GEOSTATE: QUERYABLE 3D SCENE STATE FOR COMPOSITIONAL SPATIAL REASONING

Anonymous authors
Paper under double-blind review

ABSTRACT

Reliable 3D spatial reasoning from images requires not only accurate perception, but also representations that support explicit geometric computation. Existing vision-language models often entangle perception, geometry, and reasoning within a single language-based process. In contrast, visual-programming approaches first extract heterogeneous perception outputs such as masks, depth, and camera parameters and subsequently generate reasoning programs. We argue that this intermediate representation is a critical but underexplored design choice in visual programming. We introduce **GeoState**, a geometry-grounded visual-programming framework that consolidates heterogeneous monocular perception outputs into a reasoning-friendly, object-centric 3D representation. Each object is represented by its semantic identity, 3D position, orientation, and extent in a shared coordinate frame, enabling spatial relations to be computed directly through simple geometric operations. GeoState further uses a planner-guided program synthesis to decompose spatial queries into high-level geometric operations and executable code over these representations. Without task-specific spatial reasoning supervision or fine-tuning, GeoState substantially outperforms prior visual-programming approaches and achieves performance competitive with or exceeding spatially fine-tuned VLMs on stratified subsets of 3DSRBench and Omni3D-Bench. Controlled ablations further show that, with the underlying perception modules fixed, consolidating their outputs into object-centric geometry consistently improves reasoning over directly composing heterogeneous model-specific representations. These results highlight the importance of reasoning-friendly intermediate representations for reliable executable 3D spatial reasoning.

1 INTRODUCTION

3D spatial reasoning from visual observations requires recovering and manipulating geometric properties that are only implicitly available in images. Questions about object locations, orientations, distances, and multi-object spatial relations often require several intermediate geometric operations before an answer can be derived (Yu et al., 2025; Yang et al., 2025b). Despite rapid progress in large vision-language models (VLMs) (Bai et al., 2025; OpenAI, 2024; Team, 2025; Cheng et al., 2024; Ma et al., 2025a; Yang et al., 2026), reliable 3D spatial reasoning remains challenging, particularly for object orientation and compositional relations (Fu et al., 2025; Ma et al., 2025a; Tong et al., 2024).

Recent approaches improve spatial reasoning through finetuning with spatial VQA supervision and chain-of-thought reasoning (Cheng et al., 2024; Ma et al., 2025b; Yang et al., 2026). However, these models must still recover object geometry from visual observations and manipulate that geometry within the same language-based reasoning process. Perception, geometric computation, and semantic reasoning are therefore tightly entangled. As illustrated in Fig. 1, even a spatially fine-tuned VLM may estimate an inaccurate 3D direction and subsequently make an error during an intermediate geometric computation. Such failures highlight the brittleness of requiring a single model to both infer geometric quantities and reason over them implicitly.

Visual programming provides a natural way to separate these components (Surís et al., 2023; Gupta & Kembhavi, 2023; Marsili et al., 2025). Instead of performing all computation inside a VLM, a model generates executable programs that invoke specialist perception models and explicitly compute the answer. This separates perception from symbolic or numerical reasoning and has proven effective

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

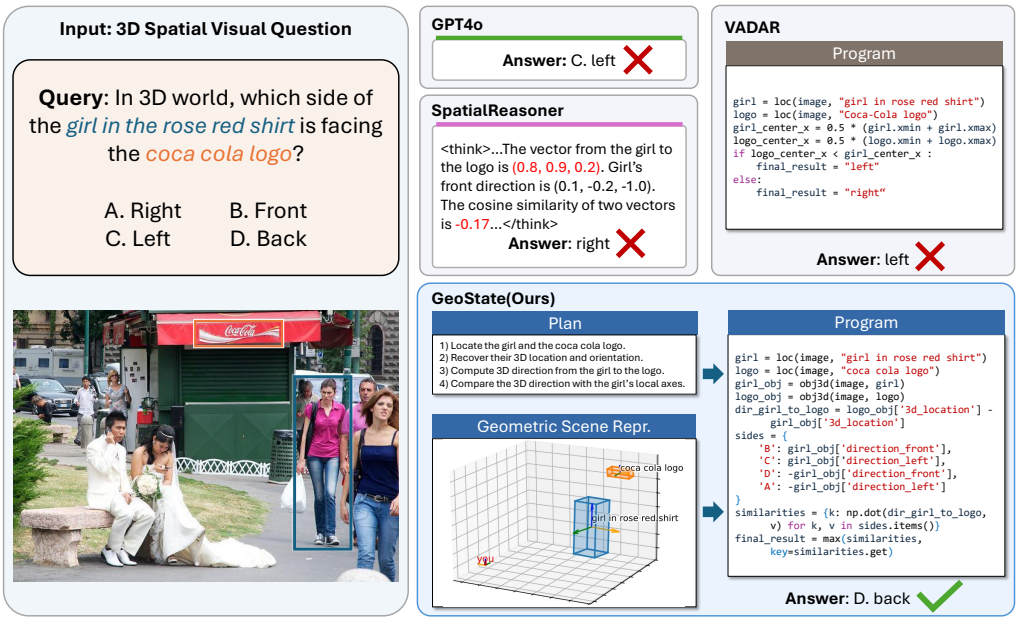


Figure 1: 3D spatial reasoning remains challenging for existing vision-language models (VLMs) and visual-programming methods: approaches relying on 2D cues often hallucinate, while limited 3D perception and poor geometric computation can lead to incorrect geometric judgments. We propose **GeoState**, a training-free agentic framework in which a planner and a program agent synthesize an executable program that constructs and queries an explicit object-centric geometric scene representation to answer spatial questions.

across a range of visual reasoning tasks. Extending this paradigm to 3D spatial reasoning, however, exposes a new challenge. Specialist perception models return heterogeneous quantities at different levels of abstraction, including segmentation masks, depth maps, camera parameters, object poses, and orientations. When these outputs are exposed through separate APIs, the generated program must associate them with the correct objects, transform them into compatible coordinate systems, and combine them into the geometric variables required by the question. Thus, although perception is externalized, *perception integration remains part of the generated reasoning program*.

We hypothesize that the representation connecting perception and programmatic reasoning is a critical design choice. Geometric reasoning should become simpler and more reliable when heterogeneous perceptual predictions are first consolidated into a representation aligned with the computations required by the task. Rather than repeatedly integrating model-specific outputs inside generated programs, we represent each relevant object through a compact set of geometric variables on which spatial relations can be computed directly. This leads to our central question: **Can consolidating heterogeneous perception outputs into an object-centric geometric representation simplify executable 3D spatial reasoning and improve its reliability?**

To investigate this question, we introduce **GeoState**, a geometry-centric visual programming framework. Its core abstraction is an object-centric scene state in which each task-related object is represented by its semantic identity, 3D position, orientation, and extent in a shared camera-centric frame, supplemented with gravity-aligned height for vertical-extent queries. GeoState exposes this representation through `obj3d()`, a composite operator that integrates heterogeneous pretrained monocular perception models and returns an oriented 3D bounding-box representation for a queried object. GeoState enables generated programs to reason directly over these homogeneous geometric entities. This representation supports a broad class of object-level spatial computations while keeping the intermediate geometry explicit and inspectable.

Given this representation, GeoState further separates high-level reasoning from program execution. A planner agent first produces a structured plan describing the geometric entities and operations required to answer a question, and a program agent then translates this plan into executable code over the object-centric representation. Execution feedback exposes intermediate program states

and supports revision when execution fails. Together, these components separate high-level spatial planning, geometric computation, and perception integration.

We evaluate GeoState on stratified subsets spanning all question types from 3DSRBench (Ma et al., 2025a) and Omni3D-Bench (Marsili et al., 2025), containing 546 and 500 questions respectively. Without spatial reasoning annotations or task-specific fine-tuning, GeoState substantially improves over prior visual-programming approaches and achieves performance competitive with or exceeding spatially fine-tuned VLMs. Importantly, controlled ablations holding the underlying perception modules fixed show that consolidating their heterogeneous outputs into object-centric geometry improves reasoning compared with directly composing the original model-specific representations. Structured planning and program repairing provide additional gains on compositional queries.

Our main contributions are:

1. **A reasoning-friendly geometric representation for visual programming.** We identify the choice of intermediate representation as a key factor in executable 3D spatial reasoning and introduce an object-centric representation that consolidates heterogeneous monocular perception outputs into position, orientation, and extent in a shared coordinate frame, directly supporting common spatial computations.
2. **Geometry-centric visual programming over object representations.** We introduce `obj3d()` to construct object-level 3D geometry from heterogeneous pretrained perception models, together with structured planning and executable program generation that reason directly over these geometric entities.
3. **Strong 3D spatial reasoning without task-specific fine-tuning.** GeoState substantially improves over prior visual-programming approaches and achieves performance competitive with or exceeding spatially fine-tuned VLMs on two 3D spatial reasoning benchmarks. Controlled ablations further isolate the benefit of the proposed representation from that of the underlying perception models.

2 Related works

VLMs for 3D Spatial Reasoning. Despite strong performance on many vision tasks, recent VLMs still exhibit limitations in 3D spatial understanding (Fu et al., 2025; Ma et al., 2025b; Cai et al., 2026; Gholami et al., 2026). Prior work addresses this gap by adding 3D or spatial supervision (Cheng et al., 2024; 2026; Yang et al., 2026; Zhan et al., 2025; Jia et al., 2024), introducing geometric intermediates (Ma et al., 2025b; Fan et al., 2025; Yang et al., 2025c; Lee et al., 2025; Zhu et al., 2025), or designing specialized training objectives (Huang et al., 2025; Chen et al., 2026; Lian et al., 2025; Lu et al., 2026). Representative methods such as SpatialVLM (Chen et al., 2024) and SpatialRGPT (Cheng et al., 2024) train on spatially grounded VQA data, while SpatialReasoner (Ma et al., 2025b) and VST (Yang et al., 2026) improve spatial reasoning through explicit 3D intermediates, scaled spatial data, and multi-stage tuning.

In contrast to these model-centric approaches, which internalize 3D reasoning within a VLM, our method decouples planning from perception: an LLM agent synthesizes programs, while specialist vision modules provide 3D perception primitives. This tool-augmented design enables explicit geometric computation with interpretable intermediate states, and remains complementary to stronger VLMs or perception models that can be integrated as improved modules.

