AGENTIC 3D CREATION VIA JOINT AGENT-PROGRAM DESIGN

Anonymous authors
Paper under double-blind review

ABSTRACT

Programmatic representations provide a compelling paradigm for 3D content creation, enabling fine-grained edits, interpretability, and explicit structural control. Yet, agentic workflows that rely on large language models (LLMs) to author 3D programs remain brittle, often failing to translate high-level intent into consistent low-level geometry. We attribute this fragility to a mismatch between existing programmatic interfaces and the reasoning strengths of LLMs, which favor semantic structure and spatial relations over fragile numeric choices. In this paper, we jointly design an Agent-centric Domain-Specific Language (ADSL) and a role-specialized multi-agent system to close this gap. ADSL bridges semantic logic and geometric constraints by emphasizing composability and spatial reasoning; it enables agents to manipulate geometry through relational operators instead of brittle absolute coordinates. Building on ADSL, our training-free multi-agent system follows a Plan–Execute–Critic loop to decompose requests, synthesize code, and iteratively repair errors and constraint violations using execution feedback. Experiments show that this co-design improves robustness, controllability, and faithfulness to user intent. Our method outperforms prior LLM-based baselines on text-to-shape and image-to-shape tasks while preserving explicit structure, editability, and interpretability. It also enables downstream applications such as articulated object creation and structured scene composition.

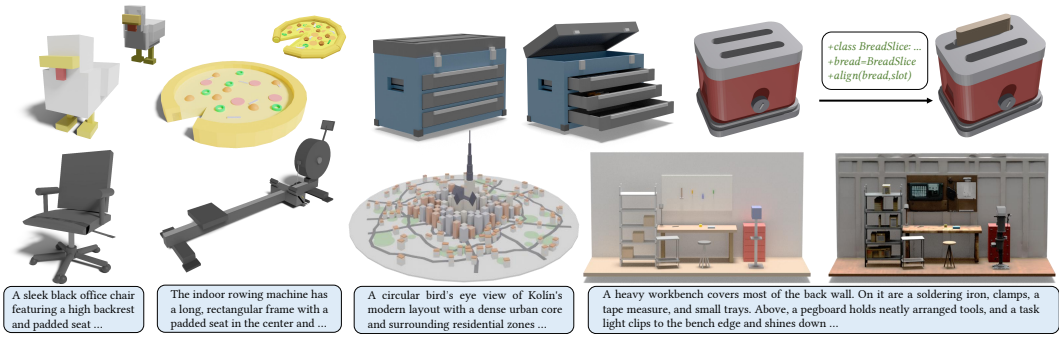

Figure 1: We present a jointly designed DSL and multi-agent system for robust text- and image-conditioned 3D modeling. The same structured representation also supports downstream applications including articulated asset modeling, shape editing and scene-level creation.

1 INTRODUCTION

3D content creation is a foundational problem in computer graphics, supporting applications spanning games, film production, and embodied AI. Recent advances in 3D generation have been largely driven by diffusion models trained on large-scale 3D datasets and diverse geometric representations (Xiong et al., 2024; Xiang et al., 2024; He et al., 2025b; Hunyuan3D, 2025). In parallel, an alternative paradigm represents 3D shapes and scenes as *programs* and leverages LLM-based agents to generate and manipulate them (Lu et al., 2025; Du et al., 2024; Zhang et al., 2025a; Hu et al., 2024; Zhang

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.

et al., 2025b). By explicitly encoding structure, parameters, and design constraints, programmatic representations enable an *agentic* workflow with better controllability, interpretability, and editability.

