3DCODEVERSE: EXPLORING 3D WORLDS AS CODE

Anonymous authors
Paper under double-blind review



Figure 1: 3DCodeVerse's showcases. Assets are built **only by code** (*e.g.*, Blender Python, Three.js, and OpenGL), ranging from 3D objects to articulated objects, scenes, and graphics renderings.

ABSTRACT

We introduce 3DCodeVerse, a research ecosystem that enables coding agents to create editable and verifiable 3D content at scale. 3DCodeVerse comprises large-scale, high-quality datasets of executable 3D code, a coding harness for spatial reasoning and automated verification, and practical guidance for 3D coding. *First*, we construct 3DCodeVerse-2M, a large-scale curated dataset of standalone 3D executable code by collecting open-source data, converting specialized representations, and generating new programs with coding agents. Each program is packaged with the source files and associated assets required for execution in its declared environment, along with available descriptions and verification records. *Second*, we develop the 3DCodeVerse harness, a multi-agent directed acyclic graph framework for generating and refining 3D content. It uses execution checks, spatial measurements, and visual feedback to automatically evaluate and revise generated programs without human feedback during generation. *Third*, we conduct systematic empirical studies of 3D coding, establishing baselines with fine-tuned open models and deriving practical guidance through component-level analyses of our harness. Extensive experiments demonstrate that fine-tuning on our dataset significantly improves the 3D coding capabilities of open-source models. Notably, our 3DCodeVerse harness achieves a 90% win rate against Claude Design in pairwise human evaluations when both are utilized by Claude Opus 5.5. [1]

1 INTRODUCTION

> *The description of a drawing stored in the Sketchpad system is more than a collection of static drawing parts.*
>
> *— Ivan E. Sutherland, Sketchpad (1963)*

[1]The datasets, harness, and models will be publicly released to advance research on 3D coding.

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.

Table 1: **Comparison of 3D coding datasets and harnesses.** Format counts refer to code representations. Sizes use source-specific units and are not directly comparable.

Work	Dataset					Reported size		Harness		
	Static	Artic.	Scene	Graphics	Code Types	Count	Unit	Iterative refinement	Visual feedback	Geometry checks
BlenderLLM (Du et al., 2024)	✓	✗	✗	✗	1	8 K	Pairs	✗	✗	✗
3DCodeBench (Gao et al., 2026)	✓	✗	✗	✗	1	26 K	Scripts	✓	✓	✓
Articraft (Zhou et al., 2026)	✗	✓	✗	✗	2	10 K	Assets	✓	✗	✓
VIGA (Yin et al., 2026)	—	—	—	—	—	—	—	✓	✓	✗
3DCodeVerse (Ours)	✓	✓	✓	✓	**6**	**2.03 M**	Pairs	✓	✓	✓

3D coding offers a promising approach to 3D creation by using executable programs to construct 3D geometry, define articulated motion, compose scenes, and generate graphics effects (Raistrick et al., 2023; Hu et al., 2024; Zhou et al., 2026). Unlike typical 2D generation workflows, code-based 3D generation produces structured, executable content that can be edited, reused, composed, and inspected. These properties support scalable and controllable 3D modeling for diverse applications. Recent advances in language models and coding agents further automate this process, allowing users to create complex 3D content through text instructions (Sun et al., 2023; Yin et al., 2026).

Despite these advances, shared infrastructure for developing and evaluating 3D coding systems across environments remains limited. Existing resources address complementary aspects of 3D coding: DeepCAD and BlenderLLM provide supervision for CAD construction and Blender scripting (Wu et al., 2021; Du et al., 2024); SceneCraft and VIGA refine generated content using visual feedback (Hu et al., 2024; Yin et al., 2026); and Articraft combines a specialized SDK with a harness for articulated asset generation (Zhou et al., 2026). However, these resources remain fragmented across tasks and environments. As shown in Table 1, existing datasets primarily focus on static or articulated objects, while harnesses vary in their support for visual assessment and automated corpus construction.

To bridge these gaps, we introduce 3DCodeVerse (Figure 1), an open-source foundational framework for learning and scaling 3D coding through a large-scale executable code corpus and a 3D code generation harness. We first build 3DCodeVerse-2M, a unified corpus of executable 3D programs spanning static objects, articulated assets, scenes, and graphics. Rather than simply aggregating existing datasets, we systematically convert heterogeneous 3D representations into reusable source code and remove dependencies on specialized or unavailable libraries. For example, we translate DeepCAD construction sequences into CadQuery programs and inline Articraft geometry routines while preserving URDF articulation specifications. Each program is paired with its associated assets, textual descriptions, provenance, and available verification signals.

We further develop 3DCodeVerse Harness, a 3D code generation harness for synthesizing and refining complex 3D programs from text, images, or videos. The harness decomposes multi-component projects into a directed acyclic graph (DAG) of coding and assembly tasks, allowing multiple agents to develop and integrate source files in a structured manner. Generated programs are executed and rendered, after which an independent judge agent evaluates the outputs against explicit acceptance criteria. Complementary spatial checks detect structural inconsistencies and prevent visually plausible but geometrically invalid solutions from receiving high scores. The resulting feedback is then converted into localized repair objectives, enabling repeated execution–evaluation–revision cycles without human intervention. Through this iterative process, the harness supports *inference-time scaling* to complex projects containing thousands of lines of code.

Importantly, 3DCodeVerse Harness serves not only as an inference system but also as a data-generation engine. By recording successful programs, failed attempts, targeted repairs, and preference comparisons, its execution traces provide supervision for both supervised fine-tuning and preference learning. Together, 3DCodeVerse-2M and 3DCodeVerse Harness establish not only a reusable infrastructure for 3D coding, but also a closed loop connecting data construction, program synthesis, verification, refinement, and model improvement. We further support this ecosystem with open models and reproducible training recipes. Fine-tuning on 3DCodeVerse-2M substantially improves executable 3D code generation, reaching execution rates of 84.9% with a 9B model and 92.9% with a 27B model on 3DCodeBench. Moreover, 3DCodeVerse Harness consistently improves generation quality across coding agents, yielding gains of 15.5–29.1 points over one-shot generation.

2 Related Work

Datasets and Benchmarks for 3D Coding. Learning to generate 3D content as code benefits from supervision over its construction procedures. DeepCAD and Fusion 360 Gallery expose CAD construction sequences, and Text2CAD connects such sequences to natural language (Wu et al., 2021; Willis et al., 2021; Khan et al., 2024). This supervision has expanded to executable CadQuery and Blender programs for reconstruction and language-conditioned generation through BlendNet, CAD-Recode, CAD-Coder, and MeshCoder (Du et al., 2024; Rukhovich et al., 2025; Guan et al., 2026; Dai et al., 2026). Complementary procedural resources broaden the content represented by code: Infinigen and its extensions generate assets and scenes (Raistrick et al., 2023; 2024; Joshi et al., 2025), while Shaders21k provides programs for procedural image synthesis (Baradad et al., 2022). Benchmarks such as CADPrompt, BlenderGym, and 3DCodeBench assess the resulting code generation and editing capabilities (Alrashedy et al., 2025; Gu et al., 2025; Gao et al., 2026). However, our paper focuses on scaling executable supervision across these heterogeneous coding environments. Our 3DCodeVerse-2M provides a million-scale training corpus spanning CAD, Blender, shaders, and browser graphics, with programs that are self-contained within their respective target runtimes. Its construction combines curated existing sources with automated program generation and execution-based filtering, enabling continued data expansion for training open 3D coding models.

Harness Engineering for 3D Coding. Execution and feedback allow 3D coding agents to inspect and revise the content produced by their programs. Systems such as 3D-GPT, SceneCraft, and LL3M establish language-driven planning and program generation for 3D modeling (Sun et al., 2023; Hu et al., 2024; Lu et al., 2025). Executing and rendering these programs enables visual refinement, as explored by BlenderAlchemy, CADCodeVerify, and VIGA (Huang et al., 2024; Alrashedy et al., 2025; Yin et al., 2026). Beyond rendered appearance, Articraft provides structured asset validation through a specialized SDK, while Bioinspired123D connects script quality control with a graph-based agentic workflow (Luu & Buehler, 2026b; Zhou et al., 2026). 3DHarnessBench further studies how the observation and tool interfaces available to an agent affect 3D reconstruction (Liu et al., 2026). In contrast, our 3DCodeVerse harness provides a reusable execution and verification framework across coding backends, jointly supporting inference-time refinement and scalable training data curation. Its multi-agent directed acyclic graph coordinates execution, rendering, spatial measurements, and structural checks across static and articulated objects, scenes, and visual effects. The same framework equips coding agents with visual and programmatic feedback and serves as an automated data engine that generates, refines, and filters training programs without human feedback in the curation loop.

3 3DCodeVerse-2M: A large-scale 3D coding dataset

Task Definition of 3D Coding. We define *3D coding* as writing a program f that constructs 3D content satisfying a condition c, comprising a text request and optional reference images. The program is standalone, bundling the source files and assets needed to run in a declared environment e such as Blender (`bpy`) or CadQuery. Analogous to compiling source code, executing f in e yields

$$(a, \ell) = \mathcal{E}_e(f), \qquad x = \mathbb{I}[a \neq \perp], \tag{1}$$

where a is the artifact (a mesh, an articulated asset, a scene, or an animation), ℓ is the execution feedback, and $\perp$ denotes failure. Unlike conventional code generation, success requires not only executability ($x = 1$) but also high-quality 3D content. Quality is measured by a task-specific metric $\mathcal{D}(a, a^\star)$ when a reference $a^\star$ exists, and otherwise against c using programmatic measurements and renders $o = \mathcal{R}(a)$.

3D Coding with LLMs and Coding Agents. A generation policy π produces the program. A single-shot LLM writes $f^{(0)} = \pi(c; e)$ directly. A coding agent instead iterates like a developer in a compile–debug loop, revising its code based on execution feedback and renders:

$$f^{(t)} = \pi\big(c, \{f^{(k)}, \ell^{(k)}, o^{(k)}\}_{k<t}; e\big), \tag{2}$$

where $(a^{(t)}, \ell^{(t)}) = \mathcal{E}_e(f^{(t)})$ and $o^{(t)} = \mathcal{R}(a^{(t)})$. After T refinement rounds, the agent returns $f^{(T)}$, so the single-shot setting is the case $T = 0$. The harness implements $\mathcal{E}_e$ and $\mathcal{R}$: agents call them as tools, and all final programs are evaluated with them under the same protocol.

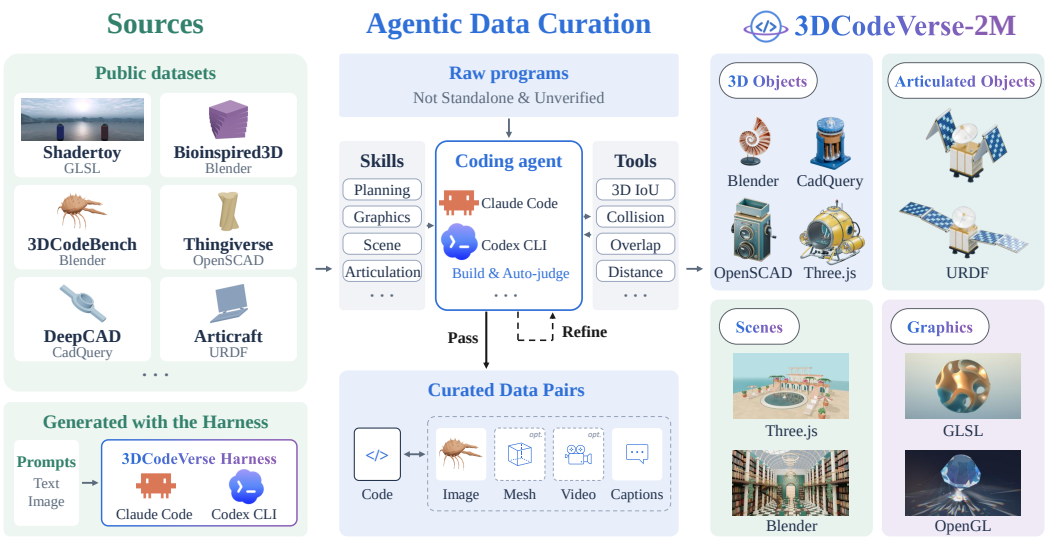

Figure 2: **Curation and composition of 3DCodeVerse-2M.** Left: We collect programs from public datasets and generate more with our harness. Middle: Our 3D coding harness, equipped with modeling skills and inspection tools (e.g., 3D IoU, collision checks), builds each program, judges the output, and refines it until it passes. Each verified program is paired with rendered images and captions, plus meshes and videos when available. Right: The resulting corpus covers 3D objects, articulated objects, scenes, and graphics across multiple programming languages and engines.

Data Sources and Composition. Figure 2 shows representative sources; Appendix B.2 summarizes the full collection. Public resources include Bioinspired3D (Luu & Buehler, 2026a), 3DCodeBench (Gao et al., 2026), Thingiverse, Shadertoy (Quilez & Jeremias, 2013), CodePen animations (Animation2Code), and multi-file Three.js/React Three Fiber projects. We convert DeepCAD sequences (Wu et al., 2021) and Articraft geometry (Zhou et al., 2026) to CadQuery, retaining Articraft's URDF. The ProcTHOR (Deitke et al., 2022) subset pairs scene specifications with Blender/Three.js reconstructions. Blender and Three.js coding-agent subsets contribute generated objects and scenes; We further synthesize and archive new programs by running 3DCodeVerse Harness on text and image prompts (§4).

Agentic Data Curation and Toolkits. Coding agents curate programs with reusable *skills* (Appendix A.10) that package authoring conventions, code examples, and repair guidance. Given a raw program f_{raw} that relies on specialized APIs, an agent produces a standalone program $f = \mathcal{N}_e(f_{\text{raw}})$ for environment e by translating those APIs, inlining required routines, and bundling sources and assets; DeepCAD is instead converted by a deterministic script (Appendix B.2).

Executing f yields $(a, \ell) = \mathcal{E}_e(f)$ (Eq. 1), and inspection *tools* return measurements $\mathcal{M}_e(a)$ (Table 5): part inventories and bounding boxes flag missing or misplaced components, contact and collision checks flag gaps and interpenetration, including across sampled joint poses, and placement and frame probes check physical support, invalid renders, and temporal defects. Findings are reported per part in the authoring frame (Appendix A.4), and execution errors or failed checks return the program to the agent for targeted repair. Conversion fidelity is measured by voxel IoU for DeepCAD and Chamfer distance for Articraft. Coverage thus varies by source: collected and converted programs are checked by execution and rendering where a runtime is available, whereas harness-generated programs also carry gate and judge results as quality tiers (Appendix A.11).

A VLM captions programs from code and renders $o = \mathcal{R}_e(a)$ with three conditions $C = \{c^{\text{inst}}, c^{\text{det}}, c^{\text{proc}}\}$: an instruction, a visual description, and a construction procedure, each usable as c in Eq. 1 (Appendix B.4). Each record $s = (f, e, C, o, m)$ stores provenance, license, and available verification results in m, keeping execution, geometric, and visual checks separate; source files, pre-conversion programs, and URDF files are retained, and meshes and videos are linked when available. Code, geometry, and render fingerprints flag near-duplicates (Appendix B.3), and 3DCodeBench factories, instances, and test splits are reserved for evaluation (Appendix C.1).

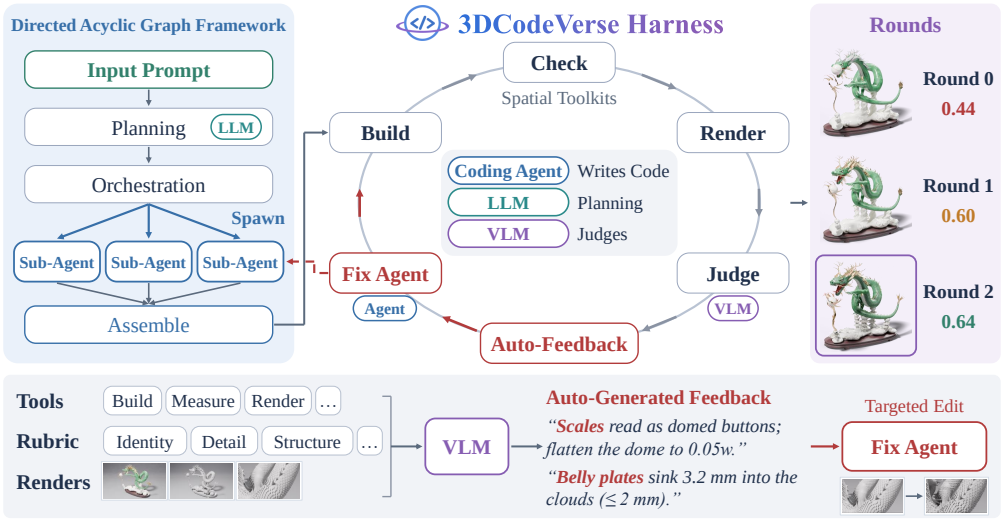

Figure 3: **Overview of the 3DCodeVerse Harness.** Given an input prompt, an LLM planner defines the target asset, and a DAG coordinates sub-agents to code and assemble a 3D program (left). Execution checks, spatial measurements, and rubric-guided VLM judgments provide automated feedback (center) for targeted code revisions by a fix agent (bottom). The right panel illustrates successive refinement rounds, with the highest-scoring program selected as the final output.

4 3DCodeVerse Harness: A Data Engine for 3D Coding

Generating high-quality 3D programs often requires iterative refinement, but reliance on human feedback makes this process difficult to scale. To address this bottleneck, we develop 3DCodeVerse Harness, a multi-agent framework that automates 3D code generation and refinement for data curation and agent-driven content creation (Figure 3). It coordinates coding and assembly through a DAG and combines execution checks, spatial tools, and rubric-guided VLM judgments to identify defects and guide targeted revisions without human feedback during generation. Because the reliability of this feedback depends on how its prompts, tools, and rubrics are configured, we further refine the judge and generation workflow through an alternating development-time adversarial loop.

Multi-Agent Directed Acyclic Graph (DAG) Framework. Given a condition c and a target execution environment e, an LLM planner first produces $P = \mathcal{A}_{\text{plan}}(c; e)$, specifying the target components, dimensions, materials, spatial relations, and acceptance criteria. The harness translates P into a DAG $\mathcal{G} = (V, E)$ of coding and assembly tasks, with edges encoding prerequisites. Within this graph, explicit file ownership allows independent coding tasks to be carried out concurrently by sub-agents. Once their prerequisites are satisfied, assembly tasks integrate the resulting code, ultimately producing the initial program $f^{(0)}$ for subsequent evaluation and feedback-driven refinement.

Automated Feedback and Iterative Refinement. At each round t, the harness executes the current program $f^{(t)}$ and collects three types of feedback:

- **Execution feedback.** Build and runtime logs $\ell^{(t)}$. If execution fails, no artifact is produced and this is the only feedback.
- **Measured feedback.** Measurement tools $\mathcal{M}_e$ check the artifact $a^{(t)}$ for dimensions, contacts, collisions, kinematics, and placement, and return component-level violations with numerical correction hints.
- **Visual feedback.** A VLM judge inspects multi-view renders $\mathcal{R}_e(a^{(t)})$ against the input c and the acceptance criteria in P, producing defect evidence, criterion scores, and proposed revisions: $j^{(t)} = \mathcal{A}_{\text{judge}}(c, P, o^{(t)})$, where $o^{(t)} = (\ell^{(t)}, \mathcal{M}_e(a^{(t)}), \mathcal{R}_e(a^{(t)}))$.

The score $s^{(t)}$ combines the weighted criteria and defect penalties in $j^{(t)}$, measured violations and unmet requirements cap it, and measurements veto visual defect claims they rule out. All feedback is

converted into revision tasks $\Delta^{(t)}$, prioritized in the order listed above, and a fix agent applies them: $f^{(t+1)} = \mathcal{A}_{\text{code}}(c, P, f^{(t)}, \Delta^{(t)}; e)$.

Adversarial Loop Engineering. The preceding loop uses automated feedback in place of per-round human inspection and revision guidance. Because the reliability of this feedback depends on how prompts, tools, and rubrics are configured, we alternate between refining the judge and the generation workflow during development. Let θ_J denote the judge's evidence presentation, rubric, response schema, and aggregation rules, and θ_H the harness's prompts, contracts, skills, tools, and refinement policy. *Judge refinement* first fixes the generation workflow and repeatedly evaluates a bank $\mathcal{B}$ of stored artifacts. Using the same artifacts separates changes in judge behavior from variability in generation. We revise view selection, evidence presentation, and defect checklists, and score aggregation to reduce judgment variability and contradictions with measured evidence. The judge is required to report observations before scores and does not receive the generator's reasoning. We summarize the consistency objective as:

$$\mathcal{L}_J(\theta_J) = \mathbb{E}_{(a,o)\sim\mathcal{B}}\Big[\mathrm{Var}_r\ s^{(r)}(o;\theta_J) + \lambda\,\rho\big(j(o;\theta_J), \mathcal{M}_e(a)\big)\Big], \tag{3}$$

where r indexes repeated judgments and ρ counts claims contradicted by applicable measurements. Preserving sensitivity to differences in artifact quality remains an additional requirement beyond this consistency objective. With the revised judge fixed, *harness refinement* uses its feedback to guide changes to the generation workflow. Candidate designs are compared on matched prompts, balancing judged output quality against observable workflow failures:

$$\mathcal{U}_H(\theta_H;\theta_J) = \mathbb{E}_c\Big[s\big(\hat{f}(c;\theta_H);\theta_J\big)\Big] - \mu\,\mathcal{L}_{\text{fail}}(\theta_H), \tag{4}$$

where $s(\hat{f};\theta_J)$ denotes the score obtained by executing and judging the selected program $\hat{f}$, and $\mathcal{L}_{\text{fail}}$ summarizes failures such as invalid plans, retried tool calls, and failed builds. The weights $\lambda, \mu \geq 0$ express the respective trade-offs.