Explicit 3D Representations. Explicit 3D representations provide a metric and structured substrate for 3D perception and reasoning (Wang et al., 2023; Ma et al., 2022). In 3D object detection, objects are often parameterized as *3D oriented bounding boxes*, enabling localization and pose estimation in a shared coordinate frame (Lang et al., 2019; Wang et al., 2021). In reconstruction, dense primitives such as *3D Gaussian Splatting* offer efficient rendering scene structure (Kerbl et al., 2023). Recent VLM-based methods further incorporate explicit geometric intermediates, such as depth maps or box-like representations, to improve spatial reasoning (Cheng et al., 2024; Ma et al., 2025b).

In contrast to prior work that uses geometry primarily as an internal perceptual representation or as an intermediate within a monolithic VLM, we expose explicit 3D structure as a *queryable geometric scene state* for visual programming. Objects are represented by 3D OBBs, and the agent can invoke APIs to dynamically query geometric attributes during program execution. This enables explicit

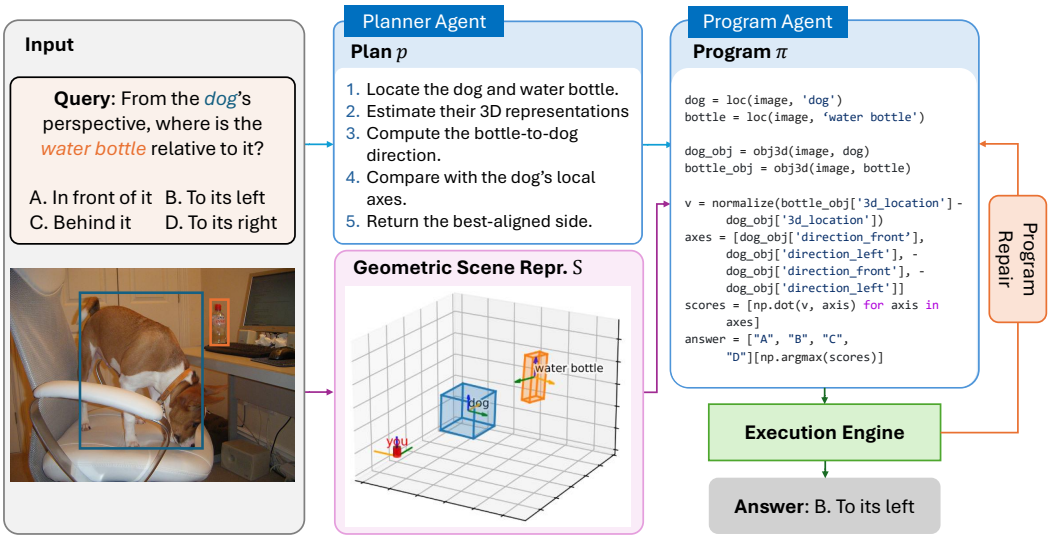


Figure 2: GeoState uses a two-agent pipeline to answer 3D spatial reasoning questions via executable computation over a compact geometric scene representation. The **Planner Agent** produces a high-level reasoning plan, and the **Program Agent** verifies the plan and compiles it into low-level executable code that constructs and queries an abstract 3D proxy of the scene. Green, yellow and blue arrows in geometric scene representation indicate the front, left and up direction for each object.

computation and verification over interpretable intermediate states, improving controllability and robustness in long-horizon 3D spatial reasoning.

Visual Programming. Visual programming methods solve visual tasks by synthesizing executable programs which invoke specialized vision modules. VisProg (Gupta & Kembhavi, 2023) demonstrates that LLMs can compose modular visual tools via in-context prompting to solve diverse visual queries. ViperGPT (Surís et al., 2023) further leverages code-oriented LLMs to generate executable Python programs with explicit tool invocation. More recently, VADAR (Marsili et al., 2025) augments visual programming with an agent that can propose and compose new APIs tailored to the input query.

Despite their flexibility, existing systems are mainly equipped with limited 3D perception ability and often turn to 2D cues or VQA modules to address 3D related questions. Their single-pass program generation can be brittle for complicated tasks requiring multi-step reasoning. In contrast, our framework explicitly incorporates geometric 3D scene representation for interaction and a planner-executor design, enabling more reliable 3D reasoning.

3 METHOD

GeoState organizes visual programming around an object-centric geometric interface. A fixed perception-integration procedure consolidates heterogeneous monocular predictions into queryable object geometry, while generated programs compose spatial operations over the resulting attributes. We first define the representation and the computations it supports, then describe its construction through `obj3d()` and the planning, programming and execution process that uses it.

3.1 OVERVIEW

Given an image I and natural-language question q, GeoState produces an answer $\hat{\mathbf{a}}$ by generating and executing a program over explicit object-centric 3D scene representations:

$$S = \{B_i^{3D}\}_{i=1}^{N}, \tag{1}$$

where each object representation B_i^{3D} encodes the semantic label, position, extent, and orientation of an object in the camera coordinate system. GeoState initializes S as empty and populates it on

demand during execution. Programs then query and manipulate S using a fixed library of perception and geometry APIs.

Model pipeline. As shown in Figure 2, GeoState consists of three modules:

1. *Planner Agent* $\mathcal{A}_{\text{plan}}$ generates a concise high-level plan $p = \mathcal{A}_{\text{plan}}(q, \mathcal{L}_p)$ that decomposes the question into reasoning steps over perception APIs $\mathcal{L}_p$.
2. *Program Agent* $\mathcal{A}_{\text{prog}}$ verifies plan–question consistency and compiles the plan into an executable Python program $\pi = \mathcal{A}_{\text{prog}}(q, p, \mathcal{L}_p)$ using the allowed APIs.
3. *Execution Engine* E runs π on image I with a persistent, lazily constructed scene S. Upon failure, E produces a structured crash report for program repair. The final answer is $\hat{\mathbf{a}} = E(\pi, I)$.

Perception library. Following VADAR (Marsili et al., 2025), we provide common perception APIs $\mathcal{L}_p$ for visual subtasks: `loc` localizes objects from text prompts, `get_2D_object_size` returns 2D object size, `depth` estimates center-point depth, and `vqa` queries image- or object-level visual attributes. We further introduce `obj3d`, a composite 3D operator that coordinates multiple perception models to construct 3D object proxies on demand.

By reasoning over object-centric 3D representations, GeoState supports compositional queries involving relative depth, 3D distance, and orientation that are ambiguous in 2D image space.

3.2 Object-Centric Geometric Representation for Visual Programming

GeoState answers spatial questions by generating executable programs over a structured 3D-aware visual programming interface. Rather than reasoning directly over pixels or 2D detections, programs operate on object-centric 3D primitives that make spatial structure explicit. This abstraction is exposed through a composite API, `obj3d`, which orchestrates perception models and geometric fitting while presenting a compact 3D representation to the program generator.

Object-centric 3D scene representation. Many spatial queries depend on object position, extent, and orientation rather than dense surface geometry. Accordingly, GeoState represents each object with an oriented 3D bounding-box abstraction

$$B_i^{3D} = (\ell_i, \mathbf{c}_i, \mathbf{R}_i, \mathbf{s}_i, h_i^g), \tag{2}$$

where $\mathbf{l}_i$ is the semantic label, $\mathbf{c}_i \in \mathbb{R}^3$ is the center in a camera-centric coordinate frame, $\mathbf{R}_i \in SO(3)$ is the object orientation, and $\mathbf{s}_i \in \mathbb{R}^3$ denotes the extents along the oriented axes. An estimated object extent along the gravity-aligned vertical direction h_i^g is also included to provide physical height information. This compact representation supports metric reasoning over relative depth, distance, containment, and facing direction, while avoiding the cost and complexity of dense 3D reconstruction.

Composite 3D operator. The `obj3d` operator links perception to geometric scene representation:

$$\texttt{obj3d}(I, b_i) \rightarrow B_i^{3D}, \tag{3}$$

mapping a 2D object box b_i in image I to an oriented 3D bounding-box proxy. Internally, `obj3d` obtains an object mask, predicts metric depth over the object region, and back-projects masked pixels into a camera-centric point cloud using estimated intrinsics $\hat{K}$. It then estimates the object's orientation relative to the camera and fits an oriented 3D box to the point cloud, yielding $(\mathbf{c}_i, \mathbf{R}_i, \mathbf{s}_i)$; the semantic label $\mathbf{l}_i$ is inherited from the object query that produced b_i.

By encapsulating calibration, depth estimation, orientation estimation, and box fitting within a single API, `obj3d` provides a concise and uniform geometric interface. This reduces the burden on the program generator and improves reliability by exposing 3D structure through a stable abstraction.

3.3 Planner-Guided Program Synthesis and Execution

Answering compositional spatial questions requires both high-level reasoning and low-level program synthesis. Directly mapping a question to code forces a single model to decompose the task, select perception operations, and produce executable implementations simultaneously, often leading to brittle programs. GeoState instead separates reasoning from implementation with a planner-executor architecture: a planner agent first decomposes the question into geometric reasoning steps, and a program agent compiles this plan into Python code over the scene representation in Section 3.2.

Table 1: Results on 3DSRBench and Omni3D-Bench. The best result is highlighted with **bold text** and the second best is shown with underline. GeoState achieves strong performance on location and multi-object questions, and outperforms all compared VLMs overall.

	3DSRBench					Omni3D-Bench				
Method	Height	Location	Orientation	Multi-Object	Overall	numeric(ct)	numeric(other)	y/n	multi-choice	Total
Proprietary VLM										
GPT4o	26.3	64.9	16.7	44.9	41.0	**28.1**	35.5	**66.7**	57.2	42.9
Gemini2.0-Flash	42.1	61.4	21.7	29.6	36.3	18.5	22.3	58.6	54.6	45.0
Open-source VLM										
Qwen2.5-VL-7B	31.6	66.7	30.0	38.8	42.7	20.0	8.9	44.0	43.0	36.3
VST-7B-RL	**68.4**	70.2	35.0	50.0	52.6	25.7	25.5	52.0	54.6	45.0
SpatialReasoner	52.6	66.7	**48.3**	53.1	55.1	14.2	25.2	53.3	24.4	30.7
Visual Programming										
VisProg	0.0	42.1	13.3	29.6	26.1	2.9	0.9	54.7	25.9	21.1
VADAR	36.8	43.9	15.0	37.8	33.3	21.7	35.5	56.0	57.6	40.4
GeoState	47.4	**71.9**	36.7	**67.3**	**59.0**	24.2	**35.8**	64.0	**59.3**	**50.2**

Planning. We utilize a Planner Agent to synthesize high-level reasoning plan before implementing a program to produce answer. The Planner Agent $\mathcal{A}_{\text{plan}}$ maps the question q to a multi-step plan

$$p = \mathcal{A}_{\text{plan}}(q, \mathcal{L}_p). \tag{4}$$

The planner receives a question with the perception library $\mathcal{L}_p$. This constrains the plan to executable operations while allowing it to focus on reasoning structure. The resulting plan p is a compact step-by-step procedure specifying the goals, entities, and operations needed to answer the question.