Despite this promise, agentic 3D creation remains unreliable: generated results often deviate from user intent and may degrade under iterative refinement. We argue that these failures stem from two coupled design axes: the *programmatic representation* (what code the agent writes) and the *agent system* (how the agent plans, generates, and improves code). Early pioneering efforts prompt LLM agents to emit general-purpose modeling programs, including Blender Python scripts, parametric CAD code, or procedural/rule-based programs (Yuan et al., 2024; Lu et al., 2025; Alrashedy et al., 2024; Du et al., 2024). Although Blender scripts and CAD code are highly expressive, they operate at a low level of abstraction, making it difficult to consistently encode semantic intent; they also tend to be brittle under localized revisions. In contrast, procedural and rule-based formalisms provide strong parametric control, but often struggle to scale to broad object diversity, stylistic variation, and fine-grained constraints (Raistrick et al., 2023; 2024). To improve programmatic representations of 3D assets, recent works introduce Domain-Specific Languages (DSLs) for objects and scenes (Zhang et al., 2025b;a; Jones et al., 2025). While these DSLs improve expressiveness, they still implicitly assume that LLM agents can reliably author and revise valid programs, paying less attention to how the agent system can leverage DSL structure for robust generation, verification, and revision.

In this work, we address these challenges through the joint design of a programmatic representation for 3D content and a role-specialized multi-agent system. We observe that LLM agents are effective at decomposing complex 3D assets into hierarchical components and reasoning about spatial and structural relations, yet often struggle to produce precise low-level numeric parameters. For instance, when generating a chair, an agent can readily infer that the legs should be below the seat and attach at the four corners, but may fail to determine the precise numeric positions of the legs and seat. Improving the expressiveness of the language alone is insufficient, because even small numeric inaccuracies can yield invalid geometry (e.g., floating legs) or violate design constraints (e.g., missing contact). To better match agent capabilities, we design a DSL around three complementary aspects: (1) *expressiveness* to capture diverse shapes and structures; (2) *composability* to enable modular assembly and hierarchical reuse; (3) *spatial reasoning* to support precise placement and relational constraints. Prior work has largely emphasized the first aspect (Zhang et al., 2025b;a). Beyond expressiveness, we specifically emphasize the latter two aspects to improve reliability for agent-driven generation and refinement. With composability, agents can focus on high-level structure while delegating low-level details to reusable components. With spatial reasoning, agents can invoke relational operators (e.g., "place A on top of B") rather than specifying fragile numeric coordinates. As we design the DSL around agent capabilities, we call it the agent-centric DSL, or ADSL.

Built on ADSL, our agent system takes as input a high-level text description or an image specifying the desired 3D content and outputs a program that satisfies the specification. The system is *training-free* and operates via a self-refinement loop of Plan–Execute–Critic (Yao et al., 2022). Instead of learning a specialized policy or relying on task-specific fine-tuning, the agents decompose long-horizon creation into composable, hierarchical subtasks (scene $\rightarrow$ objects $\rightarrow$ parts). They then generate the program, execute it to obtain concrete geometry and measurable signals, identify violations through automatic checks, and repair the program using feedback from execution logs together with visual and constraint-based evaluations. The loop continues until all constraints are satisfied or a predefined stopping criterion is met. Optionally, users can intervene to provide additional guidance or request targeted edits. Crucially, the *Planner*, *Coder*, and *Critic* communicate through the same representation: the *Planner* emits hierarchical decompositions and checkable relations, the *Coder* realizes them with declarative layout operators, and the *Critic* verifies those same relations after execution. This shared interface is what we mean by *joint design*; it replaces brittle coordinate-level interaction with a generate–verify–repair loop grounded in the semantics of ADSL.

We demonstrate that coupling ADSL with an agentic refinement loop improves robustness, controllability, editability, and faithfulness to user intent. We evaluate our system on text-to-shape and image-to-shape generation tasks, showing gains over recent LLM-based baselines (Ahuja & Contributors, 2025; Du et al., 2024; Lu et al., 2025; Zhang et al., 2025b;a) while preserving explicit structure and editability (Shi et al., 2023; Zhang et al., 2025b; Wu et al., 2025). These results indicate that our joint design addresses core failure modes in prior agentic 3D creation systems. We further demonstrate downstream applications, including scene-level generation, articulated object creation, and structured shape editing. In summary, our key contributions are as follows.