5 Experiments

5.1 Results of 3DCodeVerse Harness

Evaluation setup. We compare 3DCodeVerse against models with their native harnesses and Claude Design on 100 text-to-3D object prompts. We further compare against native harnesses on articulated-object generation, 50 scene prompts, and 50 graphics prompts. We evaluate generation quality using three metrics. *Human Preference* is the fraction of anonymized side-by-side comparisons won according to human evaluators, with ties split equally. *VLM Preference* is the fraction of the same comparisons won according to a VLM judge (Gemini 3.1 Pro), with ties split equally. *Judge Score* is Gemini 3.1 Pro's overall rubric score, combining weighted criteria with defect penalties and score caps (0–100; failed generations receive a score of zero). Following 3DCodeBench (Gao et al., 2026), we additionally report SigLIP-2 (Tschannen et al., 2025) text–image cosine similarity averaged over four views. The evaluation judge is described in Appendix C.4, and further evaluation details in Appendix C.3.

Main Results. Figure 4 shows a consistent pattern across backbones and tasks: outputs from the native harness are rarely preferred, while the same backbone within 3DCodeVerse Harness wins most comparisons. On 3D objects, *Human Preference* for 3DCodeVerse Harness ranges from 69% to 89%, while *Judge Score* improves by 11–25 points. The largest gains occur for the weaker models, including Gemini 3.8 Flash (61 to 87) and GPT-6 Luna (64 to 86). Against Claude Design, which uses Claude Opus 5.5, 3DCodeVerse Harness with the same backbone is preferred in 90% of human comparisons and achieves a Judge Score of 88 versus 37. The same pattern extends to articulated objects, scenes, and graphics, where Human Preference reaches 78–88%. The agreement between human preference and rubric-based scores suggests that the gains are not specific to the VLM rubric used during refinement.

5.2 Fine-tuning Open Models on 3DCodeVerse-2M

Setup. We fine-tune Qwen3.5-9B and Qwen3.8-27B with LoRA (Hu et al., 2021) in LLaMA-Factory (Zheng et al., 2024) on mixtures drawn from 3DCodeVerse-2M (Appendix C.1) and compare

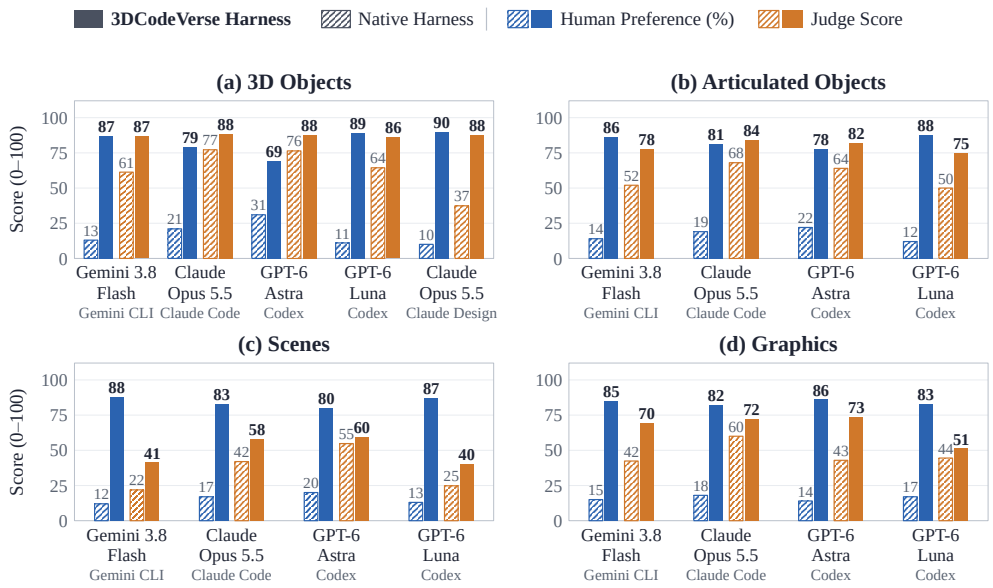

Figure 4: **Main results of 3DCodeVerse Harness across four 3D coding tasks.** Panels show (a) 3D objects, (b) articulated objects, (c) scenes, and (d) graphics. Blue bars report Human Preference (%), and orange bars report Judge Score (0–100; failed generations receive a score of zero). Hatched bars denote each backbone's native harness, while solid bars denote 3DCodeVerse Harness. The final group in (a) compares Claude Design with 3DCodeVerse Harness, both using Claude Opus 5.5.

Table 2: **Fine-tuning with 3DCodeVerse-2M makes open models executable.** Execution rate (%) before and after LoRA fine-tuning (training-set size in parentheses); Δ is the change and the better value of each pair is in bold. A program counts when it runs in the dialect's runtime and produces a result (GLSL: the shader compiles). All models are decoded at T=0.7 with one seed on the same fixed test sets.

Model	3DCodeBench (212)	Blender (103)	CadQuery (200)	OpenSCAD (50)	GLSL (200)	Three.js (40)
Qwen3.5-9B	0.0	1.9	10.5	50.0	33.0	72.5
+ SFT	**84.9**	**90.3**	**98.5**	**86.0**	**56.0**	**74.5**
Δ	+84.9	+88.4	+88.0	+36.0	+23.0	+2.0
Qwen3.8-27B	51.9	74.8	65.5	80.0	57.0	72.5
+ SFT + DPO	**92.9**	**93.2**	**98.5**	**90.0**	**71.5**	**92.5**
Δ	+41.0	+18.4	+33.0	+10.0	+14.5	+20.0

the resulting models with their respective pretrained checkpoints. Evaluation spans 805 fixed tasks across six suites: 212 text-to-Blender tasks from 3DCodeBench (Gao et al., 2026) and five test splits drawn from 3DCodeVerse-2M: Blender (103 procedural objects), CadQuery (200 mechanical parts from DeepCAD and Articraft), OpenSCAD (50 Thingiverse models), GLSL (200 Shadertoy shaders), and Three.js (40 single-object scenes). Each task pairs a natural-language request with a reference program. Executing the reference program produces the evaluation target: a mesh for the four 3D suites, a rendered frame for GLSL, and a page screenshot for Three.js. All models receive the same request and per-dialect system prompt without examples, are decoded at T=0.7 with one seed, and have every output executed in the dialect's pinned runtime (Appendix C.3). We report the execution rate (Table 2) and fidelity to the target (Table 3): F@0.05 between the generated and reference meshes, and DINOv2-L similarity between clay renders of the two meshes from four fixed views; failed programs score zero on both.

Table 3: **Fidelity to the reference before and after fine-tuning.** Same models, decoding, and layout as Table 2. *F*: F@0.05 between generated and reference meshes. *DINO*: DINOv2-L similarity between four-view renders of generated and reference meshes.

	3DCodeBench		F@0.05 per dialect		
Model	F	DINO	Blender	CadQuery	OpenSCAD
Qwen3.5-9B	.000	.000	.008	.030	.261
+ SFT	**.317**	**.266**	**.693**	**.308**	**.329**
Δ	+.317	+.266	+.685	+.279	+.069
Qwen3.8-27B	.214	.211	.439	.159	**.415**
+ SFT + DPO	**.374**	**.298**	**.798**	**.234**	.416
Δ	+.160	+.088	+.359	+.075	+.001

Fine-tuning substantially improves executability. Before fine-tuning, none of the programs generated by Qwen3.5-9B on 3DCodeBench execute successfully; most fail immediately by calling scene-clearing operators unavailable in Blender 5. After fine-tuning on 3DCodeVerse-2M, execution rises to **84.9%** on 3DCodeBench, 90.3% on Blender, and 98.5% on CadQuery, with additional gains of 36.0 and 23.0 points on OpenSCAD and GLSL (Table 2). Qwen3.8-27B starts from a stronger baseline but shows the same trend: fine-tuning raises execution on 3DCodeBench from 51.9% to **92.9%** (+41.0 points), on Blender to 93.2% (+18.4), and on CadQuery to 98.5% (+33.0).

5.3 Qualitative Analysis

For matched prompts and backbones, Figure 5(a) shows that 3DCodeVerse Harness produces more complete structures and finer details, such as the phoenix's legs and nest and the piano's interior. Figure 5(b) shows how successive refinement rounds correct specific defects, improving the phoenix's facial feathers, attaching the stool legs to the seat, and reducing block artifacts in the neon shader. Together, these examples show that automated feedback can correct both structural and appearance defects without additional human instructions.



Figure 5: **Qualitative comparisons and automated refinement. (a)** Outputs for identical prompts and backbones using the native harness (*Without*) versus 3DCodeVerse Harness (*With*). **(b)** Selected rounds of automated refinement for objects and graphics. Numbers report rubric scores on a $[0, 1]$ scale and are comparable within each pair or refinement sequence.

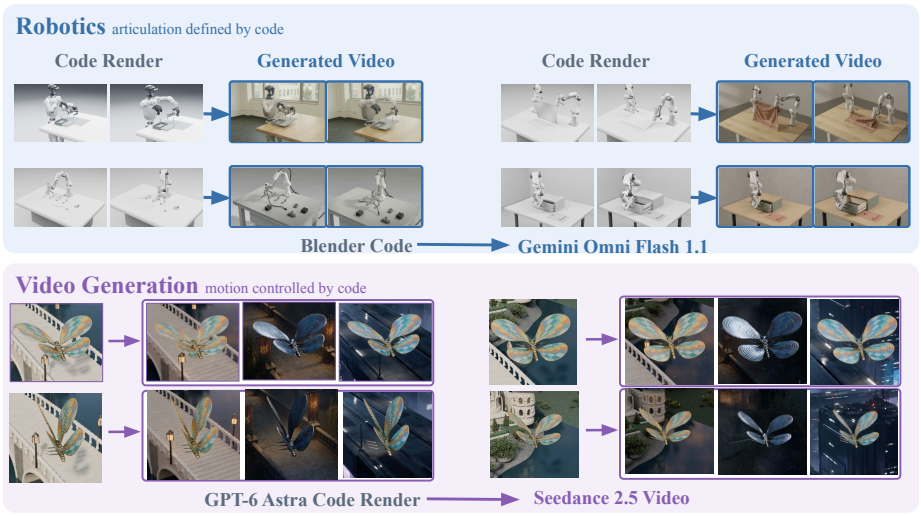

Figure 6: **Applications of 3D code.** Code renders from 3DCodeVerse Harness guide video generation with Gemini Omni Flash 1.1 for robot manipulation scenes (top) and Seedance 2.5 for a GPT-6 Astra scene (bottom).

6 Applications of 3D coding

Robotics. 3D coding enables the construction of articulated assets and scenes with explicit geometry and physical constraints for simulated robot tasks. Programs encode rigid-link geometry, joint axes, and joint limits as editable parameters, supporting programmatic checks and controlled variation of object dimensions, articulation, and initial poses. These task instances can be executed in simulations to generate robot interactions and observations. To provide a preliminary evaluation, we import an articulated asset validated and exported with 3DCodeVerse tools into MuJoCo and compare rendered observations with frozen UWM (Zhu et al., 2025) predictions across 12 cases spanning two assets, two contact states, and three control sequences, using complete scene state for rendering and image history with matched robot commands for UWM. Code renders and their UWM VAE reconstructions each pass a fixed door-and-handle geometry check in 12/12 outputs, versus 7/36 UWM predictions. These results show that 3D code can help generate articulated assets and scenes for robotics simulation with explicit geometry and physical constraints.

Video generation. 3D coding provides an editable interface for controlling the spatial and temporal conditions of video generation. Programs specify scene layout, object articulation and motion, and camera trajectories. We render generated 3D scenes using programs and harness them as reference videos, using the full sequences to condition Gemini Omni Flash (Google DeepMind, 2026) and Seedance 2.5 (ByteDance Seed, 2026). The rendered videos supply spatial and temporal guidance, while the video models are used to refine visual appearance. We present paired reference and generated frames to examine how the specified layout, object motion, and camera trajectories are preserved. This application connects explicit scene programs with video synthesis.

7 Conclusion

We present 3DCodeVerse, an open ecosystem that provides a shared foundation for 3D coding through datasets, a reusable harness, and open model baselines. The dataset brings together programs across coding environments with descriptions, associated assets, and available quality records. The harness combines execution, spatial checks, and visual judgment to support iterative generation without human feedback during a run, while retaining the resulting programs and revisions for data curation. Experiments demonstrate the utility of the data for training open models and the benefits of the harness for existing coding agents. We will release the datasets, harness, models, and training recipes to support further research on 3D generation through code.

AI USE STATEMENT

Generative AI is not merely a writing aid in this work: it is also an instrument of data production and a subject of evaluation, with its use recorded at the artifact level. *As a data-production instrument:* coding agents convert non-code assets into runnable programs (agent-converted samples), generate programs from prompt specifications (agent-distilled samples), and produce the three caption styles. The corresponding pipelines record available execution and review evidence, while harness exports record data provenance and model roles through dedicated source, generator, planner, and judge fields. *As a subject of evaluation:* our harness and fine-tuning experiments directly evaluate these systems. *As a writing and engineering aid:* this manuscript was drafted with AI assistance using the project's repositories, run logs, and internal notes, and portions of the harness and training code were generated by coding agents and reviewed by the authors. Schematic figures were also prepared with AI assistance using editable diagram elements, while the dataset previews in Figure 2 are derived from source artifacts and their renders. Experimental results are reported with their recorded configurations, and historical development studies are identified in the Appendix. We take responsibility for the final content of this work, including all text, claims, and artifacts.

ETHICS STATEMENT

This work releases a large corpus of executable code together with the renders produced by that code. We consider three associated concerns and address each through the released artifacts as well as in this statement.

Licensing: each sample records its original license alongside the dataset-level CC-BY-NC-SA-4.0 terms, with the stricter terms governing reuse. Samples with unresolved license information are explicitly marked in the metadata. In particular, 17.3% of the OpenSCAD subset carries a non-commercial clause, enabling downstream users to identify and filter these samples when commercial reuse is required.

Execution safety: because the corpus contains executable programs, running third-party samples may introduce execution risks. The release records declared execution environments and available check results, although coverage varies across collections. We recommend executing third-party programs in an isolated environment.

Content: the corpus contains no scans or reconstructions of identifiable individuals, and personal or identifying content is excluded by construction. One unresolved failure mode is that generated objects may occasionally contain invented real-world brand marks.

REPRODUCIBILITY STATEMENT

The corpus, harness, training recipes, and evaluation stack are all released. §3 and Appendices B and B.4 document the corpus composition, source files and associated assets, and the complete on-disk contract, including byte-range locators that make individual samples independently retrievable and verifiable without downloading an entire shard. Appendix C.1 specifies all training hyperparameters, the exact composition of each training mixture, the construction rules for both preference sets, and the training partitions. Appendix C.3 specifies the pinned execution environments, code-extraction procedure, geometry metrics and normalization, and evaluation protocol. It also reports variation across repeated harness runs and discusses why small development sets cannot reliably resolve modest component-level effects. Finally, current harness records store prompt and rubric hashes, role and budget settings, harness versions, and detected runtime versions; older records may contain less complete metadata.

REFERENCES

Kamel Alrashedy, Pradyumna Tambwekar, Zulfiqar Haider Zaidi, Megan Langwasser, Wei Xu, and Matthew C. Gombolay. Generating CAD code with vision-language models for 3D designs. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=BLWaTeucYX.

Manel Baradad, Richard Chen, Jonas Wulff, Tongzhou Wang, Rogerio Feris, Antonio Torralba, and Phillip Isola. Procedural image programs for representation learning. *Advances in Neural Information Processing Systems*, 35:6450–6462, 2022.

ByteDance Seed. One-take creation, flexible referencing: Introducing seedance 2.5, 2026. URL https://seed.bytedance.com/en/blog/one-take-creation-flexible-referencing-introducing-seedance-2-5. Official model announcement, July 31, 2026.

BingQuan Dai, Luo Li, Qihong Tang, Jie Wang, Xinyu Lian, Hao Xu, Minghan Qin, Xudong Xu, Bo Dai, Haoqian Wang, Zhaoyang Lyu, and Jiangmiao Pang. Meshcoder: Llm-powered structured mesh code generation from point clouds. *Advances in Neural Information Processing Systems*, 38: 8917–8954, 2026.

Matt Deitke, Eli VanderBilt, Alvaro Herrasti, Luca Weihs, Kiana Ehsani, Jordi Salvador, Winson Han, Eric Kolve, Aniruddha Kembhavi, and Roozbeh Mottaghi. ProcTHOR: large-scale embodied AI using procedural generation. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, volume 35, pp. 5982–5994. Curran Associates, Inc., 2022. doi: 10.52202/068431-0433. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/27c546ab1e4f1d7d638e6a8dfbad9a07-Abstract-Conference.html.

Yuhao Du, Shunian Chen, Wenbo Zan, Peizhao Li, Mingxuan Wang, Dingjie Song, Bo Li, Yan Hu, and Benyou Wang. BlenderLLM: training large language models for computer-aided design with self-improvement. *arXiv preprint arXiv:2412.14203*, 2024. URL https://arxiv.org/abs/2412.14203.

Yipeng Gao, Lei Shu, Genzhi Ye, Xi Xiong, Ameesh Makadia, Meiqi Guo, Laurent Itti, and Jindong Chen. 3DCodeBench: benchmarking agentic procedural 3D modeling via code. *arXiv preprint arXiv:2606.01057*, 2026. doi: 10.48550/arXiv.2606.01057. URL https://arxiv.org/abs/2606.01057.

Google DeepMind. Gemini omni, 2026. URL https://deepmind.google/models/gemini-omni/. Official model documentation. Accessed September 25, 2026.

Yunqi Gu, Ian Huang, Jihyeon Je, Guandao Yang, and Leonidas Guibas. Blendergym: Benchmarking foundational model systems for graphics editing. In *2025 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 18574–18583. IEEE, 2025.

Yandong Guan, Xilin Wang, XiMing Xing, Jing Zhang, Dong Xu, and Qian Yu. Cad-coder: Text-to-cad generation with chain-of-thought and geometric reward. *Advances in Neural Information Processing Systems*, 38:59765–59789, 2026.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021. URL https://arxiv.org/abs/2106.09685.

Ziniu Hu, Ahmet Iscen, Aashi Jain, Thomas Kipf, Yisong Yue, David A. Ross, Cordelia Schmid, and Alireza Fathi. SceneCraft: an LLM agent for synthesizing 3D scenes as blender code. In Ruslan Salakhutdinov, Zico Kolter, Katherine A. Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, volume 235 of *Proceedings of Machine Learning Research*, pp. 19252–19282. PMLR / OpenReview.net, 2024. URL https://proceedings.mlr.press/v235/hu24g.html.

Ian Huang, Guandao Yang, and Leonidas Guibas. Blenderalchemy: Editing 3d graphics with vision-language models. In *European Conference on Computer Vision*, pp. 297–314. Springer, 2024.

Abhishek Joshi, Beining Han, Jack Nugent, Max Gonzalez Saez-Diez, Yiming Zuo, Jonathan Liu, Hongyu Wen, Stamatis Alexandropoulos, Karhan Kayan, Anna Calveri, Tao Sun, Gaowen Liu, Yi Shao, Alexander Raistrick, and Jia Deng. Procedural generation of articulated simulation-ready assets. *arXiv preprint arXiv:2505.10755*, 2025. URL https://arxiv.org/abs/2505.10755.

Mohammad Sadil Khan, Sankalp Sinha, Talha Uddin Sheikh, Didier Stricker, Sk Aziz Ali, and Muhammad Zeshan Afzal. Text2cad: Generating sequential CAD models from beginner-to-expert level text prompts. *arXiv preprint arXiv:2409.17106*, 2024. URL https://arxiv.org/abs/2409.17106.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. *arXiv preprint arXiv:2309.06180*, 2023. URL https://arxiv.org/abs/2309.06180.

Ling Liu, Bingchen Gong, Amal Dev Parakkat, and Maks Ovsjanikov. 3DHarnessBench: Probing agentic 3D-to-code capabilities of frontier vision-language models, 2026. URL https://arxiv.org/abs/2609.06535.

Sining Lu, Guan Chen, Nam Anh Dinh, Itai Lang, Ari Holtzman, and Rana Hanocka. LL3M: large language 3D modelers. *arXiv preprint arXiv:2508.08228*, 2025. URL https://arxiv.org/abs/2508.08228.

Rachel K. Luu and Markus J. Buehler. Bioinspired3D. Hugging Face dataset, https://huggingface.co/datasets/lamm-mit/Bioinspired3D, 2026a. Accessed: 2026-09-21.

Rachel K. Luu and Markus J. Buehler. Bioinspired123d: generative 3d modeling system for bioinspired structures. *AI for Science*, 2(2):025004, 2026b.

Iñigo Quilez and Pol Jeremias. Shadertoy. https://www.shadertoy.com/, 2013. Accessed 2026-09-21.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. In *Advances in Neural Information Processing Systems*, volume 36, pp. 53728–53741, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/a85b405ed65c6477a4fe8302b5e06ce7-Abstract-Conference.html.

Alexander Raistrick, Lahav Lipson, Zeyu Ma, Lingjie Mei, Mingzhe Wang, Yiming Zuo, Karhan Kayan, Hongyu Wen, Beining Han, Yihan Wang, Alejandro Newell, Hei Law, Ankit Goyal, Kaiyu Yang, and Jia Deng. Infinite photorealistic worlds using procedural generation. *arXiv preprint arXiv:2306.09310*, 2023. URL https://arxiv.org/abs/2306.09310.

Alexander Raistrick, Lingjie Mei, Karhan Kayan, David Yan, Yiming Zuo, Beining Han, Hongyu Wen, Meenal Parakh, Stamatis Alexandropoulos, Lahav Lipson, Zeyu Ma, and Jia Deng. Infinigen indoors: Photorealistic indoor scenes using procedural generation. *arXiv preprint arXiv:2406.11824*, 2024. URL https://arxiv.org/abs/2406.11824.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16, 2020. doi: 10.1109/SC41405.2020.00024. URL https://doi.org/10.1109/SC41405.2020.00024.

Danila Rukhovich, Elona Dupont, Dimitrios Mallis, Kseniya Cherenkova, Anis Kacem, and Djamila Aouada. Cad-recode: Reverse engineering cad code from point clouds. In *2025 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 9801–9811. IEEE, 2025.