Program synthesis. An independent Program Agent is applied to perform plan verification and code generation. Given q and p, the Program Agent $\mathcal{A}_{\text{prog}}$ generates an executable program

$$\pi = \mathcal{A}_{\text{prog}}(q, p, \mathcal{L}_p), \tag{5}$$

using only the provided perception APIs and geometric operations. Before compilation, the agent checks whether the plan is consistent with the question and resolves missing or underspecified steps when possible. It then translates the plan into Python code that takes image I as input, constructs the required 3D object abstractions on demand, and performs geometric computations. The generated code is also instructed to handle API failures explicitly, returning diagnostic messages when required objects or attributes cannot be obtained rather than silently failing or hallucinating an answer.

Together, the planner and program agents transform a natural-language query into a program grounded in the geometric interface. This decomposition separates reasoning from implementation details while keeping the generated program constrained to verifiable perception and geometry operations.

Execution and program repair. The generated program π is executed in a controlled runtime environment with access to image I and the perception library $\mathcal{L}_p$, and its output is taken as the final answer $\hat{\mathbf{a}}$. Because perception outputs may occasionally be incomplete or inconsistent with program assumptions, execution can fail at runtime. When an exception occurs, GeoState constructs a structured crash report $R = \{E_{\text{msg}}, T_{\text{stack}}, V_{\text{state}}\}$, where E_{msg} denotes the exception message, T_{stack} the stack trace and failing line, and V_{state} the relevant local variables and intermediate results at the point of failure. The Program Agent then revises the failed program using this execution context: $\pi' = \mathcal{A}_{\text{prog}}(q, p, \pi, R)$. We repeat this process for at most K repair attempts (K=5 in our experiments); if all attempts fail, the system returns an execution error. This lightweight repair mechanism handles occasional program-generation failures without modifying the geometric representation or the high-level reasoning plan.

4 EXPERIMENTS

We evaluate GeoState on challenging benchmarks for image-based 3D spatial reasoning to assess the effectiveness of geometry-grounded visual programming without task-specific training. Our evaluation focuses on two aspects: (i) comparing GeoState with large VLMs and existing visual-programming approaches on established 3D spatial reasoning benchmarks, and (ii) analyzing the contribution of the key components of our framework through ablation studies.

4.1 Experimental Setting

Benchmarks. We primarily evaluate GeoState on **3DSRBench** (Ma et al., 2025a), a challenging benchmark for image-based 3D spatial reasoning. 3DSRBench contains *template-based* spatial reasoning questions constructed on carefully selected COCO images to minimize reliance on 2D cues. We evaluate on a proportionally stratified subset of 546 questions, covering all question types and preserving their relative frequencies in the full benchmark. This fixed subset supports matched comparisons across methods and ablations while controlling the computational and GPT-4o API costs of tool-based evaluation. During evaluation, we use the circular evaluation (Liu et al., 2024) strategy to mitigate option-order bias. A question is marked correct only if the model selects the correct option under all answer permutations.

To assess the generalization, we also evaluate on **Omni3D-Bench** (Marsili et al., 2025) with 500 challenging *non-template* spatial reasoning questions. Compared to 3DSRBench, Omni3D-Bench features more diverse, human-written queries demanding open-ended spatial reasoning. Together, they offer complementary evaluations spanning controlled templates and free-form questions.

Baselines. As a visual-programming framework, we primarily compare with representative visual-programming-based methods, including VisProg (Gupta & Kembhavi, 2023) and VADAR (Marsili et al., 2025). To contextualize performance against direct VLM inference, we also evaluate GPT-4o and Gemini 2.0 Flash without using external tools or program execution. For open-source models, we include Qwen2.5-VL-7B (Bai et al., 2025) and its spatially specialized derivatives, SpatialReasoner (Ma et al., 2025b) and VST-7B-RL (Yang et al., 2026), to contextualize the gains associated with task-specific spatial training within a shared backbone family.

Implementation details. Following prior visual-programming work (Marsili et al., 2025), we use GPT-4o as the agent backbone to facilitate comparison. Our *Planner Agent* and *Program Agent* are both powered by GPT-4o with different prompts. Following VADAR (Marsili et al., 2025), we implement an API library that covers common visual subtasks recurring in spatial reasoning questions: (i) `loc`: object localization via Qwen2.5-VL-7B-Instruct (Bai et al., 2025) prompted with text and optional bounding box; (ii) `get_2D_object_size`: object size estimation from the segmentation mask, where SAM2 (Ravi et al., 2025) returns the mask/bounding box given an object prompt; (iii) `depth`: depth estimation at a queried image location using UniDepth (Piccinelli et al., 2024); (iv) `vqa`: attribute querying (e.g., color, material) using Qwen2.5-VL-7B-Instruct, with the target object bounding box overlaid on the image. We also include `same_object` checking whether two detections refer to the same instance by computing bounding-box overlap.

To support 3D spatial reasoning, we further provide a composite API `obj3d`, integrating UniDepth (2D-to-3D unprojection), GeoCalib (Veicht et al., 2024) (camera intrinsics estimation), and OrientAnything (Wang et al., 2025) (3D orientation estimation). This interface enables the agent to dynamically query object-level 3D information during execution. Full API definitions and prompts are provided in the supplementary material. All experiments are run on a NVIDIA H100 GPU.

4.2 Experimental Results

Table 1 reports performance on the evaluated subsets of 3DSRBench and Omni3D-Bench. Figure 3 illustrates the intermediate representations, plans, and programs during inference.

Comparison with visual-programming methods. GeoState achieves overall scores of 59.0% on 3DSRBench and 50.2% on Omni3D-Bench, exceeding VADAR by 25.7 and 9.8 percentage points, respectively, and VisProg by 32.9 and 29.1 points. These comparisons establish the performance of the complete system against the evaluated visual-programming baselines. The ablations in Section 4.3 separately examine the contributions of additional perception tools, their consolidation into object-centric geometry, and planning.

Comparison with direct VLM inference. Without additional task-specific fine-tuning, GeoState obtains the highest overall score among the evaluated VLM and visual-programming methods on both test sets. On 3DSRBench, it exceeds SpatialReasoner, the highest-scoring VLM overall in this comparison, by 3.9 percentage points. Its strongest category is multi-object reasoning, where it achieves 67.3% compared with SpatialReasoner's 53.1%. The advantage is not uniform: VST-7B-RL achieves higher height accuracy (68.4% versus 47.4%), and SpatialReasoner achieves higher

Table 2: Representation and planning ablation on the 3DSRBench subset (scores in percent). All variants use VADAR's error-message-based repair. Base is VADAR without its API generator. *3D APIs* adds GeoCalib and OrientAnything as separate 3D tools; `obj3d()` exposes the consolidated object-centric representation. Structured crash-report repair is evaluated in Appendix C.

Variant	Height	Location	Orientation	Multi-Object	Overall
Base	10.5	21.1	21.7	29.6	23.9
Base + 3D APIs	26.3	49.1	26.7	37.8	35.8
Base + `obj3d()`	42.1	61.4	30.0	61.2	51.7
Base + `obj3d()` + Planner	42.1	71.9	36.7	66.3	58.1



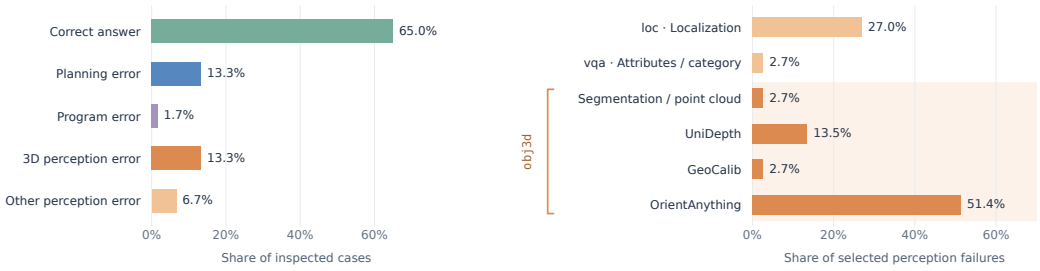
Figure 3: For each example, we show the query, input image, geometric scene representation, and the generated plan and program. Queries are drawn from 3DSRBench and cover diverse categories of 3D spatial reasoning. Please zoom in to read the programs.

orientation accuracy (48.3% versus 36.7%). These results place the main comparative strength in location and multi-object questions rather than all aspects of spatial perception.

On Omni3D-Bench, GeoState exceeds the highest overall VLM score of 45.0%, shared by Gemini 2.0 Flash and VST-7B-RL, by 5.2 percentage points. It achieves the highest scores on numeric questions other than counting and on multiple-choice questions, while GPT-4o performs better on counting and yes/no questions. Results on these human-written queries complement the template-based 3DSRBench evaluation.

4.3 Discussions

Ablation of Geometric Representation and Planning. Table 2 separates the benefit of additional 3D perception from that of the object-centric representation, with error-message-based program repair fixed. Base is VADAR without its API generator, using a single program agent and a limited tool library. Adding GeoCalib and OrientAnything as separate APIs improves overall accuracy from



(a) Overall outcomes ($N = 60$). (b) Perception failure sources ($N = 37$).

Figure 4: **Outcome and failure analysis. (a)** Outcomes of 60 randomly sampled cases from 3DSRBench, including both correct and incorrect predictions. Perception errors account for 20.0% of all cases: 3D perception errors refer to errors in our 3D representations, while other perception errors include failures in 2D grounding or VQA. **(b)** Sources of 37 separately selected failed cases involving perception errors only. The last four sources belong to `obj3d`, jointly accounting for 70.3%.

23.9% to 35.8%. Consolidating the available geometric predictions into the representation exposed by `obj3d()` further raises accuracy to 51.7% (+15.9 percentage points). The largest gain over separate APIs occurs on complicated multi-object questions, from 37.8% to 61.2% (+23.4 points), while orientation improves more modestly, from 26.7% to 30.0%. This supports the value of organizing perception outputs into compatible object attributes, particularly for computations involving several objects; access to additional perception models alone yields little improvement on multi-object questions over Base (29.6%).