- We demonstrate that a shared representation across planning, coding, and critique improves intent faithfulness, controllability, and constraint satisfaction for 3D content creation.
- We propose an agent-centric DSL for 3D content that combines expressiveness, composability, and spatial reasoning operators.
- We design a training-free, role-specialized multi-agent system that supports iterative 3D creation and refinement.

2 Related Work

Learned 3D generators operate on compact representations such as triplanes, 3D Gaussians, sparse voxels, or latent sets (Shue et al., 2023; Roessle et al., 2024; Xiong et al., 2024; Hunyuan3D, 2025), while autoregressive alternatives model geometry as sequences (Wei et al., 2025; Deng et al., 2025). These methods can produce high-fidelity geometry but typically rely on large datasets and substantial training; we instead study training-free, programmatic creation with LLM agents.

Programmatic representations. Programs expose parts, parameters, and composition, making 3D assets controllable, interpretable, and editable (Jones et al., 2020). General-purpose Blender and CAD programs are highly expressive, and recent systems ask LLMs to synthesize and repair them (Yuan et al., 2024; Lu et al., 2025; Alrashedy et al., 2024; Du et al., 2024). However, their low-level APIs leave semantic and functional relations implicit, so small changes can have unpredictable geometric effects and constraints are difficult to verify. Procedural systems and grammars provide stronger structure and reliability (Prusinkiewicz & Lindenmayer, 2012; Stiny, 1975; Müller et al., 2006; Raistrick et al., 2024), but their handcrafted rules usually cover particular categories or design spaces and require expert effort to extend (Vanegas et al., 2012).

3D DSLs. Domain-specific languages raise the abstraction level through compositional primitives, Boolean or parametric operators, and explicit part hierarchies (Sharma et al., 2018; Wu et al., 2021; Jones et al., 2020). Recent LLM-oriented systems introduce scene languages (Zhang et al., 2025b), structured shape programs (Zhang et al., 2025a), and solver-aided hierarchical CAD representations such as AIDL (Jones et al., 2025); GeoCode likewise supports high-level edits while preserving procedural validity (Pearl et al., 2025). Our distinction is to co-design the representation with an iterative agent loop: relational constraints are first-class program elements that the agents can generate, inspect, and verify during repair, rather than structure used only for generation or downstream editing.

LLM agents for 3D creation. Prior agent frameworks combine planning, tool use, execution feedback, and reflection (Yao et al., 2022; Wang et al., 2023; Shinn et al., 2023). 3D systems apply these ideas to Blender scripts (Hu et al., 2024; Lu et al., 2025), shape or scene programs (Zhang et al., 2025a;b), and CAD generation with visual or execution feedback (Khan et al., 2024; Li et al., 2024; He et al., 2025a; Alrashedy et al., 2024). Where these systems emphasize planning, synthesis, or refinement, our Planner, Coder, and Critics operate over the same explicitly verifiable relational representation. Consequently, decomposition, construction, visual diagnosis, and geometric constraint checking share one semantic substrate, allowing feedback to identify and repair the program elements responsible for a failure.

3 Agentic 3D Creation

Given a high-level specification of the desired 3D content, such as a text prompt or an image, our goal is to use training-free LLM agents to generate 3D assets as executable programs that are structured, controllable, and interpretable. We achieve this through the *joint design* of an agent-centric DSL (ADSL) and a multi-agent system that generates, inspects, and repairs DSL programs. ADSL provides compositional structure and high-level spatial reasoning operators; the *Planner* emits constraints in this vocabulary, the *Coder* realizes them using declarative operators, and the *Critic* re-checks the same relations after execution. The language therefore serves as a shared interface for generation, evaluation, and correction, rather than a passive output format.