Chunyi Sun, Junlin Han, Weijian Deng, Xinlong Wang, Zishan Qin, and Stephen Gould. 3D-GPT: procedural 3D modeling with large language models. *arXiv preprint arXiv:2310.12945*, 2023. URL https://arxiv.org/abs/2310.12945.

Michael Tschannen, Alexey Gritsenko, Xiao Wang, Muhammad Ferjad Naeem, Ibrahim Alabdulmohsin, Nikhil Parthasarathy, Talfan Evans, Lucas Beyer, Ye Xia, Basil Mustafa, Olivier Hénaff, Jeremiah Harmsen, Andreas Steiner, and Xiaohua Zhai. SigLIP 2: Multilingual vision-language encoders with improved semantic understanding, localization, and dense features. *arXiv preprint arXiv:2502.14786*, 2025. URL https://arxiv.org/abs/2502.14786.

Karl D. D. Willis, Yewen Pu, Jieliang Luo, Hang Chu, Tao Du, Joseph G. Lambourne, Armando Solar-Lezama, and Wojciech Matusik. Fusion 360 gallery. *ACM Transactions on Graphics*, 40(4), 2021. URL https://www.research.autodesk.com/publications/fusion-360-gallery/.

Rundi Wu, Chang Xiao, and Changxi Zheng. DeepCAD: a deep generative network for computer-aided design models. *arXiv preprint arXiv:2105.09492*, 2021. URL https://arxiv.org/abs/2105.09492.

Shaofeng Yin, Jiaxin Ge, Zora Zhiruo Wang, Chenyang Wang, Xiuyu Li, Michael J. Black, Trevor Darrell, Angjoo Kanazawa, and Haiwen Feng. Vision-as-inverse-graphics agent via interleaved multimodal reasoning. *arXiv preprint arXiv:2601.11109*, 2026. URL https://arxiv.org/abs/2601.11109.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyan Luo. LlamaFactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pp. 400–410, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-demos.38. URL https://aclanthology.org/2024.acl-demos.38/.

Matt Zhou, Ruining Li, Xiaoyang Lyu, Zhaomou Song, Zhening Huang, Chuanxia Zheng, Christian Rupprecht, Andrea Vedaldi, and Shangzhe Wu. Articraft: An agentic system for scalable articulated 3D asset generation. *arXiv preprint arXiv:2605.15187*, 2026. URL https://arxiv.org/abs/2605.15187.

Chuning Zhu, Raymond Yu, Siyuan Feng, Benjamin Burchfiel, Paarth Shah, and Abhishek Gupta. Unified World Models: Coupling Video and Action Diffusion for Pretraining on Large Robotic Datasets. In *Proceedings of Robotics: Science and Systems*, Los Angeles, CA, USA, June 2025. doi: 10.15607/RSS.2025.XXI.015. URL https://www.roboticsproceedings.org/rss21/p015.html.

APPENDIX

A The 3DCodeVerse Harness: Design and Implementation **15**

A.1 Components and model roles . . . 15
A.2 Planning and the job graph . . . 16
A.3 Authoring contracts and language runtimes . . . 17
A.4 Spatial tools and deterministic gates . . . 17
A.5 Automated judgment . . . 19
A.6 Refinement rounds, history, and output selection . . . 22
A.7 A recorded run . . . 22
A.8 Articulated objects: geometry, kinematics, and joint sweeps . . . 23
A.9 Scenes and graphics . . . 25
A.10 Skills . . . 25
A.11 The data engine . . . 26

B 3DCodeVerse-2M: Dataset Details **26**

B.1 Counting units and snapshots . . . 27
B.2 Sources and conversion pipelines . . . 28
B.3 The curation toolkit . . . 29
B.4 Record format and archive layout . . . 29
B.5 Comparison with related resources . . . 31

C Experimental Details and Additional Results **31**

C.1 Training details . . . 32
C.2 Training curves of the 27B data-scale runs . . . 32
C.3 Evaluation protocol . . . 33
C.4 The evaluation judge . . . 37
C.5 Additional open-model results . . . 38
C.6 Additional harness results . . . 40
C.7 Executable articulation and comparison with UWM . . . 42

D Additional Visualizations **46**

D.1 Procedural scenes . . . 46
D.2 Harness outputs across tracks . . . 46
D.3 Corpus samples . . . 46
D.4 Evidence the judge receives . . . 48

E Reproducibility, provenance, and AI use **48**

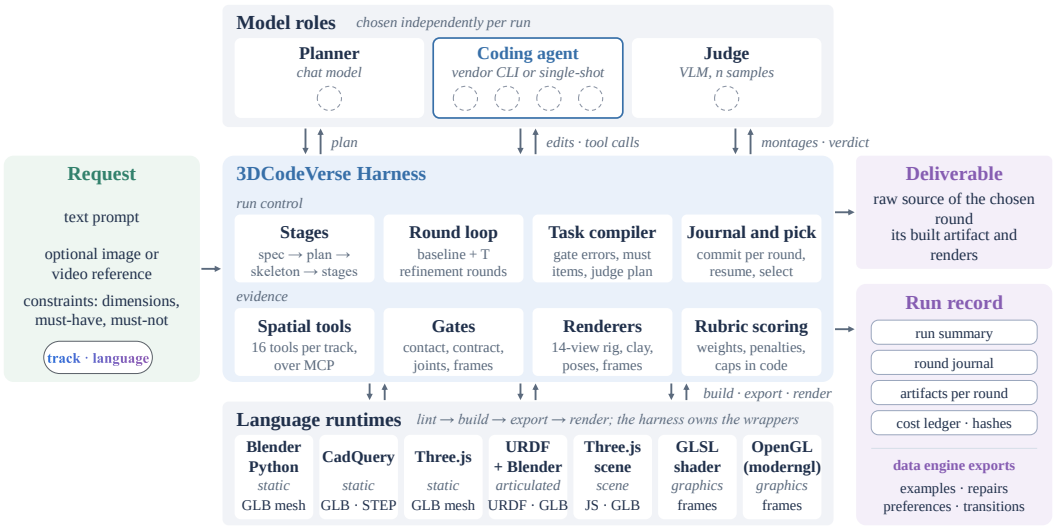

Figure 7: **Components and model roles.** A planner, a judge, and a coding agent are chosen independently per run; the harness owns run control (stages, rounds, revision tasks, output selection) and the evidence (spatial tools, deterministic gates, renderers, rubric scoring). Agent code executes through seven language runtimes; dashed circles mark logo slots.

A THE 3DCODEVERSE HARNESS: DESIGN AND IMPLEMENTATION

This part documents the implementation behind §4 in the order of a run: components and model roles (§A.1), planning and the job graph (§A.2), language runtimes (§A.3), spatial tools and gates (§A.4), automated judgment (§A.5), refinement and output selection (§A.6), a recorded run (§A.7), the articulated (§A.8) and scene and graphics (§A.9) tracks, the skill library (§A.10), the data engine (§A.11), and budgets and reproducibility (§A.6). Studies with different settings are identified in Appendix C.6.

A.1 COMPONENTS AND MODEL ROLES

Tracks and runtimes. Work is organized into four *tracks*, static objects, articulated objects, scenes, and graphics, served by seven *language runtimes* (Table 4): three for static objects, one each for articulated objects and scenes, and two for graphics. Each runtime declares its file layout, lint rules, build wrapper, export format, and render driver; frames, units, view rigs, and tolerances are defined once and shared.

Model roles. The *planner* and the *judge* are chat models behind a common interface to the major commercial APIs; the judge is a vision-language model. The *coding agent* is a vendor command-line coding agent working in a prepared workspace, with the harness tools exposed as an MCP server; alternatively, *single-shot* generation asks a chat model for a complete file envelope without tools. The harness implements no agent loop of its own. Role defaults, judge samples, round count, and the wall-clock ceiling are grouped into three cost profiles.

Design rules. *Code is the deliverable*: every round commits raw source, from which meshes and renders are regenerated. *Contracts are concrete*: rules reach the agent as skeleton code, exact numbers, and copyable snippets. *Code answers when code can*: measurements are deterministic, the judge handles only perception, and the score is computed in code from rubric weights, penalties, and caps. *Responsibility moves out of agent code*: export, cameras, placement validation, URDF composition, scene assembly, and shader headers belong to the harness. *Acceptance is evidence-based*, *inexpensive checks run first* (lint, build, gates, rendering, judging), and *generator and judge are separated*: the judge never sees the generator's reasoning or code. A wall-clock budget caps each run, every model



Figure 8: **From the plan to files to sessions**, taken from the recorded run of §A.7. The plan names parts with boxes, materials, and attachments, plus an acceptance checklist; each target maps to the files it owns, and revision tasks that share files form one session. The first refinement's nine tasks formed eight file-disjoint groups; the second's four formed four.

call is recorded in a cost ledger, and artifacts are *exported as authored*: the gates report defects and the agent corrects the source.

A.2 Planning and the job graph

The plan. The planner turns the request into a schema-validated plan. A static-object plan lists parts, each with a role, description, bounding box, material, optional attachment, symmetry, instance count, detail hint, and up to eight sub-parts inside its box. An articulated plan adds a root link and joints, each with a type (revolute, prismatic, continuous, or fixed), parent and child links, a unit axis, a pivot, limits, a rest value, a described motion, and an optional mimic coupling $q_j = m_j q_i + b_j$ checked for cycles and immobile targets. A scene plan gives zones, assets (Three.js modules or Blender-built GLB files), effects, and one to six cameras; a graphics plan gives render passes, uniforms, and motion. Every plan carries an acceptance checklist whose items name their evidence (measure, visual, probe, or articulation) and priority (must or should). Common articulated-plan slips (limits in degrees, swapped limits, a moving sub-part) are repaired deterministically, and the harness appends must items for requested dimensions (within 5%), the triangle budget, must-have and must-not phrases, grounding when nothing else is measured, and, for graphics, measured motion; on the scene track planner-written items are downgraded to should.

The planner loop. Planning is one structured call at temperature 0.4 (0.5 for graphics) whose output budget grows with the part count. Validation errors allow at most two re-asks; a collapsed plan whose joints reference undeclared links instead restarts once from the original request (Appendix C.6); a quality complaint allows one further re-ask. The loop is bounded at eight calls, and the plan is cached under a hash of the request, constraints, references, model, and seed.

File ownership. The runtime maps each task target to the files it owns: one file per part plus an assembly entry file for Blender and Three.js objects, the whole program for CadQuery, URDF, and graphics, and per-zone, per-asset, and composition modules for scenes. Each session is bracketed by two commits; the harness then reverts writes outside the session's scope or to read-only harness files and marks such a session failed.

Fan-out. An object with at least eight top-level parts in a per-part-file language is generated by scoped sessions over attachment subtrees of at most three parts, each seeing its neighbours' planned boxes, followed by one assembly session that owns the entry file (Figure 8); otherwise one session

Table 4: **Authoring contracts of the language runtimes.** Agents write raw code against the language's own API; the harness owns the wrappers that execute, map errors, export, and render. Every build runs under a 300 s timeout in a fresh process.

Runtime	Track	What the agent writes	What the harness does
Blender Python	static	a main script and one build function per part, each in its own file; Z-up, $-Y$ front, metres, standing on $z{=}0$; named objects, numbered instances	headless Blender, factory settings, seeded randomness; errors mapped to part file and line; per-object census; GLB (Y-up) and STL export
CadQuery	static	a main script whose result is preferably a named assembly; Z-up, $-Y$ front, degrees for rotations	per-part tessellation; GLB, STEP, and STL export; kernel refusals reported with hints; system access and raw OpenCascade calls linted out
Three.js	static	an object module exporting a build function and one module per part at its world pose; Y-up, $+Z$ front	Node export, seeded randomness; instanced meshes baked to numbered copies; an optional self-check that can fail the build; exported as authored
URDF + Blender	articulated	a Blender script building each link in world coordinates at the rest pose, and a hand-written URDF with link frames at pivots, joint origins as pivot differences, visual origins as negated pivots; limits relative to rest	one GLB mesh per link; URDF loading, forward-kinematics consistency and rest-pose collision checks; a hierarchical GLB with joint metadata beside the URDF
Three.js scene	scene	an environment module (terrain height, surroundings), one module per zone and asset, and shaders; Blender-built assets as GLB; a read-only effect library	deterministic scene assembly from probed components and planned cameras; a headless-browser probe compiles every shader and records the census; post-processed renders at two animation times
GLSL shader	graphics	a Shadertoy-style fragment shader, optionally with a common file and a feedback buffer pass; no version line, uniforms, or outputs	a header with time, resolution, mouse, frame index, previous frame, noise, and buffer inputs plus Shadertoy aliases; judged frames, frame sheet, preview GIF; errors mapped to the agent's line
OpenGL (moderngl)	graphics	a Python program with a setup function and a per-frame render function; no window libraries, wall clock, or file writes	a fresh subprocess per render on a GPU context with software fallback; tracebacks and shader errors mapped to the source line

writes the whole program. Refinement tasks are grouped into connected components of the relation "shares a file" and dispatched as separate sessions, except on the graphics track. Command-line sessions on one workspace are serialized by a lock, so true parallelism arises from single-shot generation and from the separate workspaces of best-of-N candidates and scene assets.

A.3 Authoring contracts and language runtimes

Table 4 summarizes each runtime's contract. Agents write raw code against the language's own API with no helper library; lint rules reject imports and calls that bypass the harness (renderers, window systems, file and network access, the wall clock). The harness owns the build wrappers, which run each program in a fresh process under a 300 s timeout and a memory cap, map failures to a file and line in the agent's source, record a census of the built objects, and export geometry in the Y-up GLB frame exactly as authored.

What a session receives. The workspace carries one instruction file, identical for every vendor agent, with the hard rules, the tool cards for the track and language, and the harness and language contracts. The task prompt adds the request, frame conventions, the language cookbook, the plan as exact-number tables with its checklist, and the finishing rule (build, then measure or probe). Single-shot prompts also inline the skeleton and expect an envelope of whole files; a truncated reply is retried once with a doubled output budget. Every prompt is hashed and stored with its session transcript.

A.4 Spatial tools and deterministic gates

Registry and exposure. Each tool (Table 5) is registered once with a typed argument model, a description, its tracks and languages, and a cost hint. It is called directly by the track pipelines, exposed to coding agents through a per-session MCP server, and described in the workspace instructions; agents without MCP support use a command-line fallback.

The observation contract. Every observation carries a verdict and a separate failure flag stating that the tool could not run. A measured collision or a failed build is thus a valid observation whose

Table 5: **The spatial tool set.** A workspace sees only the tools of its track and language (nine for static objects, ten for articulated objects, five for Three.js scenes, three for graphics). The end-of-round gates call the same functions. *obj* = static object, *art* = articulated object, *scn* = scene, *gfx* = graphics.

Tool	Tracks	What it returns
build and measure		
Build	all	lint, execute, export; then the measurement table, probe census, or frame statistics; on failure the error type, source line, and a stderr tail
Measure	obj, art	overall extents, triangles, islands, ground gap, footprint offset; per part size, height range, triangles, and watertightness
Cross section	obj, art	an axis-aligned slice: labelled image, loop count, filled area, *hollow ratio*
seeing		
Render views	obj, art	chosen views in shaded, clay, wire, normals, or silhouette mode, optionally isolated or exploded
Render sheet	obj, art	all fourteen object views as one shaded contact sheet
Isolate part	obj, art	one part rendered alone, with its measurement row
Scene views	scn	authored or overview cameras at chosen times, with frame statistics and measured motion
Frame probe / frame sheet	gfx	one frame at time t, or a sheet of the judged frames with luminance, colour, edge, difference, and NaN statistics and the frame-gate verdict
checking		
Connectivity check	obj, art	floating parts *with the translation that closes the gap*, interpenetrations with depth, stray islands, and the contact ledger
Contract check	obj, art	plan deviations: missing parts, counts, box deltas, orientation, ground gap, footprint, in the authoring frame
Reference comparison	obj, art	reference and render side by side with outline difference and IoU, and a checklist of inventory, counts, proportions, profile, materials
Joint sweep	art	joints swept over rest, limit, and random poses: pose renders, worst overlap per link pair, unattached and buried links
Shader probe	scn (Three.js)	headless compilation of every shader program, with file and line on failure
Scene probe	scn	object, material, and light census; triangles; draw calls against the budget; console errors
Placement check	scn	per asset support, ground gap, burial, contacts, interpenetration, plus plan checks on fog, backdrop, zones, scale, bounds



Figure 9: **Observations, gates, and feedback in the authoring frame.** A measured defect is a valid negative verdict, not a tool error (left); one check serves as tool, gate, and source of revision tasks (middle). Right: baseline-round findings of §A.7, each with a correction in the agent's own frame.

Table 6: **Deterministic gates and the score caps they impose.** An *error* finding becomes a revision task and, where a cap rule exists, bounds the judged score.

Track	Gate	Error condition (warnings in brackets)	Cap
objects	lint	forbidden imports, renderer or window creation, missing entry symbols	—
	connectivity	a part unconnected to the ground-supported component (contact $\leq$ 2 mm); penetration over 10 mm with $\geq 2\%$ of samples inside [over 2 mm; stray islands; nothing on the ground]	0.60 / 0.70
	contract	missing part; box beyond $3\times$ the tolerance $\max(1\,\mathrm{cm}, 0.10 \times$ planned extent$)$; lying down or stood on end; ground gap over 3 cm [beyond $1\times$; footprint offset]	0.75
	detail budget	triangles above $\min(600\,\mathrm{k}, \max(60\,\mathrm{k}, 20\,\mathrm{k} \times$ objects$))$ [below $\max(1{,}500, 350 \times$ objects$)$, at most 40 k]	—
	reference silhouette	[outline IoU below 0.6; adds a revision task]	—
articulated	FK consistency	rest forward-kinematic box off the mesh by over 1 mm; build fails with the corrected origin	0.30[†]
	rest-pose collision	penetration over 5 mm at rest; build fails	0.30[†]
	joint sweep	overlap over 2 mm in a moved pose; child over 30 mm from its parent (fixed: 2 mm); buried link [revolute gap up to 10 mm]	0.60
	motion direction	child moves against the planned direction; axis may be auto-repaired	—
scenes	scene probe	boot, import, update, or console failure; over 15,000 draw calls fails the build [over 5,000; over 3 M triangles]	0.50
	shader preflight	a shader fails to compile; build fails	0.50
	placement	floating over 15 cm or more than three floating; sunken over 30 cm and a quarter of its height; box overlap over 60% [floating over 5 cm outdoors, 2 cm indoors; sunken over 10 cm; overlap over 20%]	0.60[‡]
	render console	a console error during rendering [frame rate]	0.60
	scene frames	a camera frame dark (mean luminance below 0.12 or over 35% near black), blown (over 20% white), or flat (one band over 92%); establishing shot under 20% content; camera within 0.5 m of geometry; no motion; hero asset unseen	0.55–0.65
graphics	lint	window or file access, wall-clock time, wrong entry signature	0.60
	frame gate	NaN or infinite pixel; every frame over 97% black or white [max inter-frame difference below 0.002 (static) or above 0.35 (flicker); edge density below 0.002]	0.0 / 0.50 / 0.70[§]

[†] Any URDF error caps at 0.30; a failed build is not judged but enters the repair loop. [‡] Floating, sunken, or unsupported assets; box overlap has no cap rule. [§] NaN 0.0, black or white 0.50, static 0.50, flicker 0.70; these two are the only capping warnings.

text leads with FAIL; only the failure flag maps to the MCP error flag, because vendor agents retry calls flagged as errors (Appendix C.6). Payloads are bounded to 6,000 characters of text and 400 kB of images with a long side of at most 1,024 pixels, with no image on a failed call and one on a failing verdict.

Gates. After each successful build the track runs its deterministic gates (Table 6). Each finding carries a severity, a target, a message, a fix hint, and a typed finding kind. Errors become revision tasks and, through cap rules keyed on the finding kind, bound the judged score; warnings are shown to the judge and cap only for static or flickering frames on the graphics track. A passed check can also veto a visual defect claim (§A.5).

Connectivity and contract. In the connectivity gate, the ground-touching component with the most faces is the support, and each part outside it is reported with the nearest supported part and the closing translation. A contact ledger marks each planned attachment as contact or open with its measured gap; an open planned join is not an error on its own. The contract gate compares the built object with the plan under a per-axis tolerance (Table 6).

Feedback in the authoring frame. Meshes are measured in the Y-up export frame, but fix hints are converted back to the labelled authoring frame: a Blender seat floating 12 mm is told to move by $(0, 0, -0.012)$ m, and the same fault in Three.js by $(0, -0.012, 0)$ m (Figure 9). Articulated findings print a URDF origin or axis, and scene findings use the scene language's API vocabulary.

A.5 AUTOMATED JUDGMENT

Every Judge Score and VLM Preference reported in §5 is produced by this judgment procedure with a fixed model (Gemini 3.1 Pro), rubric, and evidence; Appendix C.4 describes that configuration.



Figure 10: **The judge: payload, answer, and score.** The judge receives text evidence and labelled montages of rendered views, never the code or the generator's reasoning (left); its answer schema forces observation before scoring (middle). Right: the score of the baseline round of §A.7, computed in code from criterion means, voted defects, and measured caps.

Payload. One payload builder serves the in-run judge, re-judging, and calibration (Figure 10). The text contains the request, a plan digest, the acceptance checklist, measurements, gate findings, track-specific results, and the previous verdict with an instruction to judge the current round on its own merits. On object tracks a *measured structure* block summarizes the contact ledger and, when connectivity passed, forbids floating and interpenetration claims. The images are at most five labelled 2×2 montages (three on the scene and graphics tracks), up to two detail crops, and, after a connectivity error, up to two cross-sections; with several judge samples, montage and tile order are shuffled per sample.

Answer schema. The judge answers in JSON at temperature 0.2, listing a summary, strengths, issues with evidence, the defect checklist, and the acceptance items before the criterion scores, followed by at most six improvement instructions; measured criteria such as outline IoU are injected by code. A parse failure triggers up to three attempts per sample at rising temperature.