Adding the planner improves accuracy to 58.1% (+6.4 points). Structured crash-report repair yields the full system's 59.0% under the same strict circular evaluation in Table 1. The effects of structured crash-report repair on accuracy and execution recovery are analyzed separately in Appendix C.

Failure Case Analysis. We manually inspect 60 randomly selected 3DSRBench cases and classify their outcomes as correct or belonging to one of four error categories. *3D perception errors* produce inaccurate position, orientation, or extent when constructing the intended object's representation; *other perception errors* concern object localization, identity matching, or attribute recognition. *Planning errors* select an incorrect geometric relation or procedure, while *program errors* incorrectly implement a correct plan or violate the output format. Each case receives one outcome label.

Figure 4a shows that 65.0% of the inspected cases are correct, with 13.3% exhibiting 3D perception errors, 6.7% other perception errors, 13.3% planning errors, and 1.7% program errors. Representation construction and planning are the largest individual error categories, suggesting that obtaining accurate geometry and choosing the correct operations over it remain challenging for spatial questions.

To diagnose perception errors further, we separately inspect 37 selected perception failures from 3DSRBench, attributing each to the first API or internal component that produces an incorrect output. Figure 4b presents the perception error distribution. Representation construction accounts for 70.3% with orientation estimation contributes most. The remaining failures arise in localization (27.0%) and attribute recognition (2.7%). Orientation is therefore the principal improvement target within these perception failures, particularly under front–back ambiguity, occlusion, and unusual viewpoints.

5 Conclusion

We introduced GeoState, a geometry-centric visual-programming framework that consolidates heterogeneous monocular perception outputs into a compact, object-centric 3D representation. Exposed through `obj3d()`, this representation lets generated programs compute spatial relations directly over compatible geometric attributes, with structured planning guiding their composition. Without task-specific spatial reasoning supervision or fine-tuning, GeoState outperforms the evaluated visual-programming baselines and is competitive with spatially fine-tuned VLMs on the evaluated subsets of 3DSRBench and Omni3D-Bench. Controlled ablations show that consolidating outputs from the same perception models improves accuracy over exposing them separately. These results support the choice of intermediate representation as an important factor in executable 3D spatial reasoning.

AI USE STATEMENT

In this work, we used generative AI tools for interpreting results. We have not used generative AI tools for helping develop theoretical models or conceptual frameworks, formulating mathematical claims, propose or refine hypotheses, designing or provide feedback on research methodology or experiments, implementing methods, assisting with translation, cleaning and reformating dataset, supporting qualitative and thematic data analysis, and generating synthetic data sets, providing critical ingredients for proving mathematical claims, assisting in the writing of proofs are not applicable to this work. Additionally, we used generative AI tools for creating or modifying scientific figures or images, suggesting experimental parameters, creating or editing software code, drafting parts of a research paper, summarizing or analyse existing literature, brainstorming, sourcing/searching for information, editing a research paper to improve readability, identifying relevant literature, suggesting a structure for a research paper, proposing a title or keywords for a research paper. We have reviewed all AI-assisted work. We checked LLM-generated texts through manual verification by authors. We use AI tools to generate scientific figures for inspiration and we draw the figures in the paper by authors with non-AI tools. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibo Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, Humen Zhong, Yuanzhi Zhu, Mingkun Yang, Zhaohai Li, Jianqiang Wan, Pengfei Wang, Wei Ding, Zheren Fu, Yiheng Xu, Jiabo Ye, Xi Zhang, Tianbao Xie, Zesen Cheng, Hang Zhang, Zhibo Yang, Haiyang Xu, and Junyang Lin. Qwen2.5-vl technical report. *arXiv preprint arXiv:2502.13923*, 2025.

Zhipeng Cai, Ching-Feng Yeh, Hu Xu, Zhuang Liu, Gregory P. Meyer, Xinjie Lei, Changsheng Zhao, Shang-Wen Li, Vikas Chandra, and Yangyang Shi. DepthLM: Metric depth from vision language models. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=ObFVZGnSFN.

Boyuan Chen, Zhuo Xu, Sean Kirmani, Brain Ichter, Dorsa Sadigh, Leonidas Guibas, and Fei Xia. Spatialvlm: Endowing vision-language models with spatial reasoning capabilities. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 14455–14465, June 2024.

Zhangquan Chen, Ruihui Zhao, Chuwei Luo, Mingze Sun, Xinlei Yu, Yangyang Kang, and Ruqi Huang. Sifthinker: Spatially-aware image focus for visual reasoning. In *AAAI*, pp. 20436–20444, 2026. URL https://doi.org/10.1609/aaai.v40i25.39178.

An-Chieh Cheng, Hongxu Yin, Yang Fu, Qiushan Guo, Ruihan Yang, Jan Kautz, Xiaolong Wang, and Sifei Liu. Spatialrgpt: Grounded spatial reasoning in vision-language models. In *NeurIPS*, 2024.

An-Chieh Cheng, Yang Fu, Yukang Chen, Zhijian Liu, Xiaolong Li, Subhashree Radhakrishnan, Song Han, Yao Lu, Jan Kautz, Pavlo Molchanov, Hongxu Yin, Xiaolong Wang, and Sifei Liu. 3d aware region prompted vision language model. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=GTpf2NuwtR.

Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, pp. 226–231. AAAI Press, 1996.

Zhiwen Fan, Jian Zhang, Renjie Li, Junge Zhang, Runjin Chen, Hezhen Hu, Kevin Wang, Huaizhi Qu, Dilin Wang, Zhicheng Yan, Hongyu Xu, Justin Theiss, Tianlong Chen, Jiachen Li, Zhengzhong Tu, Zhangyang Wang, and Rakesh Ranjan. Vlm-3r: Vision-language models augmented with instruction-aligned 3d reconstruction, 2025. URL https://arxiv.org/abs/2505.20279.

Stephanie Fu, tyler bonnen, Devin Guillory, and Trevor Darrell. Hidden in plain sight: VLMs overlook their visual representations. In *Second Conference on Language Modeling*, 2025.

Mohsen Gholami, Ahmad Rezaei, Zhou Weimin, Sitong Mao, Shunbo Zhou, Yong Zhang, and Mohammad Akbari. Spatial reasoning with vision-language models in ego-centric multi-view scenes. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=fqehqG4WvL.

Tanmay Gupta and Aniruddha Kembhavi. Visual programming: Compositional visual reasoning without training. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 14953–14962, June 2023.

Xiaohu Huang, Jingjing Wu, Qunyi Xie, and Kai Han. 3drs: Mllms need 3d-aware representation supervision for scene understanding. In *Conference on Neural Information Processing Systems*, 2025.

Baoxiong Jia, Yixin Chen, Huangyue Yu, Yan Wang, Xuesong Niu, Tengyu Liu, Qing Li, and Siyuan Huang. Sceneverse: Scaling 3d vision-language learning for grounded scene understanding. In *European Conference on Computer Vision (ECCV)*, 2024.

Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics*, 42(4), July 2023. URL https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/.

Alex H. Lang, Sourabh Vora, Holger Caesar, Lubing Zhou, Jiong Yang, and Oscar Beijbom. Pointpillars: Fast encoders for object detection from point clouds. In *CVPR*, 2019.

Phillip Y. Lee, Jihyeon Je, Chanho Park, Mikaela Angelina Uy, Leonidas Guibas, and Minhyuk Sung. Perspective-aware reasoning in vision-language models via mental imagery simulation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2025.

Shijie Lian, Changti Wu, Laurence Tianruo Yang, Hang Yuan, Bin Yu, Lei Zhang, and Kai Chen. Euclid's gift: Enhancing spatial perception and reasoning in vision-language models via geometric surrogate tasks, 2025. URL https://arxiv.org/abs/2509.24473.

Yuan Liu, Haodong Duan, Yuanhan Zhang, Bo Li, Songyang Zhang, Wangbo Zhao, Yike Yuan, Jiaqi Wang, Conghui He, Ziwei Liu, et al. Mmbench: Is your multi-modal model an all-around player? In *European conference on computer vision*, pp. 216–233. Springer, 2024.

Shuo Lu, Jianjie Cheng, Yinuo Xu, Yongcan Yu, Lijun Sheng, Peijie Wang, Siru Jiang, Yongguan Hu, Run Ling, Yihua Shao, Ao Ma, Wei Feng, Lingxiao He, Meng Wang, Qianlong Xie, Xingxing Wang, Ran He, and Jian Liang. Do mllms really understand space? a mathematical reasoning evaluation, 2026. URL https://arxiv.org/abs/2602.11635.

Wufei Ma, Angtian Wang, Alan Yuille, and Adam Kortylewski. Robust category-level 6d pose estimation with coarse-to-fine rendering of neural features. In Shai Avidan, Gabriel Brostow, Moustapha Cissé, Giovanni Maria Farinella, and Tal Hassner (eds.), *Computer Vision – ECCV 2022*, pp. 492–508, Cham, 2022. Springer Nature Switzerland. ISBN 978-3-031-20077-9.

Wufei Ma, Haoyu Chen, Guofeng Zhang, Yu-Cheng Chou, Jieneng Chen, Celso de Melo, and Alan Yuille. 3dsrbench: A comprehensive 3d spatial reasoning benchmark. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 6924–6934, October 2025a.

Wufei Ma, Yu-Cheng Chou, Qihao Liu, Xingrui Wang, Celso M de Melo, Jianwen Xie, and Alan Yuille. Spatialreasoner: Towards explicit and generalizable 3d spatial reasoning. In *Advances in Neural Information Processing Systems*, volume 38, 2025b.

Damiano Marsili, Rohun Agrawal, Yisong Yue, and Georgia Gkioxari. Visual agentic ai for spatial reasoning with a dynamic api. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pp. 19446–19455, 2025.

OpenAI. Gpt-4o system card, 2024. URL https://arxiv.org/abs/2410.21276.

Luigi Piccinelli, Yung-Hsu Yang, Christos Sakaridis, Mattia Segu, Siyuan Li, Luc Van Gool, and Fisher Yu. UniDepth: Universal monocular metric depth estimation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024.