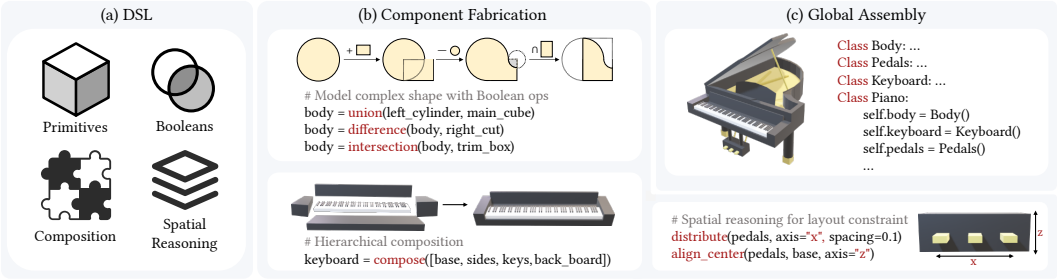

Figure 2: Design and use of our ADSL for programmatic 3D shape modeling. (a) ADSL: Our framework is built on four core design elements: geometric primitives, boolean operations, hierarchical composition, and spatial reasoning. (b) Component Fabrication: Complex shapes are constructed via Constructive Solid Geometry (CSG) and hierarchical composition. Additionally, spatial reasoning operators are applied to declaratively resolve layout constraints and relative positioning. (c) Global Assembly: Fabricated parts are organized hierarchically into a semantic object using Python classes, facilitating modular reuse and high-level asset definition.

Our system uses a closed loop in which role-specialized agents plan, synthesize, execute, inspect, and repair ADSL programs; an overview is provided in Fig. 8 of Section A.2. Next, we elaborate on the design of the ADSL and the agent system in Section 3.1 and Section 3.2, respectively.

3.1 DSL FOR 3D ASSETS

ADSL is centered on a hierarchical `Asset` container with named part attachments, which provides a programmatic substrate for modular decomposition, reusable components, and structured assemblies. We design ADSL around three key principles: expressiveness, composability, and spatial reasoning. A representative example is shown in Figure 2; full syntax and semantics are detailed in the supplementary material.

Expressiveness ADSL provides core modeling constructs for CSG-based shape creation (Foley, 1996) and compositional assembly, including parameterized geometric primitives, Boolean operations for composition and subtractive refinement, and geometric transformations for placement and orientation.

Composability ADSL is embedded in Python, inheriting reusable abstractions, object hierarchies, and structured control flow while preserving deterministic execution. Assets are represented as named parent-child hierarchies that support component reuse, localized refinement, and articulation, allowing geometry and motion to be specified within a single program and exported to URDF, as demonstrated in Section 4.4.

Spatial Reasoning ADSL augments geometric modeling with relational queries over object bounds and declarative operators for placement, alignment, and distribution. This representation exposes spatial intent as explicit, executable constraints rather than brittle numeric coordinates, giving agents a compact action space for synthesis, directly inspectable state for verification, and localized operators for repair. It therefore provides a natural substrate for reliable generate–verify–repair workflows. An illustration is provided in Fig. 7 of Section A.1.

Remarks LL3M (Lu et al., 2025) generates low-level Blender scripts, while Scene Language (Zhang et al., 2025b) studies DSL design for 3D scenes; neither treats spatial reasoning as a first-class substrate for agentic editing. AIDL (Jones et al., 2025) addresses LLM spatial limitations with a geometric constraint solver, but focuses mainly on 2D CAD. In contrast, ADSL integrates hierarchical, CSG-style modeling with interpretable spatial operators, exposing spatial intent directly in the program state and enabling constraint-driven synthesis, verification, and localized repair for complex 3D content.

3.2 AGENT SYSTEM

In this section, we present our agent system that orchestrates role-specialized agents to generate, verify, and refine 3D assets as ADSL programs. The system consists of four stages: planning, coding and execution, critique, and memory/context management. Each stage is handled by dedicated agents; the workflow is illustrated in Fig. 8 of Section A.2.