Score arithmetic. For round t, the scores of each rubric criterion k (Table 7) are averaged over samples into $\bar{s}_k^{(t)}$; acceptance items and defects are decided by majority vote, a tie following the sample closest to the mean for acceptance and counting as absent for defects. A voted defect is vetoed, with neither penalty nor cap, when the gates that measure it ran, passed, and reported no error of the corresponding kinds and no penetration warning of 8 mm or deeper. The score is

$$s^{(t)} = \min\left\{ u^{(t)},\ \max\left(0, \sum_k w_k \bar{s}_k^{(t)} - \sum_d \lambda_d \hat{d}_d^{(t)} \right) \right\}, \tag{5}$$

where $\hat{d}_d^{(t)} \in \{0, 1\}$ marks retained defects with penalties λ_d and $u^{(t)} \in [0, 1]$ is the tightest applicable cap among retained defects, fired gate rules, and unverified must items ($u^{(t)} = 1$ when nothing fires); on object rubrics the acceptance cap is graded as $0.6 + 0.4\, v/m$ for v of m verified items. A round passes when $s^{(t)}$ reaches the rubric threshold, no criterion falls below its floor, and every must item is verified. The stored verdict records every applied cap with its evidence; a judge outage leaves the round unscored rather than scored zero, and the round is re-judged once (§A.6).

Other judges and calibration. A pairwise judge declares a winner only when both display orders agree; a reference judge adds the measured silhouette criterion, and a likeness judge attaches real-

Table 7: **Track rubrics.** Each verdict carries the rubric's content hash. Floors affect only the pass flag; *measured caps* are triggered by gate findings, console errors, or missing views, not by the judge.

Rubric (pass)	Defects	Criteria (weight, floor)	Measured caps
static object (0.72)	11	intent fidelity (.22, .30) · structure (.18, .25) · geometry detail (.16) · proportions and scale (.12) · assembly fit (.12) · materials (.10) · craftsmanship (.10, .30)	build error 0 · floating part 0.60 · penetration 0.70 · contract 0.75 · must items from 0.60 (graded)
articulated (0.70)	14	intent fidelity (.16, .30) · structure (.14, .25) · articulation (.18, .30) · joint placement (.12) · geometry detail (.12) · proportions (.10) · assembly fit (.10) · craftsmanship (.08, .30)	URDF error 0.30 · joint collision 0.60 · unattached link 0.60 · floating part 0.60 · penetration 0.70 · contract 0.75 · no pose sheet 0.50 · must items from 0.60 (graded)
scene (0.70)	13	brief fidelity (.18, .30) · composition (.14, .25) · lighting (.14, .25) · asset quality (.14) · spatial coherence (.12, .25) · materials and effects (.12) · animation (.08) · technical cleanliness (.08, .30)	floating or sunken 0.60 · console 0.60 · shader 0.50 · camera in geometry 0.50 · dark 0.55 / blown 0.60 / flat 0.65 / small content 0.60 · must items 0.60
graphics, shader v2 (0.70)	15	likeness (.26, .30) · depth and detail (.16) · light and tone (.16) · composition (.10) · motion quality (.14, .20) · technical cleanliness (.10, .30) · craft (.08)	NaN 0 · black or blown 0.50 · static 0.50 · flicker 0.70 · lint 0.60 · must items 0.70
scene asset (0.70)	7	identity (.22, .30) · all sides finished (.16) · sub-parts (.16, .25) · proportions (.14) · detail level (.12) · material truth (.10) · cleanliness (.10, .30)	as static object, without contract
reference (0.70)	7	silhouette (.25, measured IoU) · proportions (.20, .25) · parts coverage (.20, .25) · material and colour (.15) · detail (.10) · cleanliness (.10, .30)	as static object, without contract

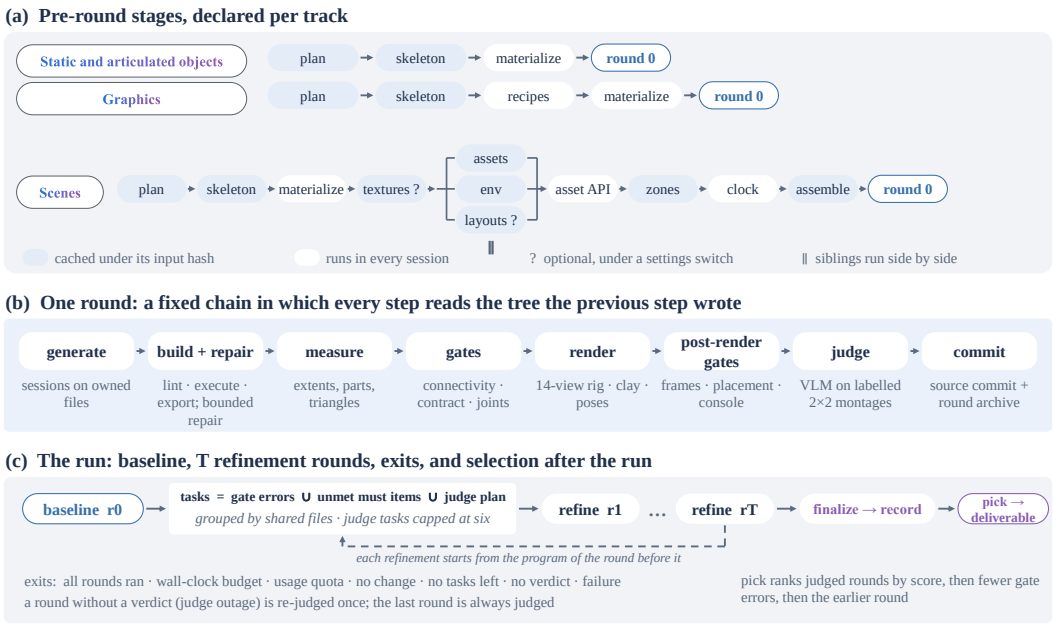

Figure 11: **Run lifecycle.** (a) Pre-round stages are declared per track and cached for resumption. (b) One round is a fixed chain in which every step reads the tree the previous step wrote. (c) A run is a baseline and T refinement rounds, each starting from the program of the round before it; the delivered round is chosen after the run.

world photographs. Re-judging chair and cabinet rounds three times with shuffled order gives a mean score standard deviation of 0.030 for a Gemini 3.1 Pro judge and 0.083 for a Gemini 3.8 Flash judge, and 0.035 with fixed order. These values measure judge repeatability, not the variation between independently generated artifacts (Appendix C.3).

A.6 Refinement rounds, history, and output selection

Stages and rounds. Every run proceeds as spec, plan, skeleton, track-specific stages, a baseline round, T refinement rounds, finalization, and the record (Figure 11). Pre-round stages form a per-track graph whose nodes are cached under a hash of their inputs. A round is a fixed chain: generate, build with bounded repair, measure, gates, render, post-render gates, judge, and commit; nothing inside a round is cached. A failed build triggers up to three repair sessions, each given the error location, a traceback tail, the lint findings, and the matching cookbook section; a repeated error signature raises the repair temperature and thinking level. Each completed round commits the source, keeps its built artifact, and appends a journal entry with its build, gates, measurement, renders, judgment, instructions, usage, and timing.

Revision tasks. Refinement tasks are compiled from the previous round in priority order: gate errors with their fix hints, then harness-derived tasks and failed must items, then the judge's improvement plan. Tasks are de-duplicated by target and kind, and targets are canonicalized to plan names. At most six tasks are kept, but the cap removes only judge tasks, so gate and acceptance tasks are never dropped; each task's instructions are compacted to at most six lines. A reference-conditioned run adds a silhouette task when the outline IoU is below 0.6. Grouping and fan-out follow §A.2.

Round policy and exits. The default budget is four refinement rounds, sixty minutes, and three repair attempts. The loop stops when every requested round has run, the wall-clock budget is spent, the vendor's usage limit is reached, a refinement round changes nothing, no refinement tasks remain, the last round has no verdict even after re-judging, or the run fails. A transient provider failure re-runs the same round once, and a round without a verdict is re-judged once. The wall-clock guard is checked at stage boundaries and around every session, whose timeout it clips to the time left; money never gates a run.

Candidates and resumption. When requested, the baseline is generated N times, each candidate judged on a quick four-view rig and ranked by build success, quick score, and fewer gate errors; the winner becomes round zero. The journal, stage cache, budget snapshot, and per-round git commits allow a run to resume at its last committed round with the clock continuing where it stopped. Finalization leaves the workspace at the last round, not at the best one.

Selection after the run. Output selection is a separate operation over the recorded rounds. The default selector ranks judged rounds by score, then fewer gate errors, then the lower round index. An optional pairwise selector is consulted only when the top two scores lie within 0.03 and replaces the leader only when it favours the runner-up with confidence of at least 0.6. The chosen round is packaged from its commit and kept artifacts, without rebuilding, with a manifest listing every file's role, size, and SHA-256.

Budgets and cost accounting. Every model call and agent session is one row of the run's cost ledger, recording the stage, role, model, token counts, applied prices with their provenance, latency, and outcome; a terminated session is priced from its event stream and marked as an estimate. The record's total usage is the ledger sum at list price, and a run's active minutes exclude time lost to provider errors, whereas the wall-clock guard counts that time against the budget. A profile sets the roles and the budget and is re-applied on resumption. For reproducibility, the record stores hash prefixes of the language contract, the cookbook, every prompt and attached skill, the rubric hash, a settings snapshot, tool versions, and the harness commit.

A.7 A recorded run

In the run of Figure 12, the planner produced a seven-part plan (a seat slab, three legs, and three tenon wedges, all attached to the seat) with eleven acceptance items, four of them signature features. The baseline built, but the connectivity gate reported three wedge–seat penetrations of about 22 mm and six floating parts up to 212 mm from their nearest supported part, two legs lay outside their planned boxes at three times the tolerance, and the model had 1,572 triangles against a floor of 2,450. The VLM judge voted the floating-part and interpenetration defects, reducing the weighted criterion mean of 0.59 by 0.18 to 0.41. Nine revision tasks, eight from gate errors and one from the unmet footprint

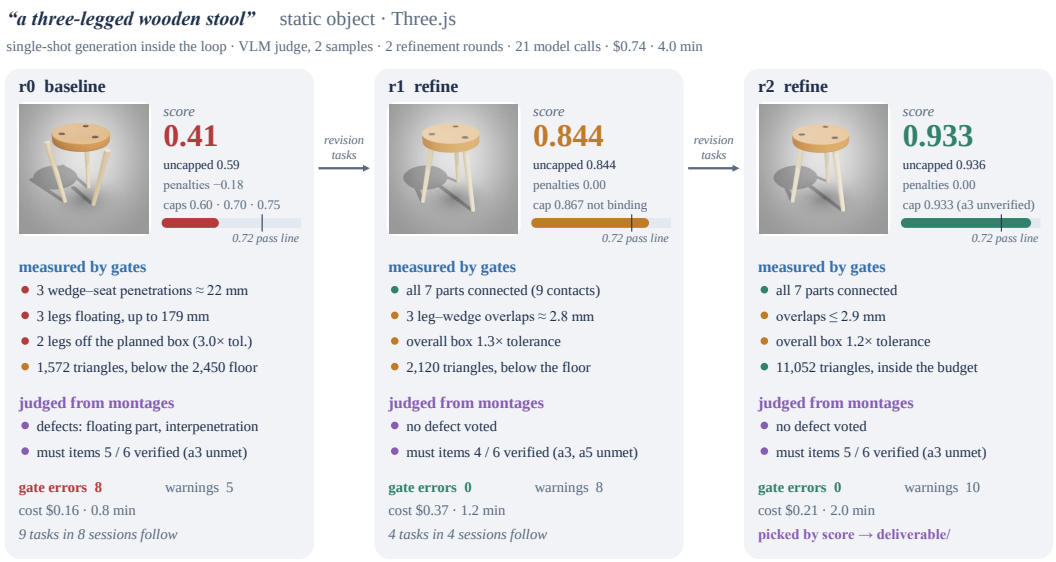

Figure 12: **A recorded run** of the request "a three-legged wooden stool" on the Three.js runtime under the lowest-cost profile. Each column shows one round: the view the judge saw, the score with its penalties and caps, the gate findings, and the judge's votes. Floating and embedded parts capped the baseline and became revision tasks; the second refinement reached 0.933, held below a pass only by the unmet footprint item, and was selected.

item, were dispatched as eight file-disjoint sessions. The first refinement connected all seven parts with nine contacts and left only warnings; its score of 0.844 lay below the cap of 0.867 imposed by four of six verified must items. The second refinement, driven by the footprint item and three judge instructions (a modelled wedge slot, a chamfered rim, more splay), reached 11,052 triangles and a score of 0.933, capped at $0.6 + 0.4 \cdot 5/6$ by the one unverified item: a measured footprint of 0.34 m against the planned 0.38 m. The run used 21 model calls, $0.74, and 4.0 active minutes, and the selector packaged round two. Representative revision tasks from the round journal (lightly reformatted) follow.

```
[gate:connectivity] LegRearLeft: part 'LegRearLeft' is floating: nearest supported part
  is 'LegRearRight' at 178.8 mm  FIX: translate 'LegRearLeft' by (+0.1788, +0.0000, +0.0000)
    m
[acceptance] overall: Acceptance item a3 NOT met: Tripod base footprint diameter is
  0.380 m +/- 0.010 m, wider than the seat diameter
[judge/geometry] SeatSlab: Bevel or chamfer the top and bottom rim of the circular
  seat slab.
```

A.8 Articulated objects: geometry, kinematics, and joint sweeps

The articulated track manages two source files that must agree (Figure 13). The Blender script authors one mesh per link in world coordinates at the rest pose, and the URDF places each link frame at its pivot. With $\mathbf{T}_l(0)$ the rest-pose frame of link l and $\mathbf{T}_{v,l}$ its visual origin, a world-authored mesh is consistent with the kinematic tree when

$$\mathbf{T}_l(0)\,\mathbf{T}_{v,l}\,\mathbf{x}_{\text{world}} = \mathbf{x}_{\text{world}}, \qquad \mathbf{T}_{v,l} = \mathbf{T}_l(0)^{-1}. \tag{6}$$

The build compares each link's forward-kinematic bounding box at $q = 0$ with the authored mesh; a discrepancy above 1 mm fails the build and reports the corrected origin, as does a rest-pose penetration deeper than 5 mm. This establishes agreement between the two components, not that the kinematic model matches a real mechanism.

The joint sweep evaluates the rest pose, each independent joint at its lower, middle, and upper limit, and eight seeded random combinations when more than one joint moves; mimic followers are resolved through the coupling graph. Overlaps deeper than 2 mm in a moved pose (5 mm at rest)

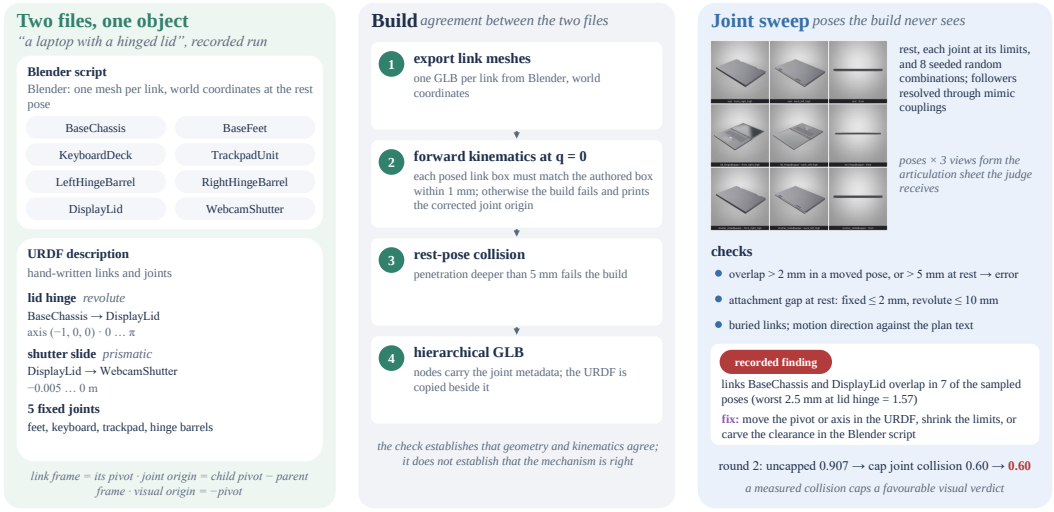

Figure 13: **Articulated objects**, from a recorded run of "a laptop with a hinged lid". Link geometry is written in Blender and kinematics in URDF; the build checks their agreement, and the joint sweep poses the mechanism through its limits. In this run a 2.5 mm lid collision capped an otherwise favourable verdict.

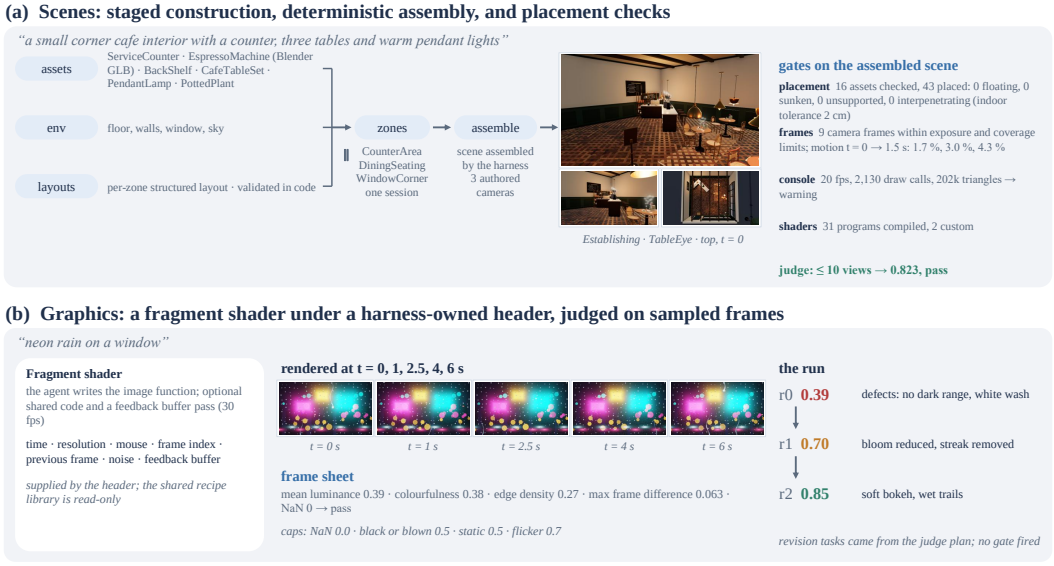

Figure 14: **The scene and graphics tracks.** (a) A scene is built in stages and then enters the common round loop with scene-specific gates. (b) A graphics program is a fragment shader whose frames are sampled at fixed times, measured, and judged as montages. Renders and scores are from recorded runs.

are errors, aggregated per link pair; a moving child more than 30 mm from its parent at rest, or a fixed child more than 2 mm away, is unattached. A motion-direction check compares each joint's displacement with the plan text and proposes a corrected axis. The judge receives the articulation sheet with these findings, and the rubric caps a round at 0.60 for a joint collision or an unattached link, 0.30 for a URDF error, and 0.50 for a missing pose sheet (Table 7). Appendix C.6 reports an ablation on explicit coupling declarations.

Table 8: **The skill library.** Seventeen bundles, each one instruction file with one reference file, at most 350 lines and 2,500 tokens. Standing routes attach a bundle by track, language, and plan signals; gate-fired routes attach it when the previous round's findings of that kind fired. At most five bundles enter a session; inherited bundles are routed only when unverified routing is enabled (the default).

Bundle	Guidance	Standing route	Gate-fired route
part-contact	contact, weld-overlap, penetration, and stray-island limits of the connectivity gate	multi-part objects	connectivity, joint sweep
bbox-contract	plan dimensions: tolerance $\max(1\,\mathrm{cm}, 10\%)$, error past $3\times$	object tracks	contract
repeats-and-mirrors	instances, mirror and radial arrays, instance naming	instances or symmetry	instance box
blender-forms	form-to-Blender-API techniques; five common lint errors	Blender, URDF+Blender	part not imported
cadquery-forms	CadQuery forms; fillet, revolve, and boolean rules	CadQuery	—
threejs-forms	form word to Three.js geometry; no CSG	Three.js objects	—
urdf-joints	axes, limits, clearance; direction of $+q$	articulated	joint sweep, motion direction
scene-composition	layout; cameras from measured bounds	scene sessions	camera, content, hero
scene-lighting	key, fill, and practical lights; linear-light albedo	scenes	dark, flat, blown frames
scene-motion	visible motion between the two animation times	scenes	no motion
threejs-shader-traps	custom GLSL that passes the shader preflight	custom shaders	shader preflight
glsl-craft	fragment-shader craft for the five sampled frames	GLSL graphics	frame gate
opengl-pipeline	moderngl setup and render contract; time sampling	OpenGL graphics	frame gate
scene-atmosphere	sky dome, environment bake, fog	scenes	—
scene-water	water as a dielectric	water in plan	—
scene-night	emissive windows, glow, light shafts	night in plan	—
scene-materials	procedural albedo variance, per-instance jitter	scenes	—

A.9 Scenes and graphics

Scenes. Scene construction is staged before the first round (Figures 11a and 14a). Assets (at most eight, four once the soft budget is spent) climb one ladder: a single-shot draft, a deterministic check, one repair, and only then an agent session; Blender-built hero assets are also judged and given one fix pass that is undone when it scores worse. The environment and zone layouts are built concurrently, one session then writes every zone so that composition can span zone boundaries, and the harness assembles the scene deterministically. The staged baseline may use 0.55 of the wall clock, after which remaining stages degrade to single-shot generation. Scene gates (Table 6) cover asset placement, console errors, exposure and coverage per authored camera, pixel-difference motion between two animation times (passed to the judge as a fact), unused assets, and a budget of 15,000 draw calls. The judge sees at most ten views, drawn from the authored cameras and up to three overview views.

Graphics. The agent writes a Shadertoy-style fragment shader, optionally with a feedback pass, or a moderngl program with setup and render functions; the harness supplies the version line, uniforms, outputs, and read-only brief-matched recipes (Figure 14b). Frames are rendered at $t \in \{0, 1, 2.5, 4, 6\}$ s, clipped to the loop duration, at 1280×720 by default. The frame gate measures luminance, colourfulness, edge density, inter-frame difference, and invalid pixels; NaN, all-black, and all-blown frames are errors, and static or flickering sequences are warnings that still cap the score (Table 7). The judge receives montages with the frame statistics as text, and refinement is one whole-program task per round.