Nikhila Ravi, Valentin Gabeur, Yuan-Ting Hu, Ronghang Hu, Chaitanya Ryali, Tengyu Ma, Haitham Khedr, Roman Rädle, Chloe Rolland, Laura Gustafson, Eric Mintun, Junting Pan, Kalyan Vasudev Alwala, Nicolas Carion, Chao-Yuan Wu, Ross Girshick, Piotr Dollar, and Christoph Feichtenhofer. SAM 2: Segment anything in images and videos. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=Ha6RTeWMd0.

Dídac Surís, Sachit Menon, and Carl Vondrick. Vipergpt: Visual inference via python execution for reasoning. *Proceedings of IEEE International Conference on Computer Vision (ICCV)*, 2023.

Gemini Team. Gemini: A family of highly capable multimodal models, 2025. URL https://arxiv.org/abs/2312.11805.

Shengbang Tong, Ellis Brown, Penghao Wu, Sanghyun Woo, Manoj Middepogu, Sai Charitha Akula, Jihan Yang, Shusheng Yang, Adithya Iyer, Xichen Pan, Austin Wang, Rob Fergus, Yann LeCun, and Saining Xie. Cambrian-1: A fully open, vision-centric exploration of multimodal llms, 2024.

Alexander Veicht, Paul-Edouard Sarlin, Philipp Lindenberger, and Marc Pollefeys. GeoCalib: Single-image Calibration with Geometric Optimization. In *ECCV*, 2024.

Tai Wang, Xinge Zhu, Jiangmiao Pang, and Dahua Lin. FCOS3D: Fully convolutional one-stage monocular 3d object detection. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) Workshops*, 2021.

Xingrui Wang, Wufei Ma, Zhuowan Li, Adam Kortylewski, and Alan Yuille. 3d-aware visual question answering about parts, poses and occlusions. *arXiv preprint arXiv:2310.17914*, 2023.

Zehan Wang, Ziang Zhang, Tianyu Pang, Chao Du, Hengshuang Zhao, and Zhou Zhao. Orient anything: Learning robust object orientation estimation from rendering 3d models. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=x4yTgv2WkJ.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025a. URL https://arxiv.org/abs/2505.09388.

Jihan Yang, Shusheng Yang, Anjali W. Gupta, Rilyn R. Han, Li Fei-Fei, and Saining Xie. Thinking in space: How multimodal large language models see, remember, and recall spaces. In *Proceedings of the Computer Vision and Pattern Recognition Conference (CVPR)*, 2025b. URL https://arxiv.org/abs/2412.14171.

Rui Yang, Ziyu Zhu, Yanwei Li, Jingjia Huang, Shen Yan, Siyuan Zhou, Zhe Liu, Xiangtai Li, Shuangye Li, Wenqian Wang, Yi Lin, and Hengshuang Zhao. Visual spatial tuning. In *Computer Vision – ECCV 2026*, pp. 192–211. Springer Nature Switzerland, 2026.

Yuncong Yang, Jiageng Liu, Zheyuan Zhang, Siyuan Zhou, Reuben Tan, Jianwei Yang, Yilun Du, and Chuang Gan. Mindjourney: Test-time scaling with world models for spatial reasoning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025c. URL https://openreview.net/forum?id=L2W4wQsNkY.

Songsong Yu, Yuxin Chen, Hao Ju, Lianjie Jia, Fuxi Zhang, Shaofei Huang, Yuhan Wu, Rundi Cui, Binghao Ran, Zaibin Zhang, Zhedong Zheng, Zhipeng Zhang, Yifan Wang, Lin Song, Lijun Wang, Yanwei Li, Ying Shan, and Huchuan Lu. How far are vlms from true visual spatial intelligence? a benchmark-driven perspective. *arXiv preprint arXiv:2509.18905*, 2025.

Xiaoyu Zhan, Wenxuan Huang, Hao Sun, Xinyu Fu, Changfeng Ma, Shaosheng Cao, Bohan Jia, Shaohui Lin, Zhenfei Yin, LEI BAI, Wanli Ouyang, Yuanqi Li, Jie Guo, and Yanwen Guo. Actial: Activate spatial reasoning ability of multimodal large language models. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=jquTBzt3Av.

Fangrui Zhu, Hanhui Wang, Yiming Xie, Jing Gu, Tianye Ding, Jianwei Yang, and Huaizu Jiang. Struct2d: A perception-guided framework for spatial reasoning in mllms. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 38, 2025.

Supplementary Material

A LIMITATIONS

GeoState inherits limitations from current 3D perception systems. Errors from pre-trained vision modules can propagate to the constructed geometric scene representation and affect downstream reasoning. Our design makes this dependency explicit and inspectable, rather than obscuring it within an end-to-end VLM. Our failure analysis shows that many errors can be traced to specific components, suggesting a clear path for improvement as these modules advance.

GeoState is designed for object-centric spatial reasoning from static single image. It may therefore struggle with cases requiring temporal evidence, multi-view consistency, or reasoning about fully occluded objects, which are outside the scope of our benchmark and claims. Within the evaluated setting, GeoState improves spatial reasoning by grounding language queries in 3D geometric abstractions.

Finally, GeoState adopts a modular pipeline rather than a single end-to-end model, which introduces additional inference overhead and dependence on external components. This modularity makes the system training-free, interpretable, and can benefit directly from future improvements in visual grounding and 3D perception. Future work could incorporate uncertainty-aware reasoning, stronger perception backbones, and multi-view or video-based scene representations.

B IMPLEMENTATION DETAILS OF `OBJ3D`

Given an RGB image $I \in \mathbb{R}^{H\times W\times 3}$ and a 2D bounding box b_i for object i, the composite operator

$$\texttt{obj3d}(I, b_i) \rightarrow B_i^{3D} \tag{6}$$

constructs an oriented 3D bounding-box proxy

$$B_i^{3D} = (\ell_i, \mathbf{c}_i, \mathbf{R}_i, \mathbf{s}_i, h_i^g), \tag{7}$$

where ℓ_i is the semantic label inherited from the object query, $\mathbf{c}_i \in \mathbb{R}^3$ is the object center, $\mathbf{R}_i$ contains the orthonormal object-left, object-up, and object-front directions in the camera frame, and $\mathbf{s}_i \in \mathbb{R}^3$ denotes the box extents along the oriented axes. The additional scalar h_i^g is the independently estimated gravity-aligned height exposed as `3d_height`; it does not change the camera-centric coordinate frame of the other attributes.

Camera intrinsics and metric lifting. We first estimate camera intrinsics $\hat{\mathbf{K}} \in \mathbb{R}^{3\times 3}$ from the input image using GeoCalib (Veicht et al., 2024). We then predict a metric depth map $D \in \mathbb{R}^{H\times W}$ with UniDepth (Piccinelli et al., 2024). For each pixel (u, v), the corresponding 3D point in the camera-centric coordinate frame is obtained by back-projection:

$$\mathbf{p}_{uv} = D(u, v)\, \hat{\mathbf{K}}^{-1}[u, v, 1]^\top \in \mathbb{R}^3. \tag{8}$$

Object masking and point-cloud extraction. Given the box b_i, SAM2 predicts a binary object mask $M_i \in \{0, 1\}^{H\times W}$. We construct an object point cloud by selecting the back-projected points inside the mask:

$$\mathcal{P}_i = \{\mathbf{p}_{uv} \mid M_i(u, v) = 1\}. \tag{9}$$

To improve robustness, we denoise $\mathcal{P}_i$ using statistical outlier removal, voxel downsampling, and DBSCAN-based clustering (Ester et al., 1996), retaining only the largest connected cluster.

Orientation estimation. The image crop around b_i is fed to OrientAnything (Wang et al., 2025), which predicts discrete azimuth α, elevation ε, and in-plane rotation ϑ, together with a confidence score $\kappa \in [0, 1]$. These angles are converted into three orthonormal object directions in the camera frame, corresponding to the object's left, up, and front axes.

Oriented box fitting. From the predicted viewpoint angles, we construct an object orientation matrix

$$\mathbf{R}_i = [\hat{\ell}_i, \hat{u}_i, \hat{f}_i], \tag{10}$$

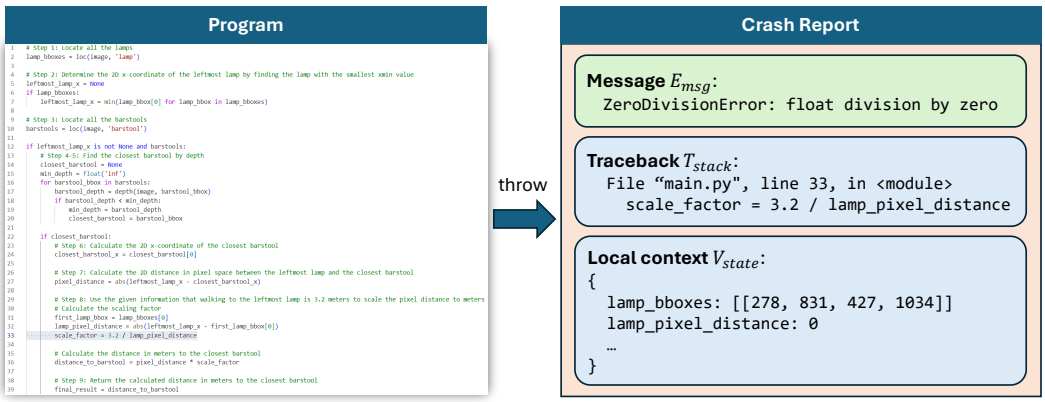


Figure 5: Structured crash report when program throws exception. Most existing visual programming frameworks only collect error message for program repair. Our crash report not only includes error message, but also function traceback and local variables in running environment to make program agent better regenerate program.

where $\hat{\ell}_i$, $\hat{u}_i$, and $\hat{f}_i$ are the unit left, up, and front directions of the object expressed in camera coordinates. We then project the object point cloud $\mathcal{P}_i$ onto this oriented frame, compute the axis-aligned extrema along the three axes, and recover the box center $\mathbf{c}_i$ and extents $\mathbf{s}_i$ from the resulting bounds.