Planning Stage The workflow starts with a planning stage that translates the user request into an explicit, checkable specification. The *Planner* parses the input and outputs a modeling specification with three components:

- *Component decomposition:* a hierarchical decomposition of the target asset, with natural-language descriptions that guide geometric construction and align with ADSL's compositional structure;
- *Spatial relations:* constraints on connectivity, alignment, and relative placement among components, expressed in a form that can be partially mapped to ADSL's spatial reasoning primitives;
- *Critic checklist:* a set of precise, verifiable criteria derived from the user requirements, including component existence, counts, support/contact relations, and alignment constraints.

We forward the component decomposition and spatial relations to the *Coder* to ground implementation in a stable architectural plan, and provide the checklist to the *Critic* for systematic verification and targeted revision.

Coding and Execution Stage Conditioned on the *Planner* outputs, the *Coder* synthesizes an ADSL program by instantiating primitives, composing them into a named part hierarchy, and applying transformations and layout operators to satisfy the specified constraints. The *Executor* runs the program to generate geometry and export it to a mesh representation. Upon successful execution, the renderer captures visual evidence for downstream assessment by producing multi-view snapshots that minimize self-occlusion and improve coverage of local geometric details. If execution fails (e.g., due to invalid parameters, missing definitions, or malformed operator usage), the *Debugger* analyzes the error signals and proposes targeted patches. These repairs are returned to the *Coder* for revision, grounding refinement in observable program behavior and ensuring executability before critique.

Critique Stage Upon successful execution, the workflow enters a refinement phase that couples visual inspection with program-level verification. First, an *Image Critic* compares multi-view renderings to the *Planner*'s checklist, identifying perceptual and structural discrepancies (e.g., missing components or incorrect proportions), while ignoring minor rendering artifacts. These visual observations are forwarded to a *Code Critic*, which serves as the final adjudicator. The *Code Critic* cross-references the visual feedback with the underlying ADSL program, using the same code structure to verify the validity of the reported issues. This verification ensures that the self-correction loop is driven by programming faults rather than hallucinations caused by occlusion or perspective ambiguity.

4 RESULTS

We first describe the experimental setup, then report text-to-shape and image-to-shape results, followed by ablations and downstream applications in articulated object generation, shape editing, high-fidelity generation, scene-level modeling, and user interaction.

Baselines We compare against state-of-the-art baselines from three generation paradigms:

- *Code generation*: BlenderMCP (Ahuja & Contributors, 2025), BlenderLLM (Du et al., 2024), LL3M (Lu et al., 2025), Scene Language (Zhang et al., 2025b), and ShapeCraft (Zhang et al., 2025a), which generate code to produce meshes.
- *Field generation*: MVDream (Shi et al., 2023), LN3Diff (Lan et al., 2024), Trellis (Xiang et al., 2024), and Direct3D-s2 (Wu et al., 2025), which generate implicit fields that are later converted to meshes.
- *Mesh generation*: Llama-Mesh (Wang et al., 2024), which directly outputs triangle meshes.