A.10 Skills

The authoring skills of §3 and §4 are seventeen bundles in the open agent-skill format, each an instruction file with one reference file, bounded at 350 lines and 2,500 estimated tokens, and labelled with an evidence level of measured, mixed, or inherited (Table 8). A routing table of 28 rows selects bundles per session: standing rows match the track, language, session kind, and plan signals, and gate-fired rows match the kind of a finding from the previous round, so that a shader compile error attaches the shader-traps bundle. The five highest-priority matches are attached; inherited bundles are routed only when unverified routing is enabled, which is the default. Bundles are copied into the skill directories of every agent interface, whose own bundled skills are disabled. Reads are measured from the agent's tool calls, with an unrouted control bundle to detect indiscriminate reading. Numbers quoted in seven bundles are pinned to the live constants by tests, so that these bundles cannot drift from the gates they describe.



Figure 15: **The harness as a data engine.** Read-only exporters turn the records of a run into program examples with quality tiers, preference and repair pairs, feedback-conditioned transitions, and captions, each carrying its provenance; training consumes these exports separately from the generation loop. The example numbers are from the run of §A.7.

A.11 The data engine

What a run leaves. Every run writes a typed record with tool versions and prompt and rubric hashes, a journal of rounds, a git history with one commit per session and per round, the kept artifacts, every verdict, and a cost ledger (Figure 15); exporters only read finished runs.

Program examples. The exporter takes the selected round (or, when nothing was judged, the latest round that built and has renders) and writes the source tree, renders, captions, and metadata with provenance, score, and a quality tier: A for a passing round without gate errors, B for a passing round with gate errors, C for any other round scoring at least 0.6, and D otherwise. Tiers are filtering attributes rather than guarantees; the stool run of §A.7 exports as tier C because its footprint item remained unverified. Exact duplicates can be dropped, and near-duplicates are marked by a comment-insensitive code hash. Samples are packed into tar archives of at most 2.5 GiB with verified byte-range locators, the layout of Appendix B.4.

Pairs and transitions. Preference pairs join judged rounds $i < j$ of one run with different commits and a score gain of at least 0.05, with the improvement plan of round i as the reason. Repair pairs join a failed build with the next round that builds, or a generated snapshot with its in-round repair, keeping the structured error. Feedback-conditioned transitions record, for each refinement round, the previous round's issues, improvement plan, and gate errors, the instructions given, the diff, and the score change, labelled improved, unchanged, regressed, or unscored at the same 0.05 threshold; rounds whose predecessor is unbuilt, unjudged, or uncommitted are dropped. The stool run yields three preference pairs and two improved transitions. One export contains 254 transitions from 87 runs (112 improved, 81 unchanged, 59 regressed, 2 unscored; median gain $+0.197$ among improvements); the experiments of this paper do not train on transitions.

Captions. An optional captioner produces a render-grounded description, an instruction naming the target language, and a code-grounded construction recipe, with vendor and API names filtered.

B 3DCodeVerse-2M: Dataset Details

This part complements §3. It first fixes the counting units used throughout the paper (§B.1), then documents the sources and conversion pipelines (§B.2), the curation toolkit (§B.3), the on-disk record format and archive layout (§B.4), and closes with the comparison with related resources (§B.5).

Table 9: **Rows of the unified text export by source.** Rows count caption and representation variants, not programs; the three variants (original format, text export, flattened shader passes) are described in the text.

Source	Rows	Source	Rows
Shadertoy	972,014	Thingiverse OpenSCAD	6,989
DeepCAD → CadQuery	765,676	Three.js coding agents	3,050
3DCodeBench factories and instances	170,188	animation2code	2,834
Articraft (CadQuery, URDF)	71,850	Blender coding agents	2,398
Bioinspired3D	33,287	**total**	**2,028,286**

Table 10: **Catalogue and training statistics per dialect.** *Samples* counts curated programs in the packaged catalogue (Appendix B.1); *Pairs* and *Mtok* count the prompt–code pairs and tokens from which the fine-tuning mixtures of Appendix C.1 are drawn, and differ from the 2.03 M text–code pairs of the unified export (Figure 2, right). URDF samples are released with their source code and associated assets. A dash denotes a statistic that is not reported. *Eval* gives the number of held-out execution tasks, with the pass rate of the reference code in parentheses; the five corpus suites total 593 tasks, and 3DCodeBench adds 212.

Dialect	Track	Source	Samples	Pairs	Mtok	Eval (ref.)
CadQuery	static object	DeepCAD → CadQuery, Articraft	135,083	390,449	395.5	200 (199/200)
GLSL	graphics	Shadertoy	119,961	353,061	470.4	200 (178/200)
Blender Python	static object	Bioinspired3D, Infinigen factories, distilled	6,909	19,609	15.2	3DCodeBench 212 + 103 (96/103)
URDF	articulated obj.	Articraft	6,146	—	—	—
OpenSCAD	static object	Thingiverse	1,193	3,312	4.9	50 (50/50)
Three.js	static obj. / scene	distilled, GitHub	305	549	2.5	40 (40/40)
Catalogue total			**269,597**			
web animation	graphics	animation2code	1,051	1,616	2.5	—
Training corpus				**768,596**	**891.0**	5 suites, 593 tasks

B.1 Counting units and snapshots

Three distinct quantities appear in the paper and must not be conflated.

Catalogue records. The packaged catalogue counts one record per curated program in the release archives. At the packaging snapshot it contains 269,597 records across nine collections (Table 11); the per-collection counts in Table 10 are those records, and the coding-agent and repository subsets that were still released as raw folders at that time are counted by their live folders.

Export rows. The unified text export, from which training mixtures are drawn, contains 2,028,286 rows across 342 export folders: 891,788 rows in the original format of each collection, 848,819 rows of a text-export variant, and 287,679 rows of the flattened multi-pass shader variant. Rows multiply records because every program is exported once per caption style and, for multi-pass shaders, once per pass, and because overlapping representations of the same program coexist; export rows must therefore not be read as unique programs or as disjoint examples. Table 9 gives the rows per source. Image-conditioned exports are registered separately (276 registries) and are not included in the text total.

Training pairs. The experimental training statistics count 768,596 prompt–code pairs and 891 M tokens, including the web-animation data, before the individual mixtures of Appendix C.1 are selected; Table 10 reports them next to the held-out suite sizes. The unified release also contains reference programs of the 3DCodeBench lineage; the training partitions for the benchmark experiments are specified separately and reserve the benchmark factory names, the seeded instance subsets, and the per-dialect test splits (Appendix C.1).

Table 11: **The packaged catalogue**, one row per collection and dialect at the packaging snapshot. Origin terms never replace the dataset-level CC-BY-NC-SA-4.0 terms; the stricter applicable terms control reuse. *Render entries* count the render paths recorded per sample, which include mesh exports next to images. †extrapolated from a $10\times$ sample. ‡under-counts, because only a subset was re-rendered at packaging time. §live folder counts at the snapshot; the released coding-agent collections are listed in §B.2.

Subset	Dialect	Samples	Share	Code MB	Render entries	Origin terms
DeepCAD	DeepCAD → CadQuery	128,201	47.55%	136.7	769,107	MIT
Shadertoy	GLSL	119,961	44.50%	394.7	1,196,826†	per shader, mostly CC BY-NC-SA
Articraft (CadQuery)	CadQuery	6,882	2.55%	120.0	3,300‡	CC BY 4.0
Bioinspired3D	Blender Python	6,343	2.35%	9.3	31,715	MIT
Articraft (URDF)	URDF	6,146	2.28%	103.4	2,846‡	CC BY 4.0
Thingiverse-OpenSCAD	OpenSCAD	1,193	0.44%	4.9	5,965	per item
3DCodeBench factories	Blender Python	486	0.18%	16.0	2,430	BSD-3-Clause
Three.js, agent-written§	Three.js (JS/HTML)	280	0.10%	3.4	1,120	CC BY 4.0 per sample
Blender, agent-written§	Blender Python	80	0.03%	0.9	320	CC BY 4.0 per sample
Three.js repositories	Three.js / R3F	25	0.01%	11.3	0	as detected
Total		**269,597**	**100.00%**	**800.6**	**2,013,629**	dataset: CC-BY-NC-SA-4.0

B.2 Sources and conversion pipelines

DeepCAD to CadQuery. DeepCAD ships 178,238 sketch–extrusion command sequences rather than programs (Wu et al., 2021). A converter turns each sequence into a standalone CadQuery program: 174,981 sequences convert (98.2%), 174,177 produce a valid solid (97.7%), and hashing the source vectors removes duplicates to 128,201 unique programs, or 71.9% of the raw count; conversion accounts for about 2% of the reduction and deduplication for the rest. Fidelity against the original solids is measured by voxel IoU at 48^3: mean 0.985 with 96.9% of programs above 0.9. The low-fidelity tail (1,114 programs) is tagged per sample rather than dropped, so that a downstream user can set a threshold. Language descriptions from Text2CAD (Khan et al., 2024) are attached to 125,162 programs under that dataset's terms, and four views are rendered per program. Because DeepCAD's official split contains duplicate groups that span its training and test sets (3,230 groups), and because several public CAD corpora share DeepCAD lineage, cross-source deduplication is a prerequisite for merging further CAD sources.

Shadertoy. The graphics collection holds 119,961 shaders, 98,164 single-pass and 21,585 multi-pass, one indexed entry per shader (Quilez & Jeremias, 2013). The entry (image) pass is stored as the code file and the complete multi-pass source alongside it, so nothing is lost although only the entry pass is indexed; renders are frame grids and a short clip, with input textures kept where a shader samples them. For training, multi-pass programs are flattened by prepending the shared tab to each pass, giving 287,679 flattened rows; automated repair of 588 programs raised the compile rate of the flattened set from 84.6% to 99.0% (92,534 of 93,434), and programs that still fail to compile are excluded from training.

Articraft. The 9,996 released Articraft records (Zhou et al., 2026) divide into 6,882 records with mesh geometry and 3,114 primitive-only records. The former are converted into standalone CadQuery programs through a clean-room reimplementation of the Articraft geometry helpers and a tree-shaking inliner (median about 900 lines per program), and verified against the source geometry by Chamfer distance below 0.02 and a bounding-box ratio within $[0.9, 1.1]$; 6,878 of 6,882 pass, and the conversion fields are retained per sample. The primitive-only records are released as URDF descriptions in a geometry-only and a textured variant (6,146 records in total, with 3,032 textured). The CadQuery and URDF subsets are disjoint sets of source records, not two representations of the same assets. Renders use eight views for the CadQuery programs and the rendered poses of the URDF loader.

Bioinspired3D. The 4,557 upstream prompt–code rows (Luu & Buehler, 2026a) each yield a geometry variant, and the 1,786 rows carrying material data also yield a textured variant, giving 6,343 Blender programs; every program passes execution, render, and export validation, with five renders

per sample. Code is light (9.3 MB in total), which makes this subset a convenient smoke test for Blender tooling.

Thingiverse OpenSCAD. The upstream collection holds 7,378 community projects, of which 1,193 parametric OpenSCAD projects are kept together with their original files; this is the only collection with genuinely per-item licensing, and it is fully captioned, with five renders per sample.

3DCodeBench factories. The 243 procedural factories derived from Infinigen (Raistrick et al., 2023) are released as a geometry and a textured variant each (486 programs); factories average about 880 lines, by far the densest code in the corpus. Running every seed yields 12,720 instances, of which the curated instance subsets (1,953 geometry and 1,951 textured) are released separately; the 212 3DCodeBench evaluation tasks are drawn from the same generator family, so factory names and instances are reserved for evaluation (Appendix C.1).

Agent-written programs. Two collections are written from prompt specifications by coding agents rather than harvested: 949 accepted Blender programs from 1,040 attempts and 1,134 accepted Three.js scenes from 1,389 attempts at the released snapshot, produced by coding agents and accepted only when the program executes and its render check passes (four Cycles views and a mesh export for Blender; four browser views for Three.js). A further 25 real multi-file Three.js projects are released as complete repository trees, the only place where the multi-file path of the record format is exercised, and 1,051 web-animation cases in 21 categories supply the *web animation* row of Table 10.

Harness-generated programs. Programs generated by the 3DCodeVerse Harness enter the corpus through the exporter of Appendix A.11, with their quality tier, acceptance results, gate summary, and provenance; in the reported training experiments this path supplies the 15,704 execution-verified bootstrap programs used in the fine-tuning mixtures.

B.3 The curation toolkit

The agentic curation loop of Figure 2 rests on a command-line toolkit whose stages run in a fixed order: conversion, validation, duplicate checking, execution, rendering, contact sheets, captioning, and publication. *Validation* builds the metadata record of a project from its files: the identifier, the dialect inferred from the entry file's extension, the modality, and the code size, with warnings for programs over 5,000 lines or 1 MB and for missing references or prompts; the only hard error is the absence of a code file. *Duplicate checking* fingerprints each project three ways: a hash of the code with strings, comments, and whitespace removed, a hash of the first mesh's vertices after centring, scale normalization, and rounding, and a perceptual hash of the first render; a match against the registry (identical code or geometry, or a perceptual-hash Hamming distance of at most six) is flagged for review rather than dropped. *Execution* runs each dialect in its pinned environment (headless Blender with factory settings, CadQuery in process, OpenSCAD and FreeCAD through their command lines) with a 600 s timeout, and records success only when an artifact with geometry results; a missing runtime is recorded as not available rather than as a failure. *Rendering* produces four turntable views at $0°$, $90°$, $180°$, and $270°$ at 512×512 pixels with a camera at 2.2 times the object extent, in clay or textured mode, exporting a mesh first for non-Blender dialects, and *contact sheets* tile the views for review. *Captioning* scripts prompt a vision-language model with the renders and the code for the three caption fields of §B.4. *Publication* validates again, merges preserved fields (execution results, license, provenance), fingerprints, checks duplicates, writes a validation report, uploads the project, and registers it; approved projects are ingested into the core tree and packed into the chunk layout below. Programs converted by coding agents (Articraft, the agent-written collections) and programs generated by the harness pass through the same validation, execution, and rendering stages before packaging.

B.4 Record format and archive layout

A sample. A sample is a folder named by its key (Figure 16). It contains the entry program in its own language, a metadata file, a caption file, a folder of multi-view stills and, where available, a video clip and a mesh export, and any other files the program needs: the full multi-pass shader source,

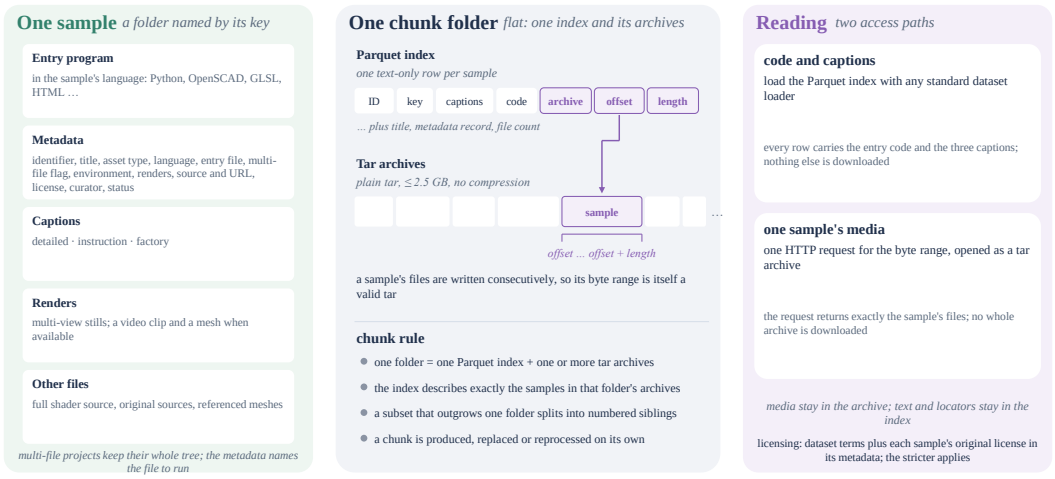

Figure 16: **Record format and archive layout.** Left: a sample is a folder holding the entry code, its metadata, three captions, the renders, and any other source files. Middle: a chunk folder pairs one text-only index with plain tar archives in which each sample's files are consecutive, so that the byte range recorded in the index is itself a valid archive. Right: the index alone serves code and captions, and one ranged request serves a sample's media.

the original program a conversion started from, referenced meshes, input textures. Multi-file projects keep their whole tree under the key, and the metadata names the file to run.

Metadata and captions. The metadata record carries the identifier (subset, optional group, and key), a title, the asset class (for example 3D object, articulated object, shader), the dialect and runtime (for example Blender Python, CadQuery, GLSL), the entry file, a multi-file flag, the declared execution environment (for example Blender 5.0, Shadertoy/WebGL2), the list of render paths, the source collection and its URL, the sample's original license, the curator, and a status; subsets add their own keys, such as the shader identifier, author, tags, and pass structure for Shadertoy or the conversion fidelity fields for Articraft. Status values record the curation state of the packaged record (curated or marked for revision in reviewed collections, and the harvesting state for collections packaged before review). The caption file holds three views of the same sample, each grounded differently: a *detailed* objective description of the rendered output, including how it animates; an *instruction*, phrased as a user request to generate the sample as code and naming the target language; and a *factory* recipe, the technique and algorithmic steps needed to reproduce the sample from the code.

Chunks. The corpus is stored in chunks. One folder holds one Parquet index and one or more numbered tar archives; the index describes exactly the samples in that folder's archives; folders are flat; and a subset that outgrows one folder splits into numbered sibling folders so that each index stays near 20 MB. A chunk is self-contained: it can be produced, replaced, or reprocessed on its own, and adding or removing one is a single atomic commit of the repository. Archives are plain tar files of at most 2.5 GB without compression, which is what makes byte-range access possible.

The index. Each row of the index carries the identifier, the key, the title, the three captions, the verbatim metadata record, the full source of the entry file, and a locator triple: the repository-relative path of the archive holding the sample, the byte offset of the sample within it, and the byte length, together with the number of files. The entry code is kept in the index on purpose, so that standard dataset loaders and SQL engines obtain code and captions without touching the archives; for multi-pass shaders and multi-file projects the entry goes into the index and the complete source is recovered losslessly from the archive. Because a sample's files are written consecutively, the byte range is itself a valid tar, and one ranged HTTP request retrieves exactly that sample's members.

Table 12: **Contents of the main subsets.**

Subset	Code	Renders and other files
Shadertoy	GLSL entry pass and the full multi-pass source	frame grids, a video clip, input textures where used
DeepCAD	CadQuery converted from the construction sequence	four views; fidelity fields in the metadata
Articraft (CadQuery)	self-contained CadQuery and the original helper-based program	GLB mesh, eight views; conversion fields
Articraft (URDF)	the URDF description in the archive	rendered poses
3DCodeBench factories	Blender Python	GLB mesh, four views
Bioinspired3D	Blender Python, geometry and textured variants	five views
Thingiverse	the original OpenSCAD project	five views
coding-agent subsets	Blender Python or a Three.js scene	four views and a mesh export (Blender)

Table 13: **Selected data resources for programmatic 3D generation.** Sizes use each source's unit and do not imply equivalent tasks or disjoint data. Text2CAD adds approximately 660 K language annotations to the DeepCAD lineage. The 3DCodeBench training corpus is distinct from its 212-task evaluation set. †Text–code pairs in the unified export, counting each caption style and representation of a program separately; the pairs derive from 269.6 K curated programs (Appendix B.1).

Resource	Representation	Content	Reported size
DeepCAD (Wu et al., 2021)	CAD sequences	parts	178,238 models
Text2CAD (Khan et al., 2024)	CAD sequences	parts	≈170 K models
Fusion 360 Gallery (Willis et al., 2021)	CAD sequences	parts	8,625 sequences
BlendNet (Du et al., 2024)	Blender Python	objects	8 K instruction–code pairs
Articraft-10K (Zhou et al., 2026)	Python SDK, URDF	articulated	>10 K assets
3DCodeBench (Gao et al., 2026)	Blender Python	objects	≈26 K code samples
3DCodeVerse-2M	**6 formats**	**4 tracks**	**2.03 M text–code pairs**†

Per-subset conventions. Table 12 lists the file conventions of the largest subsets. Harness-generated samples are packed by the exporter of Appendix A.11 into the same index and archive layout, with the export metadata of that appendix in place of the harvesting fields.

Snapshot caveats. Three packaging deviations of the current snapshot are known and are being repaired before the release is finalized: the Shadertoy chunks predate the repository-relative archive column and name their archives relative to their own folder while carrying the full multi-pass source in an additional column; the URDF samples keep their description in the archive rather than in the code column of the index; and a small number of locators (one zero-length record and part of one instance subset) must be repacked. Readers of older chunks need the column alias; the contract above describes the target layout.

B.5 Comparison with related resources

Table 13 compares selected resources using the units reported by their authors. Counting every caption style and representation of each program, 3DCodeVerse-2M provides 2.03 M text–code pairs derived from 269.6 K curated programs (Appendix B.1). The collection integrates multiple code formats with descriptions, execution information, and provenance; dataset sizes should be interpreted together with their representation, source lineage, and annotation variants. In the packaged catalogue, CadQuery and GLSL together account for 94.6% of records and Blender Python for 2.6%, a composition that motivates the data-mixture experiments of Appendix C.5: dialect coverage, not volume, is the scarce resource, and a marginal OpenSCAD, Three.js, or articulated program carries more information than a marginal CAD or shader program.

C Experimental Details and Additional Results

This part gives the training recipes and evaluation protocol behind §5 and collects the additional quantitative results that support it. §C.1 specifies every training run and preference set; §C.2 shows the training curves of three 27B runs over the full text export; §C.3 specifies the execution environment,

the geometry metrics, and the variance of repeated harness runs; §C.5 reports additional open-model results; and §C.6 reports additional harness studies and ablations.

