Benchmarking PhD-Level Coding in 3D Geometric Computer Vision

Supplementary Materials

6. More Statistics of GeoCodeBench

To provide a clearer understanding of the composition and characteristics of GeoCodeBench, we first summarize the data sources and scale of the benchmark. As shown in Table 3, GeoCodeBench collects 47 repositories from CVPR'25, ICCV'25, and ICLR'25, resulting in a total of 100 problem instances, with CVPR'25 contributing the largest portion.

Table 3. Data Source Statistic of GeoCodeBench.

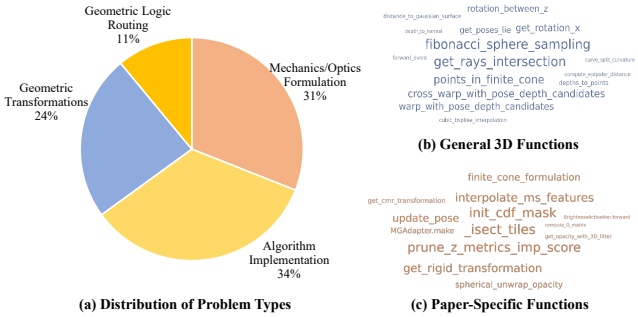
Source	Repos Nums	Problems Nums
CVPR'25	28	55
ICCV'25	15	33
ICLR'25	4	12
Total	47	100

Furthermore, we analyze the token statistics of different components within the benchmark, including the structured paper content, code-with-masked functions, and golden implementation functions (Table 4). The structured paper content is significantly longer than other components, with an average length of 20,232 tokens, whereas the golden implementation functions remain concise, averaging only 753 tokens.

Table 4. Tokens Statistic of Different Components in GeoCodeBench.

Component	Mean	Min	Max
Structured Paper Content	20,232	14,052	31,079
Code w/ Masked Function	3,802	68	14,036
Golden Implementation Function	753	222	2,917

GeoCodeBench also features diverse problem types, as illustrated in Fig. 10(a). Novel algorithm implementations tasks constitute the largest proportion (34%), followed by Mechanics/Optics formulation(31 %), geometric transformations (24%), and Geometric Logic Routing tasks (11%). Representative keywords for general 3D functions and paper-specific research ability functions are visualized in Fig. 10(b)–(c), demonstrating the benchmark's coverage of essential 3D geometry operations, such as spatial transformations, geometric intersections, optimization procedures, and problem-specific functional modules. Collectively, these statistics indicate that GeoCodeBench offers comprehensive diversity in task sources, content structure, and functional requirements, providing a robust foundation for evaluating PhD-level coding capabilities in 3D geometric computer vision.



(a) Distribution of Problem Types (b) General 3D Functions (c) Paper-Specific Functions

Figure 10. Problem Type and Representative Functions in GeoCodeBench.

7. Details of Evaluation

For each problem in GeoCodeBench, we first construct the full prompt following the format shown in Listing 1. The prompt contains (1) the structured paper content, (2) the masked implementation function that the model must rewrite, and (3) the template that defines the I/O and necessary packages.

For every problem instance, we invoke the target LLM exactly once using greedy decoding. The model's response is parsed and inserted into the corresponding code template to reconstruct the full implementation. We then execute the unit tests associated with that problem to determine whether the generated solution is functionally correct. Each problem contributes a unit test PassRate score. After evaluating all problems, we compute the final score for each model by averaging the PassRate values across the entire benchmark.

To ensure fairness across models, we standardize several important evaluation conditions: (1) all models are evaluated with identical prompts and test suites; (2) no system prompts or model-specific engineering tricks are used; (3) for unit test results, we set tolerance and precision thresholds to account for minor numerical variations; (4) for some problems, the reference implementations are not the only correct implementations which derived directly from the original templates. For these problems, we either impose additional constraints on the templates or relax the unit test acceptance criteria.

8. More Case Study

Case Study: Misinterpreting "Distance to a 3D Gaussian Surface" as Mahalanobis Radius This case comes

```
You are an expert in 3D vision. Please read the paper and think about what to fill in the
    ****EMPTY**** section of the code.
Output format: Returns the function according to the template.

## PAPER:
{paper_content_json}

## CODE with EMPTY:
{code}

## TEMPLATE:
{template}

Please fill out the template above.
Your response must strictly follow the format below:

<thinking>
[Your reasoning or thinking content here]
</thinking>

<answering>
[Fill in the complete template here, ensuring that:
 1.All required imports and components are included,
 2.The structure strictly follows the provided template,
 3.The result is a complete, ready-to-use implementation]
</answering>
```

Listing 1. Input Prompt for LLM.