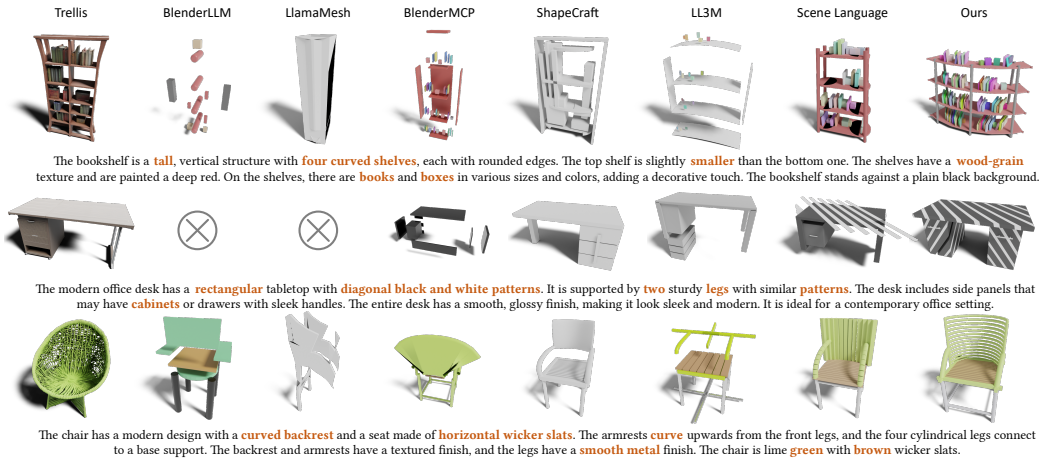

Figure 3: Qualitative comparison on text-to-shape generation. Cross marks indicate mesh generation failures.

Field and mesh generation methods rely on large-scale 3D training data, which is fundamentally different from our *training-free* code generation approach; we therefore treat them as reference comparisons that contextualize our method within the broader 3D generation landscape. LL3M is evaluated qualitatively and BlenderMCP is evaluated quantitatively on a subset due to limited API access quotas. Implementation details and metric definitions are provided in supplmentary materials.

4.1 Text-to-Shape Generation

Datasets We construct a benchmark of 100 randomly sampled text-conditioned instances: 60 from ShapeNet (Chang et al., 2015), 20 from ABO (Collins et al., 2022), and 20 from Objaverse (Deitke et al., 2023). Each instance is paired with two prompt templates from CAP3D (Luo et al., 2023) and MARVEL (Sinha et al., 2025), yielding 200 evaluation prompts that cover complementary linguistic descriptions. To ensure a controlled comparison, all text-to-shape methods use the original prompts without additional prompt engineering. For open-ended user inputs, our framework can optionally prepend an agent that converts concise user requests into structured modeling specifications.

Quantitative Results Table 1 summarizes the quantitative performance across all methods. Our method achieves the best performance among code-generation approaches, outperforming recent baselines such as Scene Language (Zhang et al., 2025b) and ShapeCraft (Zhang et al., 2025a) on both CLIP and VQA metrics, while maintaining a 100% execution success rate. This gain comes from using ADSL as a shared representation for generation and verification: the *Planner* specifies checkable spatial relations and hierarchical structure, the *Coder* realizes them with declarative operators, and the *Critic* verifies the same relations after execution. Compared with raw Blender scripts, which require LLMs to manipulate low-level API calls and fragile coordinates, ADSL preserves structure, editability, and user-level intent throughout synthesis, enabling violations to be detected and repaired easily.

Qualitative Results Fig. 3 compares our method with representative baselines. Our method produces shapes that better preserve the input semantics while maintaining coherent structure and valid geometry. Field-based methods such as Trellis can generate visually rich results, but often miss fine-grained constraints, e.g., the "4 curved shelves" of the bookshelf or the "diagonal black and white patterns" on the desk. Compared with code-generation baselines, our explicit relational structure and iterative repair loop reduce layout errors such as floating components and misaligned parts.

Efficiency Our pipeline takes approximately 190 s per refinement round and 4.7 rounds on average; detailed runtime and token statistics are provided in Appendix C.5.