C.1 Training details

Setup. All training uses an unmodified LLaMA-Factory (Zheng et al., 2024) on 4×H100 80 GB. LoRA (Hu et al., 2021) rank 64 (α 128, dropout 0.05, all linear targets) for supervised fine-tuning; rank 16 (α 32) on the merged SFT model for DPO (Rafailov et al., 2023). bf16, gradient checkpointing, sequence packing, per-device batch 1 with gradient accumulation 8 (4 for 27B), cosine schedule, warm-up ratio 0.05. SFT learning rate 1×10^{-4} (full fine-tuning: 1×10^{-5} under ZeRO-3 (Rajbhandari et al., 2020)); DPO learning rate 5×10^{-6}, $\beta = 0.1$, sigmoid loss, 2 epochs. Context length 8,192 for the 9B runs and 4,096 for 27B. 27B at 8,192 does not fit on an 80 GB card. Inference for evaluation uses vLLM (Kwon et al., 2023), greedy unless a sampled (T=0.7) number is marked as such.

Throughput. 9B LoRA on 4 GPUs sustains ≈17 K tokens/s (270 K pairs / 293 M tokens in 4 h 46 m); 27B LoRA under DDP sustains ≈6.2 K tokens/s at the step level. LoRA training of the 27B model under ZeRO-3 is **13× slower** than plain DDP (≈460 vs. ≈6.2 K tokens/s): the LoRA optimizer state is small, so ZeRO-3 adds parameter all-gather traffic without a corresponding memory saving. Enabling PyTorch's expandable memory segments *causes* OOM in the fused linear-attention Triton backward, which allocates outside PyTorch's caching allocator.

Data format and contamination control. Samples are ShareGPT-format chat pairs: a system prompt naming the dialect and its output contract, the caption as the user turn, and the program in a fenced block as the assistant turn. Samples whose full chat exceeds the context length are *dropped*, never truncated. The 212 3DCodeBench factories, all 3,904 re-seeded instances of these factories, and the per-dialect test splits are reserved for evaluation; no sample in the reported training mixtures shares an identifier with an evaluation sample. Table 14 gives the exact composition, token budget and wall-clock time of every mix.

Table 14: **Training mixes.** 39 logged training runs and 276 evaluations went into §5; this table gives the mixes that the reported models were trained on. *boot* = execution-verified programs written from scratch by the harness and admitted only after they run. *cap/sample* is the number of caption styles each program was expanded over.

Mix	Base	Pairs	Mtok	cap/sample	Composition	Wall
5.2 K Blender	9B	5,193	3.8	mixed	Bioinspired3D + non-test factories	30 m
25.6 K	9B	25,600	26	1	4 dialects, merged single-dialect sets	1 h46
102 K	9B	101,851	107	1	CadQ. 40 K, GLSL 40 K, OpenSCAD all, Blender 5 K + 15.7 K boot	3 h25
116 K *(bal)*	9B	116,000	110	1	as above, Blender share 15% → 34% by *repetition*	2 h26
270 K	9B	270,173	293	1	all CadQuery, all GLSL, Blender 5 K + 15.7 K boot, OpenSCAD ×4	4 h46
364 K	9B	364,000	304	**3**	all six sources, **no boot**	6 h43
273 K	9B	273,000	300	1	all six sources, **+ all boot**	4 h50
76 K	27B	75,910	56	≤ 2	CadQuery + GLSL = 74% of tokens, no boot	3 h51
104 K	27B	103,770	75	≤ 2	Blender share **40%**, all boot restored	3 h35

Preference set	Base model	Prompts × K	Pairs built	Per-dialect
exec-DPO round 1	9B, 5.2 K	15,069 × 2	2,590	Blender 100%
exec-DPO round 2	+ round 1	9,876 × 2	587	Blender 100%
exec-DPO	9B, 102 K	6,000 × 2	1,052	OpenSCAD 407, GLSL 562, Blender 55, CadQ. 28
exec-DPO	9B, 270 K	6,000 × 2	909	GLSL 566, OpenSCAD 250, Blender 70, CadQ. 23
exec-DPO	27B, 104 K	4,800 × 2	756	OpenSCAD 301, GLSL 409, CadQ. 26, Blender 20
geo-DPO	9B, 270 K	3,000 × 4 + 1,500 × 2	2,357	Blender 1,760, CadQ. 597

Geometry pairs require $F_{\text{best}} \geq 0.25$ and $F_{\text{best}} - F_{\text{worst}} \geq 0.15$ over at least two executing samples.
Appendix C.5 reports the results of preference optimization on these sets.

C.2 Training curves of the 27B data-scale runs

This section records the optimization behaviour of 3DCodeVerse-2M at the scale of the full text export. Three LoRA runs of Qwen3.8-27B share one recipe and differ only in which export rows (Appendix B.1, Table 9) they train on.

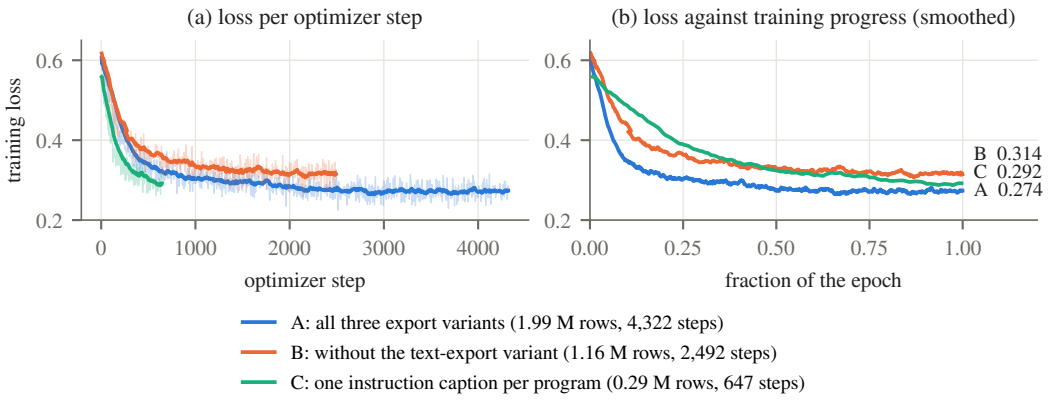

Figure 17: **Training loss of the three Qwen3.8-27B data-scale runs. (a)** Loss per optimizer step; faint lines are the logged values (every five steps), solid lines an exponential moving average with a time constant of 50 steps. **(b)** The smoothed curves against the fraction of the epoch, which aligns the learning-rate schedules of the three runs; the numbers are the smoothed final losses. Each loss is measured on the run's own training distribution, so the levels are not a comparison of model quality.

Recipe. LoRA rank 16 (α 32, dropout 0.05), learning rate 5×10^{-5} with a linear warm-up over the first ≈3% of steps and cosine decay to zero, one epoch. Samples are packed into sequences (≈7.2 samples per sequence) and each optimizer step consumes about 64 packed sequences. The loss is logged every five steps.

Data arms. **A** trains on all three export variants (original format, text-export variant, flattened multi-pass shaders): 1,992,816 rows in 276,594 packed sequences, 4,322 steps. **B** removes the text-export variant, which re-templates programs that are already present in their original format: 1,160,630 rows, 159,427 sequences, 2,492 steps. **C** keeps one copy of each program with its instruction-style caption: 292,690 rows, 39,926 sequences, 647 steps.

Observations. Figure 17 shows the curves. All three runs converge smoothly: no logged value exceeds 1.3× the mean of the preceding 20 logged values, and none of the runs diverges. Most of the decrease happens early: the loss falls from about 0.6 to 0.30 (A) and 0.36 (B, C) within the first quarter of the epoch, and A changes by less than 0.01 over its second half (0.277 at mid-epoch, 0.274 at the end). The final smoothed losses are 0.274 (A), 0.314 (B) and 0.292 (C). A reaches the lowest loss, and at an equal number of steps it stays below B (0.275 vs. 0.316 at step 2,490). Because A sees most programs in more than one representation, part of this gap reflects repeated content rather than better generalization; C, with one copy per program, descends more slowly per fraction of the epoch than A but ends below B with a quarter of B's steps.

C.3 Evaluation protocol

Execution and geometry for open-model evaluation. Generated text is passed through a code extractor that strips reasoning blocks, collects every fenced block including an unterminated trailing one, and scores candidates by dialect markers, fence language, and syntactic validity. This matters for instruct models: with thinking enabled, 52 of 212 answers from the 27B base model contained no fence at all and 74 had their best block not last. It has no effect on fine-tuned models, whose output format is fixed by fine-tuning (no answer without a fence and none with several blocks). Programs then run in the pinned environment with a 300 s timeout. Meshes are exported to GLB, centred, scaled to unit maximum extent, sampled at 10,000 surface points, and compared by symmetric Chamfer-L2 and F-score at 0.05 and 0.1, taking the best of four yaw rotations. The reference scripts self-check at Chamfer ≈ 0.016 and F@0.05 ≈ 1.0.

Execution environment. Blender 5.0.1's bundled Python does not include SciPy, whereas **15 of the 212 reference scripts import it**. Programs that follow the reference style therefore failed for reasons unrelated to the model. Installing SciPy and re-executing raised the 27B v1 result from 49.5%

to 57.6% (+8.1 points) and left the other runs largely unchanged. All reported numbers use the corrected environment; execution-based benchmarks are sensitive to the runtime as well as to the model.

Object prompts and compared systems. The object suite contains 100 prompts that request Three.js programs; the object comparison uses the first 60 prompts in their original order. All systems receive the same prompt strings, including Claude Design, whose outputs are included in the release. The suite labels 21 prompts as simple, 17 as medium, and 62 as detailed; these labels describe prompt specificity rather than independently measured difficulty. The prompts specify no additional dimensions or required-part lists. Newly generated outputs receive the runtime and output-format instructions of their workflow, whereas the Claude Design programs are used unchanged and rebuilt with the common evaluator; the comparison therefore does not match their generation compute to that of the other workflows.

Gemini 3.8 Flash and Claude Opus 5.5 are compared with their baselines on these 60 objects. Claude Sonnet 5, GPT-6 Luna, and GPT-6 Astra are evaluated on a common 30-object subset with ten prompts from each specificity level. Each baseline, labelled by its native coding-agent backend in Table 23, generates code in a single pass without tools or feedback; 3DCodeVerse Harness adds planning, tools, and automatic repair around the same model. The comparison thus measures the gain of the complete workflow for a fixed model, including its auxiliary model calls; token usage and estimated cost quantify the additional computation. For Gemini 3.8 Flash, the baseline uses a direct API call and 3DCodeVerse Harness uses the Gemini CLI backend. Claude Opus 5.5 and Claude Sonnet 5 baselines request high reasoning effort, and GPT-6 Astra uses medium effort in both settings.

Single user request and internal refinement. Single-request generation constrains human interaction: the system receives one prompt and no subsequent human corrections or candidate selection. It permits multiple internal model calls, tool invocations, and automatic repairs. The object comparison sets $T = 0$ in the notation of §4, retaining the initial generation, its internal tool use and execution repair, and its final automatic judgment. This corresponds to one harness round without additional refinement rounds. A separate study with $T = 4$ adds up to four refinement rounds and automatic candidate selection, still under a single user request. In the Gemini 3.8 Flash object runs, the planner is Gemini 3.8 Flash and the in-loop judge is Gemini 3.1 Pro.

Evaluation and reference scores. All object runs use the same harness version. The evaluator rebuilds each delivered program, extracts geometric measurements and connectivity checks, and renders canonical multi-view and clay images. Gemini 3.1 Pro scores each artifact against the original prompt three times, with the same object rubric for all systems. Across the four systems evaluated on all 60 prompts, 233 artifacts receive three valid judgments, three receive two, and four generations fail to build. Scores average the available judgments. The evaluator does not use the plan generated by the harness as an acceptance checklist. Although the in-loop and final judges share a model and rubric family, the final scores are computed independently from the delivered artifacts; human preference provides additional validation.

The seven object criteria are intent fidelity, structural plausibility, geometry detail, proportions and scale, assembly fit, materials, and craftsmanship, with weights $(0.22, 0.18, 0.16, 0.12, 0.12, 0.10, 0.10)$. The overall score S_{all}, reported as the Judge Score in Table 23, includes the evaluator's defect penalties and score caps. For the comparison with Claude Design, we also report a visual-only score:

$$S_{\text{vis}} = \frac{\sum_{k \neq \text{scale}} w_k s_k}{\sum_{k \neq \text{scale}} w_k}, \tag{7}$$

where s_k is the mean criterion score and the excluded criterion is proportions and scale. This supplementary score appears only in Tables 15 and 17; the main comparison uses the full Judge Score. It is computed before aggregate defect penalties and caps, because the object prompts do not specify metric units. It is not an image-only re-evaluation: the criterion scores are assigned with geometric evidence, and excluding the scale criterion does not remove every influence of that evidence. Both scores serve as reference measurements, and failed generations contribute zero to both.

SigLIP-2 text–render similarity. We use SigLIP-2 (So400m, patch size 16, NaFlex) at a fixed checkpoint revision to encode the original prompt and four canonical shaded views of each object.

The views are the elevated front-right, back-right, back-left, and front-left views; contact sheets and clay views are excluded. The official processor lowercases the text with a 64-token limit and resizes images to at most 256 patches of size 16×16. We use the original prompt without a classification template, normalize both embeddings to unit length, and average their cosine similarities over views:

$$S_{\mathrm{SigLIP}}(c, f) = \frac{1}{4} \sum_{v=1}^{4} \frac{E_{\mathrm{text}}(c)^{\top} E_{\mathrm{image}}(I_v(f))}{\|E_{\mathrm{text}}(c)\|_2 \, \|E_{\mathrm{image}}(I_v(f))\|_2}. \tag{8}$$

Table 23 averages these object-level scores over the same 60 prompts used for the Judge Scores and assigns zero to failed generations. We report raw cosine similarity without the learned logit scale, bias, or sigmoid. The encoder runs in evaluation mode with float32 inference on the same renders used for judging. None of the 100 object prompts exceeds the text-token limit. The Gemini 3.8 Flash baseline has four generation failures on the 60 prompts; these receive zero, and its mean over the 56 successful generations is 0.1453.

Pairwise preference protocol. Human and VLM comparisons use the original prompt and anonymized multi-view renderings with matched cameras, framing, and presentation size. The primary judgment is overall preference, informed by prompt adherence, structural plausibility, and visual quality. For W wins, L losses, and T_{tie} ties, the reported preference is $(W + T_{\mathrm{tie}}/2)/(W + L + T_{\mathrm{tie}})$. VLM comparisons present both candidate orders; conflicting orders count as a tie and are reported separately from explicit ties and judge failures. Human comparisons randomize the presentation order and use a common prompt subset, fixed in advance, for all compared systems. Prompt-level bootstrap intervals treat repeated views and ratings as observations of the same task.

A failed generation counts as an outcome: a valid artifact beats a missing one, two missing artifacts tie, and each system's execution rate is reported alongside. A second estimate restricted to pairs of valid artifacts separates artifact quality from execution reliability. Provider outages and evaluator failures are counted separately and are neither losses nor ties. The W/L/T counts in Tables 15 and 17 are derived from score differences and are not preference votes.

Object and cross-task results. Tables 15–17 report score comparisons for objects, scenes, and graphics. The object evaluation covers 100 Claude Design outputs, 100 generations from each of the Gemini 3.8 Flash and Claude Opus 5.5 baselines, and 62 Gemini 3.8 Flash generations with 3DCodeVerse Harness; scores and resource summaries are computed on the first 60 prompts, which all four systems completed. This subset contains 52 detailed, four simple, and four medium prompts, so its specificity distribution differs from that of the full suite; the same prompts underlie the corresponding Gemini 3.8 Flash, Claude Opus 5.5, and Claude Design scores in Table 23. Score intervals are paired Student-t intervals over prompts. Graphics and scene results are reported separately because their rubrics and sample counts differ. The graphics evaluation additionally includes nine same-model VLM pairwise judgments with both presentation orders (Table 18).

Table 15: **Object scores.** All scores and differences use a 0–100 scale. W/L/T counts the signs of per-prompt score differences and is not a preference vote.

A vs. B	n	S^A_{all}	S^B_{all}	ΔS_{all}	**95% CI**	**W/L/T**
Flash + 3DCodeVerse vs. Flash baseline	60	86.6	61.3	+25.3	[19.0, 31.6]	56/4/0
Flash + 3DCodeVerse vs. Claude Design	60	86.6	37.4	+49.2	[43.6, 54.9]	59/1/0
Opus baseline vs. Claude Design	60	77.2	37.4	+39.9	[34.5, 45.2]	59/1/0
Flash baseline vs. Claude Design	60	61.3	37.4	+23.9	[17.3, 30.5]	51/9/0
A vs. B	n	S^A_{vis}	S^B_{vis}	ΔS_{vis}	**95% CI**	**W/L/T***
Flash + 3DCodeVerse vs. Flash baseline	60	88.1	69.3	+18.8	[13.2, 24.5]	54/6/0
Flash + 3DCodeVerse vs. Claude Design	60	88.1	62.4	+25.6	[22.2, 29.1]	58/2/0
Opus baseline vs. Claude Design	60	84.5	62.4	+22.0	[19.1, 25.0]	60/0/0
Flash baseline vs. Claude Design	60	69.3	62.4	+6.8	[1.4, 12.2]	45/15/0

*Lower-panel W/L/T uses S_{vis} differences. Flash denotes Gemini 3.8 Flash and Opus denotes Claude Opus 5.5; a baseline generates code in a single pass without tools or feedback (Appendix C.3).

Table 16: **Object execution rate and resource usage.** Usage is averaged over prompts, including generation failures, and excludes final evaluation. Resource budgets follow the generation workflow of each system.

System	n	Exec. (%)	Input (k)	Cached (k)	Output (k)	Est. USD
Claude Design	60	100.0	–	–	–	–
Flash baseline	60	93.3	0.3	0.0	9.4	0.035
Opus baseline	60	100.0	3.0	0.5	28.5	0.353
Flash + 3DCodeVerse	60	100.0	20,816.0	19,551.8	234.9	3.312

Cached tokens are a subset of input tokens; output includes reported reasoning tokens. Claude Design programs require no generation calls in our evaluator, so their original generation usage is not measured.

Table 17: **Scene and graphics scores.** Differences compare 3DCodeVerse with the baseline of the same backbone, in points on a 0–100 scale. W/L/T is derived from score differences; intervals use paired prompts, and n denotes completed pairs. Flash, Sonnet, Luna, and Astra denote Gemini 3.8 Flash, Claude Sonnet 5, GPT-6 Luna, and GPT-6 Astra.

Track	Backbone	n	ΔS_{all}	95% CI	W/L/T	ΔS_{vis}
Graphics	Flash	11	$+27.2$	$[9.0, 45.3]$	10/1/0	$+19.1$
Graphics	Sonnet	11	-5.4	$[-20.8, 10.1]$	6/5/0	-4.3
Graphics	Luna	11	$+6.8$	$[-1.2, 14.8]$	8/3/0	$+6.5$
Graphics	Astra	11	$+30.4$	$[8.6, 52.2]$	10/1/0	$+28.2$
Scenes	Flash	10	$+19.4$	$[0.2, 38.6]$	9/1/0	$+15.4$
Scenes	Sonnet	10	$+0.1$	$[-31.1, 31.3]$	3/6/1	$+2.8$
Scenes	Luna	7	$+14.6$	$[2.5, 26.8]$	7/0/0	$+9.1$

For graphics, S_{vis} retains all criterion weights before aggregate penalties and caps. For scenes, it additionally excludes the combined spatial-coherence-and-scale criterion.

Table 18: **VLM pairwise preferences on graphics.** Only judgments with both presentation orders are included; each row compares 3DCodeVerse with the baseline of the same model. Coverage is limited to two or three prompts per model.

Model	Judged pairs	W/L/T	Baseline (%)	3DCodeVerse (%)
Gemini 3.8 Flash	3	2/1/0	33.3	66.7
Claude Sonnet 5	2	1/0/1	25.0	75.0
GPT-6 Luna	2	1/0/1	25.0	75.0
GPT-6 Astra	2	1/1/0	50.0	50.0

W/L/T is reported for 3DCodeVerse; ties include disagreements between presentation orders and count as half a win for each system. These are direct pairwise judgments, unlike the score-derived W/L/T counts in the preceding tables.

Generation tokens and cost. Table 16 reports input, cached-input, and output tokens per generation attempt with a completed outcome. Input tokens include cached tokens, which are shown separately and are not counted twice; output tokens include reasoning tokens. Harness usage covers the planner, coding agents, repairs, and in-loop judgments, and excludes final evaluation calls. Costs are estimated from provider prices at the time of generation, with CLI usage valued at API-equivalent rates rather than subscription charges; they describe the measured workflows and do not imply matched compute across systems.

The Claude Design programs involve no generation calls in our evaluator, so their generation tokens and cost are reported as unavailable. The token log accompanying these programs records only the prompt and the final code, without per-call accounting of input, cached, output, and auxiliary calls, and is therefore not comparable.

Judge noise and run-to-run variation. The additional harness studies in Appendix C.6 use an evaluation pipeline that rebuilds, renders, and judges the generated code under matched conditions, with two judge samples per scored output and a configuration independent of the in-loop feedback. Repeated judgments of fixed artifacts give standard deviations of 0.030 for a Gemini 3.1 Pro judge

and 0.083 for a Gemini 3.8 Flash judge on a 0–1 scale. These values capture judge noise only, not the variation between independently generated artifacts.

Table 19: **Variation between repeated harness runs.** Two runs use the same prompts, generator, judge, and budget. Left: paired statistics. Right: the pair with the largest regression, whose generated plans differ. These runs stop once an output passes, whereas 3DCodeVerse Harness otherwise runs a fixed number of rounds.