Gravity-aligned 3D height. In addition to camera intrinsics, GeoCalib (Veicht et al., 2024) could also estimate the gravity direction from I. We use its predicted camera roll and pitch, with yaw fixed to zero, to construct a rotation $\mathbf{R}_g$ from the camera frame to a gravity-aligned frame whose positive y-axis points upward, opposite to gravity. This rotation is distinct from the object orientation $\mathbf{R}_i$. The gravity-aligned scene points are

$$\mathbf{q}_{uv} = \mathbf{R}_g \mathbf{p}_{uv}. \tag{11}$$

Here, $\mathbf{R}_g$ is expressed in the same camera-coordinate convention as $\mathbf{p}_{uv}$; any coordinate-axis sign conversions used internally are absorbed into this transformation. Let $\mathbf{e}_y = (0, 1, 0)^\top$ and $\mathbf{e}_z = (0, 0, 1)^\top$ denote the upward and forward axes in the aligned frame. To estimate a ground reference, we select the $\lfloor HW/2 \rfloor$ scene pixels with the smallest aligned forward coordinates $\mathbf{e}_z^\top \mathbf{q}_{uv}$, denoting their index set by Ω_{near}. We then take their minimum vertical coordinate:

$$\hat{g} = \min_{(u,v)\in\Omega_{\text{near}}} \mathbf{e}_y^\top \mathbf{q}_{uv}. \tag{12}$$

This provides a heuristic ground elevation from the full, unfiltered scene reconstruction, without explicitly fitting a ground plane. Using $\mathcal{P}_i$ to denote the object point cloud after the denoising steps above, we compute

$$h_i^g = \max_{\mathbf{p}\in\mathcal{P}_i} \mathbf{e}_y^\top \mathbf{R}_g \mathbf{p} - \hat{g}. \tag{13}$$

Thus, h_i^g measures the highest retained object point above the estimated ground reference, rather than the object's intrinsic vertical extent. Its physical interpretation assumes that the selected scene subset contains a reliable ground reference and that the retained object points capture its top. Gravity alignment is used only for this scalar; $\mathbf{c}_i$, $\mathbf{R}_i$, and $\mathbf{s}_i$ retain their camera-centric representation.

Together this yields the final object-centric geometric representation

$$B_i^{3D} = (\ell_i, \mathbf{c}_i, \mathbf{R}_i, \mathbf{s}_i, h_i^g). \tag{14}$$

C Effect of Execution-Guided Program Repair

We separate repair from the representation and planning ablations by fixing the `obj3d()` representation and planner. *No repair* executes the initial program once; *error-message repair* revises failed programs using the exception message; and *structured crash-report repair* additionally supplies

Table 3: Repair comparison with the object-centric representation and planner fixed. Accuracy and final execution-failure rate are percentages; repair calls are averaged per query. Zero repair calls in the no-repair condition is fixed by design.

Repair strategy	Accuracy	Execution failures	Mean repair calls
No repair	18.9	32.8	0
Error message	58.8	4.0	0.31
Structured crash report	60.3	2.0	0.27

the traceback and relevant local state. Figure 5 illustrates an example of our crash report. This comparison distinguishes the benefit of execution-guided retries from the additional value of richer diagnostic context.

Table 3 reports average answer accuracy without the strict circular-evaluation criterion, the percentage of queries ending in execution failure, and mean repair calls per query. Execution failures are queries that do not complete successfully, rather than all incorrect answers; repair calls are averaged over all queries, including those requiring no repair. These accuracy scores are therefore distinct from the strict circular-evaluation results reported in the main experiments.

Error-message repair raises accuracy from 18.9% to 58.8% and reduces final execution failures from 32.8% to 4.0%, demonstrating the practical importance of recovery in this setting. Providing structured crash reports further improves accuracy to 60.3% (+1.5 percentage points), halves the remaining execution-failure rate to 2.0%, and reduces mean repair calls from 0.31 to 0.27. The richer context thus supports more effective recovery with fewer repair calls on average. Under strict circular evaluation, the corresponding accuracy improvement is from 58.1% to 59.0% (+0.9 points). Repair addresses execution failures; it does not guarantee correct geometric estimates or reasoning in programs that execute successfully.

D SYSTEM-LEVEL METRICS

We report system-level metrics on 3DSRBench to characterize the efficiency and reliability of GeoState.

Table 4: Runtime for each Agent in seconds per query.

Planner	Program	Execution
3.35 ± 0.96	5.67 ± 2.46	14.21 ± 5.1

Runtime. Table 4 reports the average runtime of each stage on a NVIDIA H100 GPU. The planner introduces modest overhead, taking 3.35 seconds per query on average, while execution dominates the total runtime due to calls to external perception and geometric APIs. This suggests that the main computational cost comes from visual grounding and 3D perception rather than program design and generation.

`obj3d` usage. Table 5 shows that generated programs call the `obj3d` API 1.94 times per query on average. Multi-object questions require the most frequent 3D queries, reflecting the need to compare geometric attributes across objects. In contrast, orientation questions require fewer calls, as they often involve reasoning about a single target object.

Table 5: Average number of `obj3d` API calls per query on 3DSRBench.

Height	Location	Orientation	Multi-Object	Overall
1.94	1.87	1.48	2.28	1.94

Execution reliability. Table 6 summarizes system-level execution outcomes. GeoState successfully completes 98.0% of queries on 3DSRBench. Among the remaining cases, 0.9% fail due to program

synthesis failures and 1.1% fail because the perception API cannot detect objects mentioned in the question. These results indicate that the modular program-based framework is reliable in practice, with most failures attributable to either perception coverage or residual program-generation errors.

Table 6: System-level execution outcome on 3DSRBench.

Success	Program Failures	Perception Failures
98.0%	0.9%	1.1%

Table 7: Average performance with standard deviation of GeoState under 3 repeated program generation.

	Height	Location	Orientation	Multi-Obejct	Overall
Performance	54.4±6.1	70.7±2.0	37.2±0.9	63.3±2.1	58.7±0.9

E Additional Quantitative Analysis

Experimental Variability. We evaluate the stability of GeoState by running the same program-generation model three times and reporting the average performance with their standard deviations across runs. Table 7 shows relatively stable overall performance, with an standard deviation 0.9. Location, orientation, and multi-object reasoning also exhibit low variance, suggesting that the generated programs are generally consistent across runs. Height reasoning shows larger variation, indicating that this category is more sensitive to stochasticity in program generation or downstream perceptual estimates. Overall, across three program-generation runs on the same evaluation subset, our method presents stable aggregate performance under repeated generation in this setting.

Comparison with visual programming using `obj3d`. We augment VADAR (Marsili et al., 2025) with `obj3d()` to compare reasoning frameworks with access to the same geometric representation. Table 8 shows that GeoState achieves 59.0% overall accuracy versus 53.4% for VADAR+`obj3d()`, a gain of 5.6 percentage points, with higher scores in every reported category. The largest gain is on multi-object questions (+8.1 points). This suggests that the proposed geometric representation also benefits an existing visual-programming framework. Together, these comparisons support both the utility of the representation and the effectiveness of GeoState's pipeline for composing reasoning operations over it.

Sensitivity to the agent backbone. Table 9 compares GeoState and VADAR on the evaluated 3DSRBench subset using GPT-4o (OpenAI, 2024) and Qwen3-4B-Instruct (Yang et al., 2025a) as agent backbones. GeoState achieves higher accuracy than VADAR in every reported category under both settings, with overall margins of 25.7 and 26.9 percentage points, respectively. Its advantage over VADAR therefore persists across the two tested backbones.

Switching from GPT-4o to Qwen3-4B-Instruct reduces overall accuracy by 12.0 points for GeoState (59.0% to 47.0%) and 13.2 points for VADAR (33.3% to 20.1%). Sensitivity varies by category: the multi-object drop is smaller for GeoState (22.4 versus 29.6 points), while its height drop is larger (10.6 versus 5.2 points).

Although both methods are sensitive to the agent backbone, GeoState maintains a substantial performance advantage under both tested settings. These results show that the framework remains effective with the smaller open-source agent, extending its benefit beyond the GPT-4o setting.

Table 8: Comparison with VADAR augmented with the same `obj3d` API on 3DSRBench.

Method	Height	Location	Orientation	Multi-Object	Overall
VADAR+`obj3d`	42.1	68.4	33.3	59.2	53.4
Ours	**47.4**	**71.9**	**36.7**	**67.3**	**59.0**

Table 9: Sensitivity to LLM backbone choice for visual programming methods. GeoState outperforms VADAR with both tested backbones and both systems have lower overall accuracy with Qwen3-4B-Instruct than with GPT-4o.

Model	Backbone	Height	Location	Orientation	Multi-Object	Mean
VADAR	Qwen3-4B-Instruct	31.6	40.4	16.7	8.2	20.1
	GPT4o	36.8	43.9	15.0	37.8	33.3
GeoState	Qwen3-4B-Instruct	36.8	70.2	31.7	44.9	47.0
	GPT4o	47.4	71.9	36.7	67.3	59.0

F Qualitative Examples for Omni3D-Bench

Figure 6 shows additional qualitative results on Omni3D-Bench. Given a single image, GeoState can construct a robust 3D geometric scene representation and generates a plan together with an executable program to answer each query. Notably, the generated programs closely follow the queried reasoning steps, indicating that decomposing program synthesis into plan synthesis and plan-guided program generation leads to semantically coherent and geometrically grounded reasoning. These qualitative results help explain the strong quantitative performance of GeoState: the model explicitly separates scene abstraction from program generation, which reduces ambiguity in complex spatial reasoning and yields outputs that are both accurate and interpretable.

G Prompts

Predefined Module Signatures. Figure 7 presents the docstrings of the predefined modules in GeoState. In addition to standard image-perception modules, GeoState includes a `obj3d` module for querying object-centric 3D geometric representations directly from an image. Together, these module signatures define the action space for both plan synthesis and program synthesis.

Prompts for Plan Synthesis. Figure 8 and Figure 9 show the prompts used by the Planner Agent for 3DSRBench and Omni3D-Bench, respectively. Given a user query, the planner produces a high-level natural-language plan that decomposes the reasoning process into executable steps. We include in-context examples to improve robustness and consistency. For Omni3D-Bench, we adopt the same in-context examples as VADAR for fair comparison.

Prompts for Program Synthesis. Figure 10 and Figure 11 present the prompts used for program synthesis on 3DSRBench and Omni3D-Bench, respectively. The Program Agent takes the user question and the generated plan as input and synthesizes an executable program over the predefined modules. As in plan synthesis, we use in-context examples, augmented with their corresponding programs, to guide generation.