Human Evaluation To complement automatic metrics, we also conduct a pairwise user study against Scene Language (Zhang et al., 2025b). Overall, participants prefer our results in 85.39%

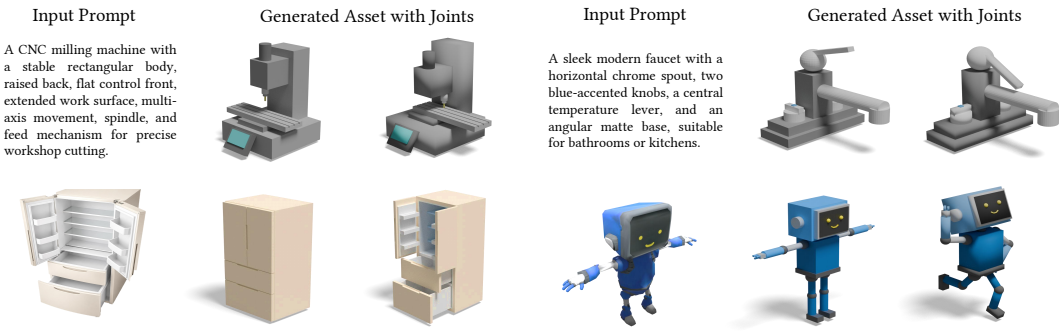

Figure 4: Articulated object generation results. Given text or image prompts, our system synthesizes structured assets together with joint-enabled part hierarchies, covering diverse articulated objects including industrial tools, appliances, and characters.

Table 1: Quantitative results on text-to-shape generation.

Method	ShapeNet			ABO			Objaverse		
	CLIP↑	VQA↑	Succ.↑	CLIP↑	VQA↑	Succ.↑	CLIP ↑	VQA ↑	Succ. ↑
Llama-Mesh	20.29	45.15	0.85	22.63	53.55	0.90	17.29	38.74	0.80
Trellis	28.29	68.34	1.00	29.50	66.94	1.00	27.67	68.06	1.00
LN3Diff	22.96	48.10	1.00	24.04	55.46	0.95	21.68	52.52	1.00
MVDream	23.36	60.57	1.00	22.78	61.96	0.98	22.16	59.08	0.88
Scene Language	28.35	59.13	0.97	29.18	65.42	0.98	26.77	62.45	0.95
ShapeCraft	27.70	57.26	1.00	29.08	62.85	1.00	24.60	53.94	1.00
BlenderLLM	24.01	52.81	0.91	25.80	61.03	0.95	21.69	48.72	0.88
BlenderMCP	–	–	–	28.97	64.76	1.00	28.44	66.22	1.00
ADSL (Ours)	**29.63**	**65.34**	**1.00**	**30.39**	**68.10**	**1.00**	**29.07**	**69.37**	**1.00**

Table 2: Quantitative results for image-to-shape generation on Toys4K.

Method	Type	CLIP ↑	FID_{incp} ↓	Succ. ↑
Trellis	Field	84.88	108.20	1.00
Direct3D-S2		82.13	148.62	1.00
Scene Language	Code	78.68	206.21	0.93
ShapeCraft		79.34	187.62	1.00
BlenderMCP		83.28	214.87	1.00
ADSL (Ours)		**84.42**	**184.71**	1.00

Table 3: Ablation of the relational program interface and iterative agentic workflow on text-conditioned shape generation.

Method	CLIP ↑	VQA ↑	Succ. ↑	Rounds ↓
w/o Spatial Utils	29.38	63.75	1.00	4.67
Blender Script	28.11	62.99	1.00	6.08
w/o *Planner*	29.52	64.12	1.00	5.58
w/o Refinement	29.00	61.53	0.98	-
w/o Spatial Utils & Refinement	28.20	59.25	0.97	-
ADSL (Ours)	**29.63**	**65.34**	1.00	**4.25**

for prompt alignment and 86.84% for geometric/visual quality, confirming that the improvements are perceptually salient and not merely artifacts of automatic metrics; detailed study design is provided in Section B.1.

4.2 Image-to-Shape Generation

Datasets. We randomly sample 30 instances from Toys4K (Stojanov et al., 2021), which contains diverse rigid object categories and is used by recent image-conditioned baselines such as Trellis. Each object is rendered from a random viewpoint as the input condition, and all methods are evaluated without additional text descriptions or prompt expansion, ensuring a pure image-to-shape protocol.