A/A statistic	
scored pairs	6
mean paired Δ	$+0.043$
median paired Δ	-0.008
sd of paired Δ	**0.202**
SE of paired Δ	0.082
separated at 2 SE?	✗
regressions ($\Delta < -0.03$)	2 of 6
sign test p	1.00
judge σ (Pro, 3 re-judgements)	0.030
pairs needed for ± 0.02	**408**

worst pair	run A	run B
planned parts	**1**	**10**
rounds run	1	2
triangles	4,796	61,340
tool calls	28	168
generation cost	\$0.33	\$1.61
stop reason	passed	budget
fixed-judge score	0.906	0.700
$\Rightarrow \Delta = -0.206$ between identical runs		

An A/A experiment, in which two runs use identical settings, yields a paired standard deviation of 0.202 across six scored pairs (Table 19). In one pair, the two runs plan one and ten parts, respectively, and differ substantially in mesh size, tool calls, and cost. Planning is therefore one source of variation in this example, although the experiment does not attribute all run-to-run variation to the planner. At the observed spread, approximately 400 paired prompts are required for a two-standard-error interval with a half-width of 0.02. Studies with a few dozen prompts cannot resolve effects of that size.

A second A/A experiment on articulated objects holds the plan fixed and obtains 12 scored pairs with a mean difference of -0.046, a paired standard deviation of 0.225, and a two-standard-error interval of ± 0.130. Substantial variation remains when planning is controlled, indicating that the downstream generation sessions also vary. Because the two A/A experiments use different prompt populations, their variances do not provide a quantitative decomposition of planner and generator effects. Appendix C.6 complements artifact-level scores with measurements of specific failure events.

C.4 The evaluation judge

All Judge Scores and VLM Preferences reported in §5 come from one fixed evaluator that is independent of the system under evaluation. The evaluator reuses the automated judgment of 3DCodeVerse Harness (Appendix A.5) with a fixed model, rubric, and evidence, so that systems with and without the harness are scored by an identical procedure.

Evidence. Each delivered program is rebuilt in the pinned runtime; a program that fails to build receives a score of zero. For 3D objects, the evaluator measures the exported mesh, runs the connectivity check in the authoring frame, and renders fourteen shaded views together with clay views. Articulated objects additionally receive the joint sweep and its pose renders; scenes are rendered from their authored cameras and an orbit rig at two animation times and pass the frame checks; graphics programs are evaluated on their sampled frame sheet.

Judge input. The judge input is assembled by the same payload builder as that of the in-loop judge for the first round, with two differences: no generated plan or previous verdict is supplied, and the acceptance checklist consists of the must-have items of the original prompt. All systems are therefore judged against the same checklist, and the judge never receives code, plans, or the generator's reasoning.

Rubric and score. Gemini 3.1 Pro judges each artifact three times with the rubric of its task. For 3D objects, the rubric scores seven criteria: intent fidelity (weight 0.22), structural plausibility (0.18), geometry detail (0.16), proportions and scale (0.12), assembly fit (0.12), materials (0.10), and craftsmanship (0.10), with floors of 0.30, 0.25, and 0.30 on intent fidelity, structural plausibility, and

craftsmanship, respectively. A binary checklist of eleven defects, including a wrong object, missing named parts, floating parts, interpenetration, and inverted faces, subtracts penalties and imposes caps, and measured connectivity errors cap the score without a vote (Eq. 5). Criterion scores are averaged over the three samples and defects are decided by majority vote; the Judge Score is the resulting overall score, scaled to 0–100.

Pairwise preference. VLM Preference uses the same model on anonymized multi-view renders of two outputs for the same prompt, presented in both orders; a winner is declared only when both orders agree, and conflicting orders count as a tie (Appendix C.3).

Relation to the in-loop judge. Inside the harness, the judge that guides refinement follows the same design and, in the object runs, uses the same model. The evaluator nevertheless scores the delivered artifacts independently from a rebuild, without the plan or any in-loop verdict; repeated judgments of fixed artifacts vary with a standard deviation of 0.030 on a 0–1 scale. Because the in-loop and final judges share a model and rubric family, we additionally report human preferences, which agree with the Judge Score (§5).

C.5 Additional open-model results

This section reports the complete fine-tuning and preference-optimization results for the open models of §5.2, which assess 3DCodeVerse-2M and the accompanying training recipes.

Supervised fine-tuning. Table 20 reports per-dialect execution rates and F@0.05 for every fine-tuned model, including smaller and differently composed training mixtures; all rows use the decoding settings and test sets of Table 2.

The corpus turns a model that cannot write Blender at all into one that matches the reference code's own execution rate. Qwen3.5-9B scores **0/212** zero-shot: not one generated script executes. An error analysis explains what the corpus supplies: 165 of the 212 scripts open with a scene-clearing call that does not exist in the Blender API, and patching only that line lifts it to 4/212; the rest is Blender-5.0 API drift. Trained on 270 K pairs it reaches **84.9%** (T=0.7, as in Table 2); on the held-out Blender suite the 5.2 K and 102 K models reach **93.2%**, exactly the pass rate of the ground-truth code itself (96/103), and the 270 K model 90.3%. The 27B model starts far higher (51.9%) and ends higher still (**92.9%**), with the best 3DCodeBench fidelity of any model we train.

Scale and fine-tuning repair different deficits: fine-tuning mainly improves executability, whereas a larger base model improves geometry. Across the three 9B mixtures (5.2 K to 270 K pairs), conditional geometry stays in a narrow band: F@0.05 over executed tasks is 0.354–0.391. Scaling data changes *whether the program runs* more than *how much the result looks like the target*; the 27B models reach 0.397–0.424, and the untuned 27B model already scores 0.420 on the programs it can execute.

Preference optimization. We sample several completions per prompt, construct preference pairs from their execution outcomes or geometric scores, and apply DPO (Rafailov et al., 2023). Execution feedback pairs a program that runs with one that fails for the same prompt; geometry feedback pairs the programs with the highest and lowest F@0.05. Table 21 reports all preference-optimization runs, and Table 22 reports results across four decoding seeds.

For the 9B model trained on 5.2 K Blender pairs, execution-feedback DPO raises greedy 3DCodeBench execution from 75.0–82.1% to 93.9% after one round and to 96.2% after two. The gain depends on the starting checkpoint: for the 27B model trained on 104 K pairs, execution decreases from 93.4% to 91.5%, a change smaller than the seed-to-seed variation at T=0.7 in Table 22. Geometry-feedback DPO on the 9B model trained on 270 K pairs raises F@0.05 over executed programs from 0.387 to 0.403, while execution decreases from 90.1% to 88.2%. Table 14 lists the training configurations and preference-pair counts.

Termination feedback resolves non-terminating outputs. Under greedy decoding, fine-tuned models often repeat lines until the length limit on dialects whose programs end with a closing brace: a 9B model trained on 310 K pairs executes only 8.0% of OpenSCAD programs, whose outputs average 30,163 tokens. We pair each non-terminating greedy output with the model's own shortest

Table 20: **All fine-tuning mixtures.** Full results behind Table 2, including every training mixture and per-dialect F@0.05. All models are decoded at T=0.7 with one seed and scored on the same fixed test sets. *Exec*: programs that run in the dialect's runtime (%); for GLSL, shaders that compile. *F*: F-score at τ=0.05 between the generated and reference meshes, with failed programs scored 0. *DINO*: DINOv2-L cosine similarity between four-view renders of the generated and reference meshes (chance 0.159). Best model result per column in bold.

		3DCodeBench (212)			Blender (103)		CadQuery (200)		OpenSCAD (50)		GLSL (200)	Three.js (40)
Model	**Training data**	Exec	F	DINO	Exec	F	Exec	F	Exec	F	Exec	Exec
Zero-shot												
Qwen3.5-9B	—	0.0	.000	.000	1.9	.008	10.5	.030	50.0	.261	33.0	72.5
Qwen3.8-27B	—	51.9	.214	.211	74.8	.439	65.5	.159	80.0	**.415**	57.0	72.5
Qwen3.5-9B + LoRA SFT												
	5.2 K, Blender	73.1	.257	.228	**93.2**	.694	31.5	.081	32.0	.099	23.0	47.5
	102 K, 4 dialects	84.4	.328	.269	**93.2**	.725	99.0	.268	70.0	.276	61.0	0.0
	270 K, 4 dialects	84.9	.317	.266	90.3	.693	98.5	**.308**	86.0	.329	56.0	32.5
Qwen3.8-27B + LoRA SFT												
	76 K, CadQ./GLSL-heavy	53.3	.224	.222	92.2	.795	**99.5**	.227	76.0	.349	63.0	87.5
	104 K, Blender 40%	**92.9**	.365	**.303**	91.3	.786	99.0	.226	72.0	.298	**73.0**	**92.5**
	104 K + exec-DPO	**92.9**	**.374**	.298	**93.2**	**.798**	98.5	.234	**90.0**	.314	71.5	**92.5**

Table 21: **Preference optimization from the executor, all runs.** *Exec-DPO* pairs a program that runs against one that does not, for the same prompt; *geo-DPO* pairs the highest-F against the lowest-F program. *Termination* pairs the model's own non-terminating greedy output against its own shortest terminating sample for the same training prompt. *pairs* counts how many preference pairs the base model's own errors made available, the quantity that determines the gain. All runs here use greedy decoding, so the *before* column differs from the T=0.7 numbers of Table 2. Best-of-N: Table 22.

		3DCodeBench exec		3DCodeBench geometry		Blender held-out	
Base → +DPO	**pairs**	before	after	F^{ok}	CD↓	F^{ok}	CD↓
execution feedback							
9B, 5.2 K base → round 1	2,590	75.0–82.1	**93.9**	0.384	0.201	—	—
→ round 2	587	93.9	**96.2**	0.383	0.200	—	—
9B, 102 K base	1,052	84.9	87.3	0.375	0.202	0.783	0.077
9B, 270 K base	909	90.1	89.6	0.388	0.204	0.778	0.078
27B, 104 K base	756	**93.4**	91.5	0.395	0.195	0.872	0.052
geometry feedback							
9B, 270 K base	2,357	90.1	88.2	**0.403**	**0.195**	**0.800**	**0.070**
termination feedback							
9B, 310 K base → round 1	1,779	79.2	85.9	0.373	0.206	0.885	0.054
→ round 2	478	85.9	84.9	0.369	0.205	0.876	0.055

terminating sample for the same *training* prompt and apply DPO (Table 21, termination feedback). One round (1,779 pairs) raises greedy execution from 8.0% to 74.0% on OpenSCAD and from 59.0% to 79.5% on GLSL; a second round, built from the remaining failures of the first-round model (478 pairs), reaches 86.0% and 89.5%, and mean OpenSCAD output length falls to 2,884 tokens. 3DCodeBench execution rises from 79.2% to 85.9% after the first round and to 84.9% after the second, while F@0.05 over executed programs remains nearly unchanged; the gain therefore comes from programs that now terminate rather than from different content.

Table 22: **Best-of-N sampling** at T=0.7, four seeds. Execution-feedback DPO lifts both the mean seed and the union: pass@4 reaches 99.1%.

best-of-N (T=0.7, N=4)	per-seed exec				pass@4
9B, 5.2 K base	76.4	66.5	64.6	75.0	91.0
+ exec-DPO	90.6	93.4	86.3	88.2	**99.1**
9B, 270 K + geo-DPO	88.2	83.5	82.1	86.8	98.1

Supplementary data-mixture observations. Increasing the Blender share of the training mixture from 15% to 34% by repetition (Table 14) raises the held-out Blender F@0.05 from 0.705 to 0.816, with failed programs scored zero, and greedy OpenSCAD execution from 4% to 56%, whereas 3DCodeBench execution decreases from 84.9% to 82.5%. Gains on an individual dialect suite therefore do not necessarily transfer to 3DCodeBench.

In a smaller mixture comparison, CadQuery execution reaches 98% with mixed-dialect training versus 73% with single-dialect training. Fine-tuning on 280 long Three.js programs written by coding agents reduces execution from 77.5% to 30%, and non-terminating outputs appear in this configuration. These observations document the sensitivity of the model baselines to their training mixture and output lengths; they do not establish a general training prescription.

C.6 Additional harness results

Numerical results for Figure 4. Table 23 lists the 3D-object values shown in Figure 4a, together with the VLM Preference and SigLIP-2 text–image similarity that the figure omits.

The following studies complement the object evaluation of §5.1 and Appendix C.3. Each study specifies its own generator, rubric, and stopping rule; scores are therefore comparable within a study but not across studies or with the Judge Score. Table 24 compares 3DCodeVerse Harness with one-shot generation across models on sets of hard and mixed prompts.

Refinement across tracks. On 12 articulated-object prompts, the mean score increases from 0.605 after the baseline round to 0.789 after refinement, and 58% of the outputs pass under the corresponding rubric. The four hard prompts improve by $+0.284$ on average. On six GLSL prompts, the mean score increases from 0.712 to 0.748, and five outputs pass. These within-run comparisons between the baseline and final rounds use small prompt sets and therefore do not isolate the contribution of any individual harness component. Table 25 reports the mean score, execution rate, and cost per run of each model on the hard-prompt set.

Cost accounting. Over 61 audited runs, model usage totals $89.10, or $1.46 per run and $2.48 per passing artifact (36 passing runs). These costs depend on the prompt population, provider prices, and generation settings of the audited runs. The harness records the cost of every model call together with the applicable price table, whereas run control relies on the wall-clock budget described in Appendix A.

Explicit planning of joint couplings. An ablation on ten linkage prompts compares a planner with an explicit coupling field against a variant without the field and its planning instruction. Both variants generate only the baseline round, and the coding contract permits URDF mimic declarations in both. The resulting URDFs declare couplings for 10/10 prompts with the coupling field and 3/10 without it; with the field, eight outputs have a single independent degree of freedom, and the remaining two retain additional degrees of freedom for wheel rotation or mirror tilt. The fixed-judge score difference over the nine scored pairs, however, is not separated from zero: $\Delta = -0.006$, with a 95% confidence interval of $[-0.193, +0.181]$. Joint-sweep failures also persist (5/10 versus 4/10). The results support making kinematic relationships explicit in the plan, whereas geometric self-intersection remains a generation problem that the coupling declaration alone does not resolve.

Measuring specific failure events. The A/A experiments in Appendix C.3 show substantial variation between artifacts generated independently under identical settings. None of the individual changes in Table 26 is separated from zero by its paired score estimate. For these changes, we additionally measure directly observable failure events, such as invalid plans and retried tool calls. These measurements characterize the targeted mechanism even when a small artifact-level study cannot establish an effect on quality.

Invalid plans. A common planner failure is a collapsed hierarchy in which joints reference undeclared links. Restarting from the request and the violated rule, instead of continuing the conversation that contains the invalid plan, reduces the failure rate from 13/277 (4.7%) to 2/276 (0.7%) among calls with recorded plan outcomes in a window of 560 calls (two-sided Fisher exact test, $p = 0.0067$). The cost per call remains similar ($0.0345 versus $0.0337). A follow-up comparison of three restart

Table 23: **3D object generation.** Numerical values of Figure 4a–c. Comparisons at a fixed backbone (upper) and against Claude Design (lower), on identical prompts. SigLIP-2 reports cosine similarity; Judge Score reports overall rubric quality (0–100).

Model	Harness	Human Pref. (%)↑	VLM (%)↑	SigLIP-2↑	Judge Score↑
Same Model with Different Harnesses					
Gemini 3.8 Flash	Gemini CLI	13%	15%	0.1356	61.3
Gemini 3.8 Flash	3DCodeVerse	87%	85%	0.1468	86.6
Claude Opus 5.5	Claude Code	21%	19%	0.1437	77.2
Claude Opus 5.5	3DCodeVerse	79%	81%	0.1490	88.5
GPT-6 Astra	Codex	31%	28%	0.1456	76.4
GPT-6 Astra	3DCodeVerse	69%	72%	0.1488	87.7
GPT-6 Luna	Codex	11%	16%	0.1293	64.4
GPT-6 Luna	3DCodeVerse	89%	84%	0.1341	85.9
System comparison: 3DCodeVerse vs. Claude Design					
Claude Opus 5.5	Claude Design	9%	13%	0.1438	37.4
Gemini 3.8 Flash	3DCodeVerse	91%	87%	0.1468	86.6
Claude Opus 5.5	Claude Design	10%	14%	0.1438	37.4
Claude Opus 5.5	3DCodeVerse	90%	86%	0.1493	87.7

Table 24: **3DCodeVerse Harness vs. one-shot generation across models.** Scores are assigned by a fixed judge with two samples per output. In both settings, the GPT-6 models use the Codex CLI backend and Gemini 3.8 Flash uses the Gemini CLI backend. Δ is the paired score difference in points, and W/L/T counts per-prompt wins, losses, and ties by score difference. The generation settings differ from those of the object evaluation in Figure 4a.

Model	Prompts	n	Δ	95% CI	W/L/T
GPT-6 Luna	hard	10	$+29.1$	$[14.1, 44.1]$	10/0/0
GPT-6 Terra	hard	11	$+19.1$	$[1.3, 36.8]$	8/2/1
GPT-6 Sol	hard	11	$+15.5$	$[3.1, 27.9]$	8/2/1
Gemini 3.8 Flash	hard	8	$+40.1$	$[20.9, 59.2]$	7/0/1
Gemini 3.8 Flash	mixed	8	$+52.3$	$[34.5, 70.1]$	8/0/0

Table 25: **Mean score, execution rate, and cost per run** on the hard-prompt set, with 3DCodeVerse Harness and with one-shot generation. Execution rate is the fraction of programs that build and export successfully. The GPT-6 models use the Codex CLI backend. The lower panel compares configurations on eight matched hard prompts under the same fixed judge; differences and confidence intervals are in points (scores $\times 100$).

	score		execution		$ / run	
Model	harness	1-shot	harness	1-shot	harness	1-shot
GPT-6 Sol	**0.758**	0.603	100%	100%	6.72	0.29
GPT-6 Terra	0.646	0.441	100%	92%	3.99	0.05
GPT-6 Luna	0.601	0.289	100%	75%	1.05	0.01
Gemini 3.8 Flash	0.625	0.146	100%	38–57%	2.41	0.04
Claude (Claude Code default model, one-shot only)	—	0.472	—	75%	—	1.17

Does the harness substitute for a strong model? Paired on the same eight hard prompts, same fixed judge.

Comparison	n	Δ	95% CI	W/L/T	sep.
Gemini 3.8 Flash with harness − GPT-6 Sol one-shot	8	-0.2	$-11.8, +11.4$	5/3/0	✗
Gemini 3.8 Flash with harness − GPT-6 Sol with harness	8	-16.9	$-22.9, -10.8$	0/8/0	✓

policies over 700 calls, including a narrower restart trigger, does not separate the policies on the overall failure rate. For dangling-reference failures, the rate is 3/140 without restarts and 0/276 with the narrower trigger ($p = 0.038$); because this class-specific comparison follows the overall-rate

Table 26: **Effects of individual harness changes.** Top: paired score differences; none is separated from zero by the reported band. Bottom: failure-event rates, token usage, and structural counts before and after each change. Significance tests are reported where available; the remaining comparisons are descriptive.

Change	Method	n	effect	band / p	sep.
Artifact quality: paired score difference Δ					
identical settings (A/A)	paired score	12	-0.046	± 0.130	✗
plan-time geometry re-ask	paired score	14	$+0.064$	± 0.25	✗
Gemini 3.1 Pro vs. 3.8 Flash planner	paired score	13	$+0.090$	± 0.23	✗
deterministic joint repairs	paired score	12	-0.039	± 0.105	✗
fewer agent turns	paired score	14	$+0.015$	± 0.126	✗
Failure events that end a run or a round					
degenerate-plan re-sampling	plan-stage loss rate	560	$4.7\% \rightarrow 0.7\%$	p=0.0067	✓
tool verdict $\neq$ MCP error	retried tool calls	3,677	$26.6\% \rightarrow 1.3\%$	—	—
same change	uncached tokens/req.	5,194	$38{,}521 \rightarrow 13{,}480$	-65%	—
same change	generator $ per round	142	$1.572 \rightarrow 0.950$ (med.)	-40%	—
coupled-mechanism support	mechanisms declaring it	10	$3/10 \rightarrow 10/10$	—	—

comparisons, it is exploratory. Other invalid plans, such as joints whose parent and child links coincide, still occur.

Tool verdicts and execution errors. Reporting a valid negative tool verdict as a tool error degrades agent behavior. When a call successfully measures a collision or returns a failed build, such an error causes retries and, in one backend's error path, converts image payloads to base64 text. Separating tool execution failures from returned verdicts and bounding payload sizes reduces retried calls from 26.6% to 1.3% in two runs on the same 14 prompts with matched generator, judge, and budget. Uncached prompt tokens per main-role request decrease from 38,521 to 13,480, and the median generator cost per round decreases from $1.572 to $0.950. These measurements reflect both changes jointly; they neither isolate the effect of each change nor establish a corresponding gain in artifact quality.

Conversation history and token usage. Telemetry from an in-process agent backend covers 1,063 sessions and 30,401 turns. In these sessions, input-token usage per turn grows approximately as $\text{input}(t) = 12{,}364 + 1{,}822t$, and total usage per session scales approximately as $n^{1.60}$ with the number of turns n. Recomputing the usage of transcripts truncated to half of their turns reduces tokens by 68% and estimated cost by 55%, since cached history is priced lower. This analysis re-accounts existing traces; it is not a generation experiment and does not show that quality is preserved with fewer turns.

C.7 Executable articulation and comparison with UWM

Executable code makes the specified articulation explicit. We examine whether a generated 3D program can support robot interaction while retaining a rigid-link and joint structure, and compare its rendered outputs with a frozen world-action model, UWM. This application uses one GPT-6 Astra-generated articulated asset, validated and exported with 3DCodeVerse tooling, rather than the fine-tuned models above. The asset follows a supplied microwave geometry and hinge specification; its MuJoCo simulation inherits the native physical parameters. Independent forward-kinematics, rigid-distance and hinge checks pass at five poses. We evaluate this asset and the native asset at two initial states under three controls—contact push, hold and retreat—giving 12 cases. Each case executes the same 16 future robot commands at 20 Hz and retains all three UWM seeds. Code replay uses complete geometry and state in the same simulator as the reference, and is therefore an oracle reference, not an equal-information predictor.

Code renders more often satisfy the fixed RGB geometric criterion. We apply the same orientation-aware door-and-handle edge criterion to code renders, UWM predictions, and a last-frame baseline. The criterion returns an angle candidate or abstains when feature support is insufficient or ambiguous. Its thresholds are fixed before the new predictions: on 24 held-out static renders at six angles across

Table 27: Outputs meeting the frozen RGB geometric criterion. The 12 cases comprise two asset domains, two initial states and three controls. UWM contributes three seeds per case; the other methods contribute one output each. These are criterion-acceptance counts, not physical correctness or action-success rates.