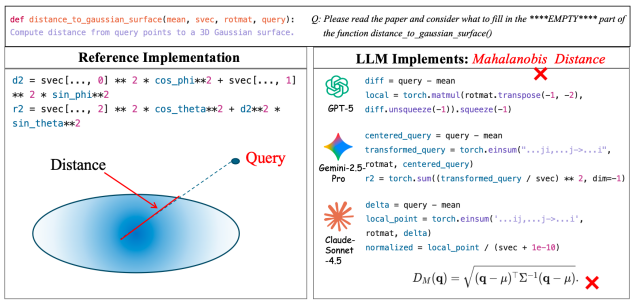


Figure 11. Case Study: Incorrect Type of Distance Implemented.

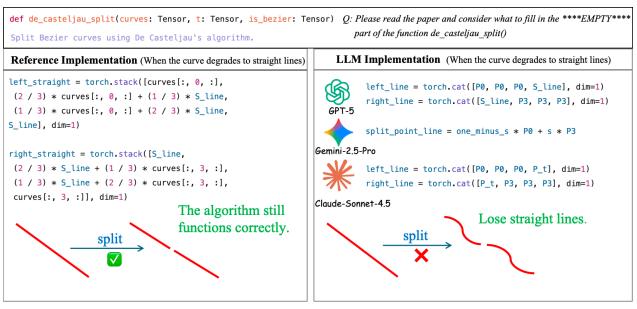


Figure 12. Case Study: Errors of De Casteljau Split When Degrading to Lines.

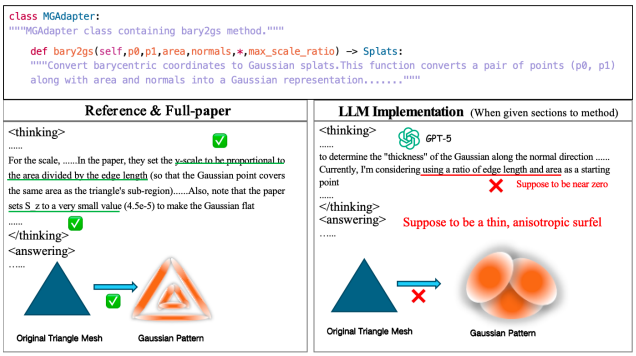


Figure 13. Case Study: MGadapter Fidelity Differs with Text Length.

from[48], which evaluates the ability of LLMs to implement a "distance to a 3D Gaussian surface" function in a three-dimensional geometry and linear algebra setting.

In the reference implementation, the function `distance_to_gaussian_surface` is defined as follows: it returns the distance from the Gaussian center along the ray query − mean to the 1-sigma ellipsoidal surface, i.e., the surface radius of the ellipsoid in that direction. This quantity is independent of the actual radial position of the query point and represents a true physical distance that can be directly used for geometric decisions.

By contrast, the LLM-generated implementation computes the Mahalanobis radius from the point to the center

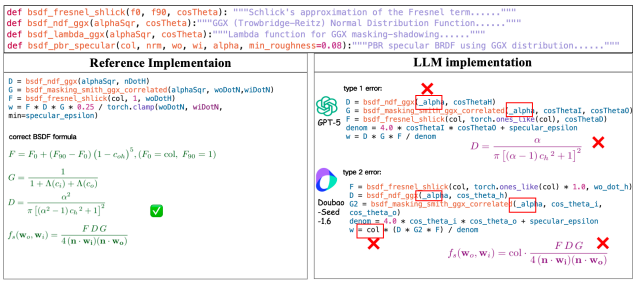


Figure 14. Case Study: Incorrect Implementation of the BRDF Formulas.

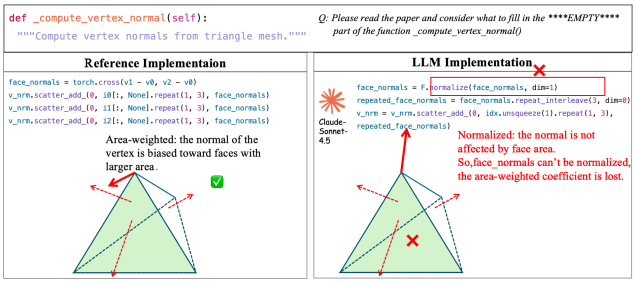


Figure 15. Case Study: Incorrect Normalization that Ignores Area-weighting Coefficients.

of the ellipsoid, which is a normalized, dimensionless distance. This value must be further transformed to fit the logic of the current geometric pipeline. It is more naturally suited to statistical analysis and anomaly detection, rather than to direct geometric distance computation.