Figure 6: Qualitative results of GeoState on Omni3D-Bench. Each example shows the query, input image, geometric scene representation, generated plan, and synthesized executable program. The queries require complex 3D spatial reasoning. From a single image, GeoState can build an abstract 3D scene representation and generates effective plans and executable programs to answer each query. Please zoom in for readability.

```
\"\"\"
Locates objects in an image. Object prompts should be simple and contain few words. To return all objects,
     pass "objects" as the prompt.

Args:
    image (image): Image to search.
    object_prompt (string): Description of object to locate.
Returns:
    list: A list of bounding boxes [xmin, ymin, xmax, ymax] for all of the objects located in pixel space.
\"\"\"
def loc(image, object_prompt):

\"\"\"
Answers a natural-language question about the image, optionally restricted to a region.

Args:
    image (image): Image of the scene.
    question (string): Question about the object in the bounding box.
    bbox (list | None, optional): If provided, a bounding box [xmin, ymin, xmax, ymax] in pixel coordinates
      that specifies the region of interest. The answer should be based primarily on visual evidence inside
      this region. If omitted or None, the question is answered using the entire image (global VQA).


Returns:
    string: Answer to the question about the object in the image.
\"\"\"
def vqa(image, question, bbox):

\"\"\"
Returns the depth of an object (specified by a bounding box) in meters.

Args:
    image (image): Image of the scene.
    bbox (list): A bounding box [xmin, ymin, xmax, ymax] containing the object.

Returns:
    float: The depth of the object specified by the coordinates in meters.
\"\"\"
def depth(image, bbox):

\"\"\"
Estimate the 3D position and orientation axes of the object inside `bbox`.

Args:
    image (image): Image of the scene.
    bbox (list): A bounding box [xmin, ymin, xmax, ymax] containing the object.

Returns:
    dict: A dictionary with the following keys:
        - "3d_height" (float): Estimated height of object in meters along the world up direction.
        - "3d_location" (list[float]): Length-3 list [x, y, z] giving the estimated 3D position of the object'
     s center in the camera coordinate frame, in meters. Convention: camera is at [0, 0, 0]; x is right, y is
      up, z is forward (depth).
        - "direction_left" (list[float]): Length-3 unit vector pointing to the object's left in the same 3D
     coordinate frame.
        - "direction_front" (list[float]): Length-3 unit vector pointing to the object's front (facing
     direction) in the same frame.
        - "direction_up" (list[float]): Length-3 unit vector pointing to the object's upward direction in the
      same frame.
        - "3d_size" (list[float]): Length-3 list [x, y, z] indicating the 3D boudning box size for the object
      along the direction left, up and front.

\"\"\"
def obj3d(image, bbox):

\"\"\"
Checks if two bounding boxes correspond to the same object.

Args:
    image (image): Image of the scene.
    bbox1 (list): A bounding box [xmin, ymin, xmax, ymax] containing object1.
    bbox2 (list): A bounding box [xmin, ymin, xmax, ymax] containing object2.

Returns:
    bool: True if object 1 is the same object as object 2, False otherwise.
\"\"\"
def same_object(image, bbox1, bbox2):

\"\"\"
Returns the width and height of the object in 2D pixel space.

Args:
    image (image): Image of the scene.
    bbox (list): A bounding box [xmin, ymin, xmax, ymax] containing the object.

Returns:
    tuple: (width, height) of the object in 2D pixel space.
\"\"\"
def get_2D_object_size(image, bbox):
```

Figure 7: **Predefined Modules for GeoState.** These modules define the interfaces available to the planner and program agents.

```
You are an expert logician capable of answering spatial reasoning problems. You excel at breaking down
difficult spatial and visual questions into clear, logical step-by-step plans that can later be implemented as
 code using a predefined API.

I am going to give you several examples of how you might approach a problem in pseudocode steps, then I will
give you a question for you to plan a solution for.

Example 1:
Question: "Which object is closer to the camera? A. Toilet B. Faucet"
Plan:
<plan>
1) Locate all the Toilet (toilet_bboxes=loc(image, "toilet"))
2) Locate all the Faucet (faucet_bboxes=loc(image, "faucet"))
3) Get 3D pose for the first toilet found (toilet_pose=obj3d(image, toilet_bboxes[0]))
4) Get 3D pose for the first faucet found (faucet_pose=obj3d(image, faucet_bboxes[0]))
5) Extract 3D location of toilet from its 3D pose (toilet_3d_location=toilet_pose['3d_location'])
6) Extract 3D location of faucet from its 3D pose (faucet_3d_location=faucet_pose['3d_location'])
7) Assume the camera is at [0, 0, 0].
8) Compute L2 distances from toilet to camera.
9) Compute L2 distances from faucet to camera.
10) Return A if the distance from toilet to camera is smaller than the distance from faucet to camera,
otherwise return B.
</plan>

Example 2:
Question: "Which side of the cat is facing the backpack? A. front B. left C. back D. right"
Plan:
<plan>
1) Locate all the cats (cat_bboxes = loc(image, "cat"))
2) Locate all the backpacks (bp_bboxes = loc(image, "backpack"))
3) Get 3D pose for the first cat found (cat_pose = obj3d(image, cat_bboxes[0]))
4) Get 3D pose for the first backpack found (bp_pose = obj3d(image, bp_bboxes[0]))
5) Extract 3D location, the 3D front direction and the 3D left direction of cat
(cat_loc=cat_pose['3d_location'],
cat_front=cat_pose['direction_front'],
cat_left=cat_pose['direction_left'])
6) Extract 3D location of backpack (bp_loc=bp_pose['3d_location'])
7) Compute 3D direction vector from cat to backpack (dir_cat_to_bp=bp_loc-cat_loc)
8) Compute cosine similarities between direction from cat to backpack and the 3D front, back, left and right
directions of cat respectively.
9) Pick the side of cat with the maximum similarity.
10) Return the corresponding option letter.
</plan>

Now here is an API of methods, you will want to solve the problem in a logical and sequential manner as I
showed you.

------------------ API ------------------
{predef_signatures}
------------------ API ------------------

Please do not use synonyms, even if they are present in the question.
Output your step-by-step solution plan inside the tags <plan></plan>.
Ensure that the final step always specifies what option letter to return as the answer.
Ensure that the answer is just one option letter.

Here are some helpful definitions:
1) 2D distance/size refers to distance/size in pixel space.
2) 3D distance/size refers to distance/size in the real world. 3D size is equal to 2D size times the depth of
the object.
3) We define (width, height, depth) as the values along the (x, y, z) axis. Width = x axis, height = y axis,
depth = z axis.
4) The camera is always located at [0, 0, 0] in real-world 3D space.
5) The real-world 3D up direction is always [0, 1, 0].
6) "Depth" measures distance from the camera in 3D.

Here are some helpful tips:
1) When you need to search over objects satisfying a condition, remember to check all the objects that satisfy
 the condition and don't just return the first one.
2) When searching for objects to compare to a reference object, make sure to remove the reference object from
the retrieved objects. You can check if two objects are the same with the same_object method.
3) If two objects have the same 2D width, then the object with the largest depth has the largest 3D width.
4) If two objects have the same 2D height, then the object with the largest depth has the largest 3D height.
5) 2D sizes convey the height and width in IMAGE SPACE. To convert to height and width in 3D space, it needs
to be multiplied by the depth!
6) Some questions may present hypothesis measurements (e.g. "X is 3.5m wide"), these are hypothesis
measurements and should be used ONLY to scale your outputs accordingly.

AGAIN, output your step-by-step solution plan inside the tags <plan></plan> and ensure the final step
specifies what option letter to return.

<question>{question}</question>
```

Figure 8: **Plan Agent Prompt for GeoState on 3DSRBench.**

```
You are an expert logician capable of answering spatial reasoning problems. You excel at breaking down
difficult spatial and visual questions into clear, logical step-by-step plans that can later be implemented as
 code using a predefined API.

I am going to give you examples of how you might approach a problem in pseudocode steps, then I will give you
an API and some instructions for you to answer with a plan.

Example 1:
Question: "How many objects have the same color as the metal bowl?"
Plan:
<plan>
1) Set a counter to 0
2) Find all the bowls (loc(image, 'bowls')).
3) If bowls are found, loop through each of the bowls found.
4) For each bowl found, check if the material of this bowl is metal. Store the metal bowl if you find it and
break from the loop.
5) Find and store the color of the metal bowl.
6) Find all the objects.
7) For each object O, check if O is the same object as the small bowl (same_object(image, metal_bowl_bbox,
object_bbox)). If it is, skip it.
8) For each O you don't skip, check if the color of O is the same as the color of the metal bowl.
9) If it is, increment the counter.
10) When you are done looping, return the counter.
</plan>

Example 2:
Question: "How many objects of the same height as the mug would you have to stack to achieve an object the
same height as the cabinet?"
Plan:
<plan>
1) Locate the mug (loc(image, "mug"))
2) Locate the cabinet (loc(image, "cabinet"))
3) Find the 2D height of the mug, multiply it by the depth and store this value.
4) Find the 2D height of the cabinet, multiply it by the depth and store this value.
5) Return the height of the cabinet divided by the height of the mug, do NOT round.
</plan>

Now here is an API of methods, you will want to solve the problem in a logical and sequential manner as I
showed you.

------------------- API -------------------
{predef_signatures}
------------------- API -------------------

Here are some helpful definitions:
1) 2D distance/size refers to distance/size in pixel space.
2) 3D distance/size refers to distance/size in the real world. 3D size is equal to 2D size times the depth of
the object.
3) We define (width, height, length) as the values along the (x, y, z) axis. Width = x axis, height = y axis,
length = z axis.
4) "Depth" measures distance from the camera in 3D.

Here are some helpful tips:
1) When you need to search over objects satisfying a condition, remember to check all the objects that satisfy
 the condition and don't just return the first one.
2) When searching for objects to compare to a reference object, make sure to remove the reference object from
the retrieved objects. You can check if two objects are the same with the same_object method.
3) If two objects have the same 2D width, then the object with the largest depth has the largest 3D width.
4) If two objects have the same 2D height, then the object with the largest depth has the largest 3D height.
5) 2D sizes convey the height and width in IMAGE SPACE. To convert to height and width in 3D space, it needs
to be multiplied by the depth!
6) Some questions may present hypothesis measurements (e.g. "X is 3.5m wide"), these are hypothesis
measurements and should be used ONLY to scale your outputs accordingly.
7) Do NOT round your answers! Always leave your answers as decimals even when it feels intuitive to round or
ceiling your answer - do not do it!
8) When a query asks to find all objects in a container just count the number of objects.

Please do not use synonyms, even if they are present in the question.
Output your step-by-step solution plan inside the tags <plan></plan>.
Ensure that the final step always specifies what value to return as the answer.
Ensure that the answer is either yes/no, one word, or one number.

AGAIN, output your step-by-step solution plan inside the tags <plan></plan> and ensure the final step
specifies what value to return.

<question>{question}</question>
```