Method	Criterion met	No accepted estimate
Code replay (oracle)	12/12	0
Code replay through UWM VAE	12/12	0
Last frame	12/12	0
UWM	7/36	29

four asset/robot-pose blocks, it returns 20 candidates, all within 1° of the known angle. This validates a limited rendering test, not the interpretation of arbitrary UWM images. We also pass each code render through UWM's frozen VAE to control for reconstruction effects. Across all 72 outputs, code and code-through-VAE each meet the criterion in 12/12 outputs, compared with 7/36 UWM predictions (Table 27). The other 29 UWM outputs are unassessable under this criterion. Seeds are repeated predictions within a case, and the cases share one task and closely matched geometry, so we report descriptive counts rather than independent-sample significance.

Preserving structure is not the same as predicting the action result. The last-frame baseline also meets the criterion in 12/12 outputs despite predicting no motion. The comparison therefore supports an advantage under this particular RGB criterion, which persists through the VAE control, but does not establish more accurate action-conditioned motion. The code provides directly inspectable rigid-link transforms; the experiment does not show that the generator inferred the correct physical parameters. Moreover, angle measurements on generated images and UWM's exact physical output time remain unvalidated, and the inherited simulator masks disable door–housing self-collision. We consequently make no claim of superior joint-angle prediction, contact dynamics or general physical accuracy. This application demonstrates executable articulated structure and a controlled comparison of its visible geometry; establishing better physical prediction requires a validated shared outcome measurement and an independent reference.

Qualitative comparison. Figures 18 and 19 show the generated asset from two initial states under contact push, hold and retreat. Qualitatively, code replay retains consistent door and handle geometry across controls, including after reconstruction through UWM's VAE, whereas UWM predictions exhibit greater variation in these contours across seeds. These observations support the preservation of visible articulated structure, but do not establish superior physical prediction: code replay uses complete scene and state information, and UWM's exact output-time alignment remains unvalidated. The last-frame baseline also preserves visible structure despite predicting no motion.

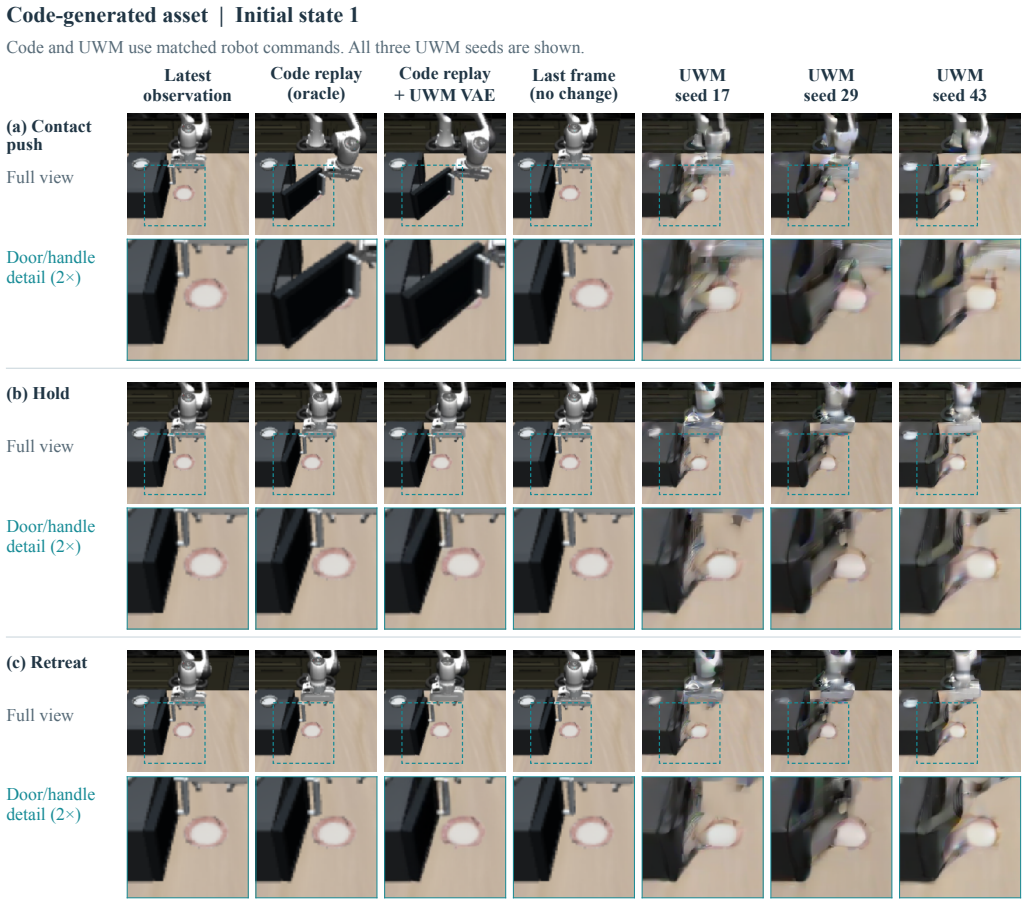

Figure 18: **Code replay and UWM on the generated asset: initial state 1.** Rows show contact push, hold and retreat. Columns show the latest observation, code replay (oracle), its reconstruction through UWM's frozen VAE, the last-frame baseline, and all three UWM seeds. Dashed boxes identify the same fixed image-space crop, enlarged 2× below each full view without image enhancement or retouching. Code and UWM receive matched robot commands; the last-frame baseline predicts no change. Code renders follow 16 commands, while UWM panels use the preselected decoded temporal index 0; exact physical-time alignment remains unvalidated.

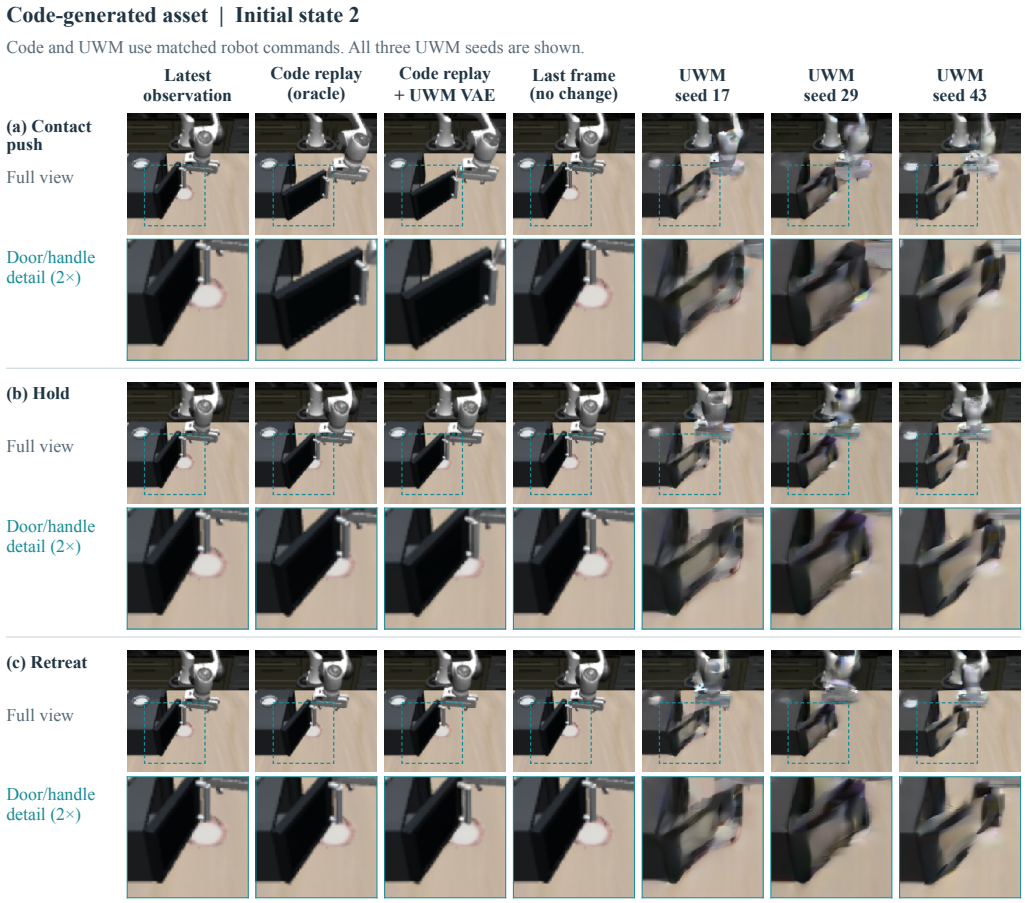

Figure 19: **Code replay and UWM on the generated asset: initial state 2.** The controls, column order and fixed 2× crops match Figure 18. All nine UWM predictions for this initial state are shown; together, the two figures retain all 18 UWM predictions for the generated asset. The VAE column controls for reconstruction of code renders, and the last-frame column preserves the observation without predicting motion. Code replay is an oracle reference with complete scene and state information. Visible contour differences are not measurements of joint-angle error or identified physical violations.

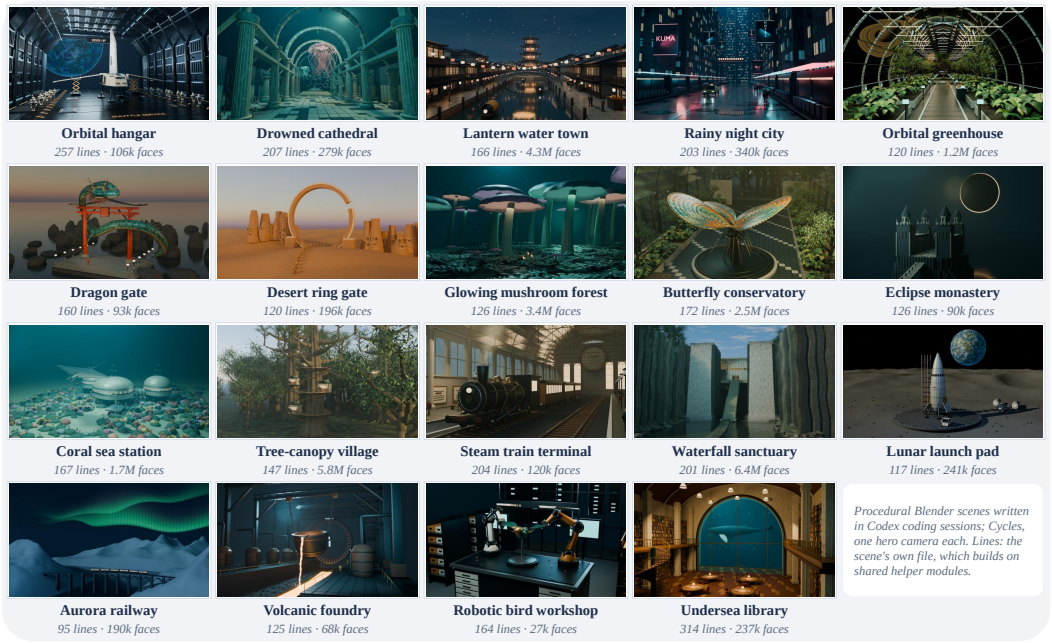

Figure 20: **Procedural scenes written by Codex.** Nineteen Blender scenes not shown in Figure 1, each rendered from its hero camera. Below each render: the length of the scene's own program file, which builds on shared helper modules, and the triangle-mesh face count of the rendered scene.

D Additional Visualizations

This part adds qualitative material to the quantitative results: a gallery of procedural scenes written by a coding agent (§D.1), harness outputs across the four tracks (§D.2), one sample from each corpus collection with its instruction caption (§D.3), and the evidence the judge receives in one round of each track (§D.4).

D.1 Procedural scenes

Figure 20 collects further procedural Blender scenes in the style of those in Figure 1, which it does not repeat. Each was written and revised by Codex in an interactive coding session and rendered with Cycles from one authored camera; the scenes use only procedural geometry, node materials and lights, without image textures, backplates or purchased meshes. They illustrate the scale of program the scene track targets, not a controlled comparison.

D.2 Harness outputs across tracks

Figure 21a extends Figure 5a with further prompts from the paired evaluation of §5.1: the same backbone, Gemini 3.8 Flash, generates once through Gemini CLI and once inside the 3DCodeVerse Harness, and both outputs are scored by the same judge. On the shader and scene prompts several one-shot outputs render an empty or uniform frame, which the judge scores zero. Figure 21b shows the first round and the selected round of four recorded multi-round runs, one per track; the articulated object is shown at the upper limit of its lid joint.

D.3 Corpus samples

Figure 22 shows one record from each collection of 3DCodeVerse-2M, rendered from its program, together with the record's instruction caption. Two cells are not the corpus's own renders: the Articraft CadQuery record is rendered here from its program, and the Three.js repositories carry no captions, so that cell shows a frame from the repository itself. The two harness cells show harness outputs of the kind these collections contain.

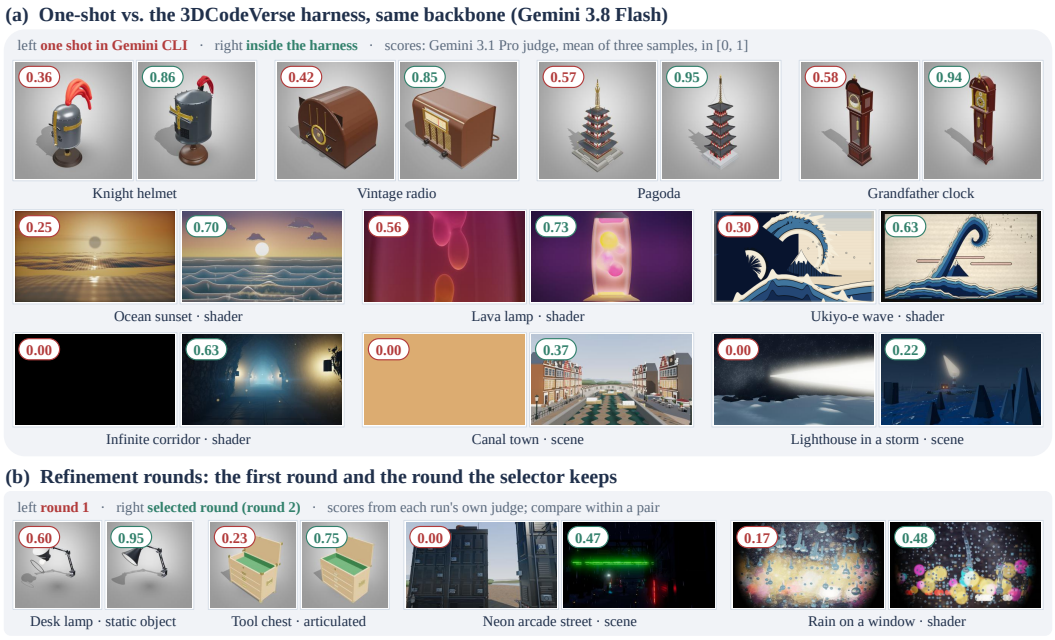

Figure 21: **Harness outputs across tracks.** **(a)** One-shot generation in Gemini CLI (left of each pair) versus the 3DCodeVerse Harness (right) with the same backbone, Gemini 3.8 Flash, on objects, shaders and scenes; the numbers are Gemini 3.1 Pro judge scores in $[0, 1]$ (mean of three samples). **(b)** Round 1 versus the round kept by the selector in four multi-round runs (static object, articulated object, scene, shader); the numbers are each run's own judge scores and are comparable only within a pair.

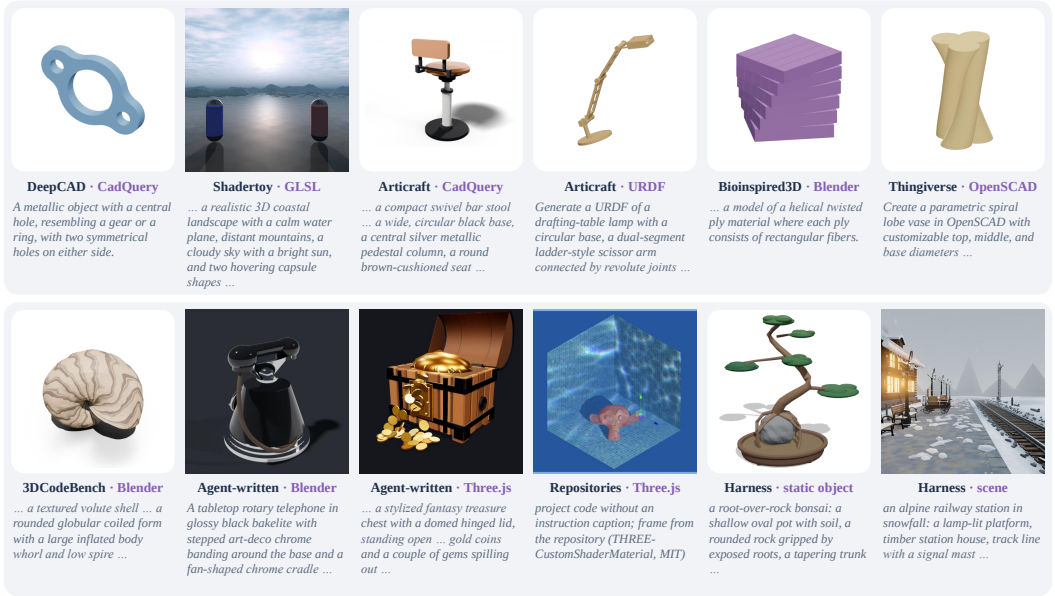

Figure 22: **Corpus samples.** One sample per collection (source · language) with its instruction caption; "…" marks where a caption is shortened.

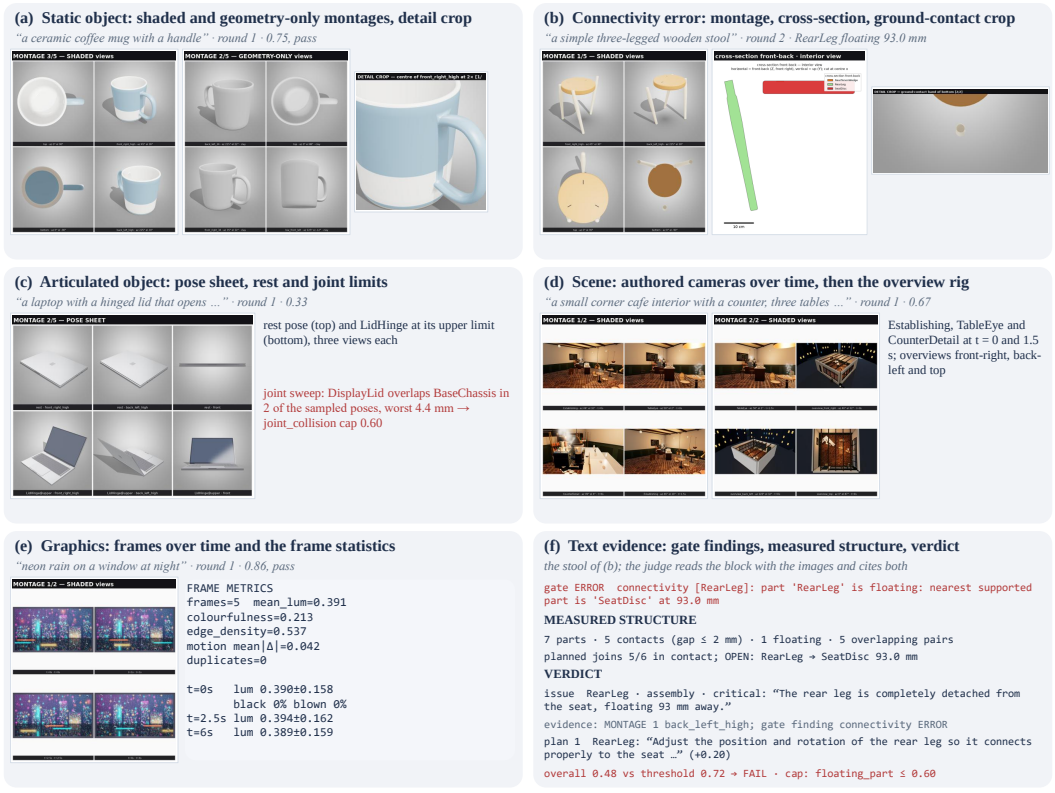

Figure 23: **Judge evidence.** **(a)** Shaded and geometry-only montages and a detail crop for a static object. **(b)** A connectivity error: montage, cross-section through the floating leg, and the ground-contact crop. **(c)** The pose sheet of an articulated object at rest and at the joint limit, with the joint-sweep finding. **(d)** A scene's authored cameras at two times and the overview rig. **(e)** Graphics frames over time with the frame statistics. **(f)** The text evidence of the round in (b): gate finding, measured structure, and the tail of the verdict.

D.4 Evidence the judge receives

Figure 23 reproduces the images and text the judge received in one round of each track, as described in Appendix A.5. The images are the labelled montages and crops exactly as sent; panels (b) and (f) belong to the same round, in which the connectivity gate measured a floating leg, and the judge cites both the montage and the gate finding.

E Reproducibility, provenance, and AI use

The metadata of each sample records its source collection, its source URL where available, and its license information. The release includes the conversion utilities and, for each data chunk, an index with byte-range locators from which individual samples can be retrieved without unpacking the archives. Missing provenance or verification results are marked explicitly and remain distinguishable from completed checks. The dataset is released under CC-BY-NC-SA-4.0, with per-sample original terms taking precedence where they are stricter. Each harness record stores the role and budget settings, hashes of the prompts, rubric, and pricing table, the harness version, and the detected runtime versions. A per-round log links each completed program revision to its source snapshot, check results, renders, and judgment. Not every run logs every field; a field that a run does not log is omitted from its record.

Generative AI is used as a *data-production instrument* in this work, and its use is recorded per sample rather than only disclosed in prose: coding agents transcribe assets into runnable code, write programs from scratch to a prompt spec, and generate the three caption styles. In harness-generated samples,

one metadata field identifies the source collection, and separate fields identify the models acting as generator, planner, and judge.