From a code quality perspective, the LLM implementation correctly handles translation by the mean, rotation into the local frame, scaling by the axis lengths, and broadcasting across batch dimensions. Mathematically, it is internally consistent and implements a standard and widely used notion of "distance to a Gaussian". However, from the standpoint of the benchmark, it exhibits a critical geometric semantic error: the distance measure it computes does not match the definition used in the reference implementation. As a result, on our designed test cases, the numerical outputs of the two implementations diverge systematically, and the LLM solution is judged incorrect.

Insight This case exposes a typical failure mode of LLMs in 3D geometry–oriented code generation: while they are strong at implementing canonical mathematical constructs, they rely heavily on natural-language specifications and struggle to infer task-specific implicit semantics. Consequently, they tend to default to the most common textbook definition—here, the Mahalanobis distance—rather than the problem-specific "directional surface radius" actually required by the dataset and evaluation. See figure 11.

Key Finding 5

LLMs tend to default to the most common textbook definition rather than the problem-specific implementation actually required by the paper, which reflects the limitation of reasoning ability of a statistical model.

Case Study: Preserving Straight Lines in De Casteljau-Based Bezier Splitting This case comes from [15], designed to evaluate the ability of LLMs to implement the Bezier curve splitting function `de_casteljau_split` in a three-dimensional geometric setting. The function takes as input a batch of curve control points curves $\in \mathbb{R}^{B\times D\times 4}$, a split parameter t, a Boolean mask, and it outputs the left and right sub-curves, each represented in a unified four-control-point form. In the benchmark, the reference implementation and the LLM-generated implementations are essentially equivalent in their De Casteljau recursion for true Bezier curves; however, they exhibit a critical geometric semantic discrepancy in the representation convention for straight-line segments (non-Bezier cases).

In the reference implementation, curves are split using the standard cubic Bezier De Casteljau algorithm, yielding left and right segments. For straight-line segments, the resulting sub-curves are still geometrically straight: all four control points are strictly collinear, and the parameterization remains consistent with the canonical representation of a straight line as a cubic Bezier curve. This implicit constraint is crucial for subsequent geometric processing.

In contrast, the LLM implementation correctly applies the De Casteljau recursion in the Bezier case, but adopts a more "intuitive yet pipeline-incompatible" strategy for straight lines: the two intermediate control points are simply duplicated from the endpoints, violating the system's representational convention. When interpreted as cubic Bezier curves, such control-point configurations cause the curve to "collapse" near the endpoints and deviate from a true straight line.

Insight This case shows that the LLM is capable of reproducing the local De Casteljau splitting pattern, but struggles with preserving global representation invariants and downstream contracts. It defaults to a locally simple control-point layout that is mathematically valid yet semantically incompatible with the canonical line-encoding convention, demonstrating that the model's code generation is driven by local plausibility rather than by a consistent understanding of the project's geometric conventions. See figure 12.

Case Study: Context Length and Geometry–Gaussian Consistency This task evaluates whether LLMs can implement the MGadapter-style mapping from mesh-sampled geometric elements to 3D Gaussian splats, following the design used in GeoSplatting[70]. The target function `bary2gs` takes two points on a triangle edge (p_0, p_1), a per-sample area, and a surface normal, and returns a `Splats` object (means, anisotropic scales, orientation quaternions, colors, and opacities). The correct implementation encodes a very specific geometric intent: each splat should behave as a thin surfel aligned with the local mesh geometry—its long axis aligned with the edge direction in the tangent plane, its short in-plane axis derived from the projected area, and an almost-zero thickness along the normal (log scale fixed at -10) to enforce a consistent geometry–Gaussian correspondence.

When given the **full paper**, the LLM implementation qualitatively captures much of this intent. It correctly places the means at the edge midpoint, constructs an orthonormal frame from a projected edge direction and the normal, and assigns anisotropic scales where the major axis depends on edge length and the minor axis depends on area, with a very small thickness along the normal.

However, in the **to-method** setting, the implementation drifts further toward a generic 3DGS heuristic. The LLM still constructs a local frame from the normal and the edge direction and uses edge length and area to define three scales, but the coupling between these quantities and the axes is looser: some implementations tend to isotropic Gaussian and some set a non-zero value for thickness. The resulting splats are less tightly constrained by the triangle area—making them less faithful to the MGadapter's core design.

Insight This case highlights two effects of context length on LLM code generation. With richer context, the model more accurately reconstructs the intended anisotropic surfel parameterization and uses the right geometric ingredients. With reduced context, it reverts to a more generic ellipsoid construction that is mathematically sound but no longer matches the specialized geometry–Gaussian consistency of the original MGadapter. See figure 13.