Figure 9: **Plan Agent Prompt for GeoState on Omni3D-Bench.**

```
You are an expert logician capable of answering spatial reasoning problems with code. You excel at using a
      predefined API to implement a working program that answers spatial and complex reasoning problems.

You will be given a question and a step-by-step solution plan. Your job is to use the provided plan as a
      guiding reference and write a correct program using the provided API. The plan may contain errors or
      imprecisions - treat it as a helpful suggestion rather than a strict specification. If you spot a
      logical error or a better way to achieve the same goal, correct it in your implementation.

I am going to give you one example of how a question and plan maps to a program, then I will give you an API
      and a plan for you to implement in real code.

Example 1:
Question: "Which object is closer to the camera? A. Toilet B. Faucet"
Plan:
1) Locate all the Toilet (toilet_bboxes=loc(image, "toilet"))
2) Locate all the Faucet (faucet_bboxes=loc(image, "faucet"))
3) Get 3D pose for the first toilet found (toilet_pose=obj3d(image, toilet_bboxes[0]))
4) Get 3D pose for the first faucet found (faucet_pose=obj3d(image, faucet_bboxes[0]))
5) Extract 3D location of toilet from its 3D pose (toilet_3d_location=toilet_pose['3d_location'])
6) Extract 3D location of faucet from its 3D pose (faucet_3d_location=faucet_pose['3d_location'])
7) Assume the camera is at [0, 0, 0].
8) Compute L2 distances from toilet to camera.
9) Compute L2 distances from faucet to camera.
10) Return A if the distance from toilet to camera is smaller than the distance from faucet to camera,
      otherwise return B.
<program>
# Step 1-2: Locate toilet and faucet
toilet_bboxes = loc(image, 'toilet')
faucet_bboxes = loc(image, 'faucet')
# Step 3: Fallback if either is not detected
if not toilet_bboxes or not faucet_bboxes:
    # Step 4-5: Get 3D pose for first detected toilet and faucet
    toilet_pose = obj3d(image, toilet_bboxes[0])
    faucet_pose = obj3d(image, faucet_bboxes[0])
    # Step 6-7: Extract 3D locations
    toilet_loc = np.array(toilet_pose['3d_location'])
    faucet_loc = np.array(faucet_pose['3d_location'])
    # Step 8-10: Camera is at origin, compute L2 distances
    camera = np.array([0, 0, 0])
    toilet_dist = np.linalg.norm(toilet_loc - camera)
    faucet_dist = np.linalg.norm(faucet_loc - camera)
    # Step 11: Return A if toilet is closer, otherwise B
    final_result = 'A' if toilet_dist < faucet_dist else 'B'
else:
    final_result = "Unable to locate either toilet or faucet."
</program>

Now here is an API of methods. Use the provided plan as a reference to guide your implementation. If you find
       any logical errors, missing steps, or improvements, correct them in your program.
------------------ API ------------------
{predef_signatures}
------------------ API ------------------

Please do not use synonyms, even if they are present in the question.
Using the provided API, output a program inside the tags <program></program> to answer the question.
It is critical that the final answer is stored in a variable called "final_result".
Ensure that the answer is just one option letter.

Here are some helpful definitions:
1) 2D distance/size refers to distance/size in pixel space.
2) 3D distance/size refers to distance/size in the real world. 3D size is equal to 2D size times the depth of
       the object.
3) We define (width, height, depth) as the values along the (x, y, z) axis. Width=x axis, height=y axis,
      depth=z axis.
4) The camera is always located at [0, 0, 0] in 3D space.
5) The 3D up direction is always [0, 1, 0].
6) "Depth" measures distance from the camera in 3D.

Here are some helpful tips:
1) When you need to search over objects satisfying a condition, remember to check all the objects that
      satisfy the condition and don't just return the first one.
2) You already have an initialized variable named "image" - no need to initialize it yourself!
3) When searching for objects to compare to a reference object, make sure to remove the reference object from
       the retrieved objects. You can check if two objects are the same with the same_object method.
4) Some questions may present hypothesis measurements (e.g. "X is 3.5m wide"), these are hypothesis
      measurements and should be used ONLY to scale your outputs accordingly.
5) Do NOT round your answers! Always leave your answers as decimals even when it feels intuitive to round or
      ceiling your answer - do not do it!
6) When a query asks to find all objects in a container just count the number of objects.
7) If anything goes wrong during the program execution, return a fallback value like "unable to locate
      required objects".

Use the plan as a guide but do not blindly follow it. If a step in the plan is logically incorrect, redundant,
       or missing, fix it in your implementation. Add inline comments to explain your reasoning where you
      deviate from the plan. You do not need to define a function - just write your program in the tags.
      Assume "image" has already been initialized - do not modify it!

The question you need to answer is in <question></question> tags and the guiding plan is in <plan></plan>
      tags below.

<question>{question}</question>

<plan>{plan}</plan>
```

Figure 10: **Program Agent Prompt for GeoState on 3DSRBench.**

```
You are an expert logician capable of answering spatial reasoning problems with code. You excel at using a
      predefined API to implement a logical plan as a working program that answers spatial and complex
      reasoning problems.

You will be given a question and a step-by-step solution plan. Your job is to use the provided plan as a
      guiding reference and write a correct program using the provided API. The plan may contain errors or
      imprecisions - treat it as a helpful suggestion rather than a strict specification. If you spot a
      logical error or a better way to achieve the same goal, correct it in your implementation.

I am going to give you some examples of how a question and plan maps to a program, then I will give you an
      API and a plan for you to implement in real code.

Example 1:
Question: "How many objects have the same color as the metal bowl?"
Plan:
1) Set a counter to 0
2) Find all the bowls (loc(image, 'bowls')).
3) If bowls are found, loop through each of the bowls found.
4) For each bowl found, check if the material of this bowl is metal. Store the metal bowl if you find it and
      break from the loop.
5) Find and store the color of the metal bowl.
6) Find all the objects.
7) For each object O, check if O is the same object as the metal bowl (same_object(image, metal_bowl_bbox,
      object_bbox)). If it is, skip it.
8) For each O you don't skip, check if the color of O is the same as the color of the metal bowl.
9) If it is, increment the counter.
10) When you are done looping, return the counter.
Program:
<program>
# Step 1: Set a counter to 0
counter = 0
# Step 2: Find all the bowls
bowls = loc(image, 'bowl')
# Step 3-4: Loop through bowls and find the metal one
metal_bowl_bbox = None
for bowl_bbox in bowls:
    material = vqa(image, 'What is the material of this bowl?', bowl_bbox)
    if 'metal' in material.lower():
        metal_bowl_bbox = bowl_bbox
        break
final_result = 0
if metal_bowl_bbox is not None:
    # Step 5: Find and store the color of the metal bowl
    metal_bowl_color = vqa(image, 'What is the color of this bowl?', metal_bowl_bbox)
    # Step 6: Find all objects
    all_objects = loc(image, 'objects')
    # Step 7-9: Loop through all objects, skip the metal bowl itself, count color matches
    for object_bbox in all_objects:
        if same_object(image, metal_bowl_bbox, object_bbox):
            continue
        object_color = vqa(image, 'What is the color of this object?', object_bbox)
        if metal_bowl_color.lower() in object_color.lower() or object_color.lower() in metal_bowl_color.lower
     ():
            counter += 1
    final_result = counter
</program>

Now here is an API of methods. Use the provided plan as a reference to guide your implementation. If you find
       any logical errors, missing steps, or improvements, correct them in your program.
------------------ API ------------------
{predef_signatures}
------------------ API ------------------

Please do not use synonyms, even if they are present in the question.
Using the provided API, output a program inside the tags <program></program> that is guided by the plan to
      answer the question.
It is critical that the final answer is stored in a variable called "final_result".
Ensure that the answer is either yes/no, one word, or one number.

Here are some helpful definitions:
1) 2D distance/size refers to distance/size in pixel space.
2) 3D distance/size refers to distance/size in the real world. 3D size is equal to 2D size times the depth of
       the object.
3) We define (width, height, length) as the values along the (x, y, z) axis. Width = x axis, height = y axis,
       length = z axis.
4) "Depth" measures distance from the camera in 3D.

Here are some helpful tips:
1) When you need to search over objects satisfying a condition, remember to check all the objects that
      satisfy the condition and don't just return the first one.
2) You already have an initialized variable named "image" - no need to initialize it!
3) When searching for objects to compare to a reference object, make sure to remove the reference object from
       the retrieved objects. You can check if two objects are the same with the same_object method.
4) If two objects have the same 2D width, then the object with the largest depth has the largest 3D width.
5) If two objects have the same 2D height, then the object with the largest depth has the largest 3D height.
6) 2D sizes convey the height and width in IMAGE SPACE. To convert to height and width in 3D space, it needs
      to be multiplied by the depth!
7) Some questions may present hypothesis measurements (e.g. "X is 3.5m wide"), these are hypothesis
      measurements and should be used ONLY to scale your outputs accordingly.
8) Do NOT round your answers! Always leave your answers as decimals even when it feels intuitive to round or
      ceiling your answer - do not do it!
9) When a query asks to find all objects in a container just count the number of objects.
10) The vqa() function may return responses containing extra words beyond the expected answer. Avoid exact
      string matching. Instead, use substring checking (e.g. 'metal' in material.lower()) to robustly handle
      varied responses. For comparisons between two vqa() responses, check in both directions (e.g. color_a.
      lower() in color_b.lower() or color_b.lower() in color_a.lower()) to account for varied phrasings on
      both sides.

Use the plan as a guide but do not blindly follow it. If a step in the plan is logically incorrect, redundant,
       or missing, fix it in your implementation. Add inline comments to explain your reasoning where you
      deviate from the plan. You do not need to define a function - just write your program in the tags.
      Assume "image" has already been initialized - do not modify it!

AGAIN, output a program inside the tags <program></program> that is guided by but not strictly bound to the
      provided plan, and stores the final answer in a variable called "final_result".
It is critical that the final answer is stored in a variable called "final_result".
Ensure that the answer is either yes/no, one word, or one number.

The question you need to answer is wrapped in <question></question> tags and the guiding plan is wrapped in <
      plan></plan> tags below.

<question>{question}</question>

<plan>{plan}</plan>
```

Figure 11: **Program Agent Prompt for GeoState on Omni3D-Bench.**