.12712.