Case Study: GGX Roughness Semantics Mismatch This case comes from [87], designed to evaluate the ability of LLMs to implement a physically correct specular BRDF based on the GGX (Trowbridge–Reitz) microfacet model. The reference implementation provides a complete set of BSDF components, including Schlick Fresnel (`bsdf_fresnel_shlick`), the GGX normal distribution function (`bsdf_ndf_ggx`), the GGX masking–shadowing lambda term (`bsdf_lambda_ggx`), Smith's correlated masking function (`bsdf_masking_smith_ggx_correlated`), and the final PBR specular BRDF `bsdf_pbr_specular`. The interface contract is explicit: the first argument of these GGX helper functions is α^2 (the square of the roughness parameter), while the externally exposed `alpha` is squared to obtain `alphaSqr`, which is propagated throughout the GGX computation chain.

At first glance, the LLM-generated implementation appears very similar to the reference code, but it mishandles the semantics of the roughness parameter. The LLM implementation retains the name `alphaSqr`, but passes `_alpha` directly from `bsdf_pbr_specular` without squaring it. As a consequence, the entire GGX computation chain is numerically equivalent to using α in place of α^2, thereby altering the mapping between the user-visible roughness parameter and the underlying GGX slope distribution. This leads to systematically different specular lobe shapes compared to the reference, especially in the low-roughness regime. Although each function looks locally correct in terms of its formulae, the implementation deviates from the ground-truth behavior from the perspective of the API contract.

Insight This case highlights a broader limitation of current LLMs in scientific and graphics code: they are very good at reconstructing familiar mathematical structures and formulations, but much less reliable at tracking subtle parameter semantics and system-level contracts that are changeable in the codebase. In particular, they tend to preserve the "shape" of an algorithm while drifting on the meaning of its parameters, revealing a gap between surface-level code synthesis and a deeper understanding of physical models and engineering interfaces. See figure 14.

Case Study: Loss of Area Weighting in Vertex Normal Computation This case comes from [18], designed to evaluate whether LLMs, when computing vertex normals on triangle meshes, can simultaneously reproduce both the geometric details and the engineering robustness of the reference implementation. The reference implementation first uses the triangle index buffer `t_pos_idx` to fetch the three vertex positions `v0`, `v1`, and `v2` for each face, and then computes the face normals `face_normals` via `torch.cross(v1 - v0, v2 - v0)`. These face normals are *not* normalized; their magnitudes are proportional to the corresponding triangle areas. Consequently, when these face normals are splatted to vertices and accumulated, the method naturally implements an *area-weighted* vertex normal estimation: large triangles contribute more to their incident vertex normals, while small triangles contribute less. Before normalization, the reference implementation explicitly handles degenerate cases: for vertices whose accumulated normal has (almost) zero norm, the normal is replaced by a default direction `[0, 0,`

`1]`, and only then passed through `F.normalize`. In this way, even in the presence of degenerate triangles or "isolated" vertices (vertices not referenced by any face), the output is guaranteed to consist of finite unit normals, satisfying the assumptions made by subsequent rendering and geometry processing stages.

The LLM-generated implementation is structurally close to the reference: it also computes face normals from triangle vertices, accumulates them at vertices, and finally applies `F.normalize` to obtain unit-length normals. However, it deviates from the reference design in two critical aspects. First, the LLM version immediately normalizes the face normals via `face_normals = F.normalize(face_normals, dim=1)`. This forces each triangle to contribute unit-length normals to its incident vertices, thereby discarding the area weighting and making every face contribute equally. On non-uniform meshes (e.g., regions with many small triangles adjacent to fewer large ones), this changes both the smoothing behaviour and the geometric meaning of the vertex normals, and no longer matches the behaviour of the original rendering system. Second, the LLM version does not implement any explicit handling of degeneracies: for vertices whose accumulated normal is zero, `F.normalize` relies solely on an internal `eps` for numerical protection, but does not replace those normals with a prescribed default direction. As a result, the output may still contain zero or near-zero normals, causing the unit tests for this task to fail.

Insight This case clearly illustrates the type of error that the benchmark seeks to capture. The LLM has a solid grasp of the core geometric routine "how to compute vertex normals from a triangle mesh" and can produce code that is structurally correct, shape-compatible, and behaves reasonably on typical inputs. However, it fails to recognize two crucial engineering semantics embodied in the reference implementation: (1) area-weighted smoothing achieved by preserving the magnitudes of face normals, and (2) the explicit injection of a default normal direction for degenerate or isolated vertices to ensure robustness. This gap between formula-level correctness and engineering semantics is precisely what matters in 3D geometry–related CV functions, and it is exactly the capability deficit that this benchmark aims to systematically evaluate. See figure 15 for the first type of error.

Key Finding 6

LLMs can reliably implement the main body of algorithms, but under-handle boundary conditions and corner cases, leading to implementations that appear correct yet fail to pass the full unit-test suite.