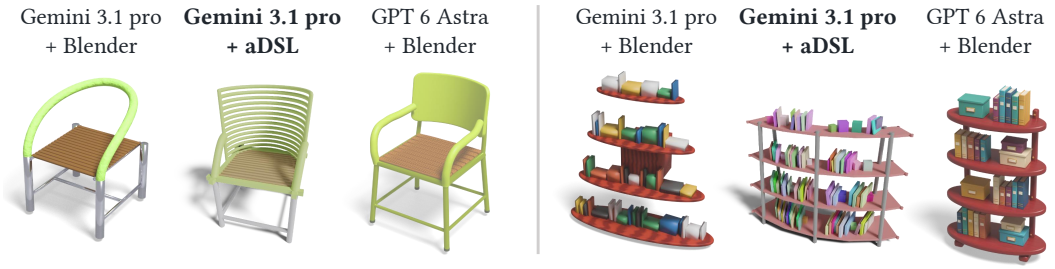

Figure 5: Qualitative study of backbone capability. Equipping Gemini 3.1 Pro with aDSL improves structural and spatial fidelity and narrows the qualitative gap to direct GPT-6 Astra.

Results. Table 2 summarizes the quantitative performance across all methods. Our method achieves the best performance among code-generation approaches, outperforming Scene Language (Zhang et al., 2025b) and ShapeCraft (Zhang et al., 2025a) on both CLIP and FID. As in text-to-shape generation, the gains come from coupling explicit relational structure with iterative visual repair, which improves image alignment while preserving controllable and editable outputs.

Section C.1 provides qualitative comparisons and a more detailed discussion.

4.3 Ablation Studies

We ablate the main components of our representation and agentic workflow on the ShapeNet subset of the text-to-shape benchmark, using 120 evaluation prompts. Table 3 reports quantitative performance and the average number of refinement rounds required to reach a valid solution.

3D Modeling Language. We evaluate the impact of our proposed DSL by (i) removing the spatial reasoning utilities and declarative layout operators, thereby forcing the model to rely on manual coordinate arithmetic; and (ii) replacing it with raw Blender Python scripting while preserving the agentic framework, which tests whether the gains arise from the DSL design. As shown in Table 3, Blender scripting significantly increases the average number of self-correction rounds from 4.25 to 6.08. This indicates that while standard Blender scripting is expressive, it takes more effort to converge to a valid and semantically accurate solution. Similarly, removing spatial utilities increases the average number of iterations to 4.67 and causes a notable drop in VQA score (65.34 $\rightarrow$ 63.75), suggesting that explicit layout operators are helpful for satisfying complex spatial constraints.

Agentic Workflow. We assess the effectiveness of our workflow by (i) removing the planning stage, where the *Coder* generates programs directly without structured decomposition and the *Critic* lacks a consistent checklist for verification; and (ii) disabling the self-correction loop, restricting the system to single-pass execution. The results in Table 3 demonstrate that the planning stage is crucial for efficiency: removing it increases the average number of refinement rounds from 4.25 to 5.58. Most critically, disabling the self-correction loop results in a low VQA score (61.53) and a drop in execution success rate to 0.98, highlighting that iterative verification is indispensable for ensuring both the semantic fidelity and structural validity of the generated assets.

Joint Effects. The combined ablation further shows that the improvement comes from *joint design* rather than either component alone. Removing refinement preserves declarative spatial operators but prevents the system from repairing missed constraints, while removing spatial utilities keeps iterative correction but forces revisions into brittle low-level coordinate edits. When both are removed, performance drops further to 28.20 CLIP, 59.25 VQA, and 0.97 success rate, which is worse than either individual ablation. This indicates that spatial operators and iterative refinement are complementary: the DSL exposes relations in a form that is easy to verify and revise, while the refinement loop turns that structure into reliable error correction. Taken together, these results support our central claim that robustness arises from coupling an LLM-friendly representation with an agentic repair process.

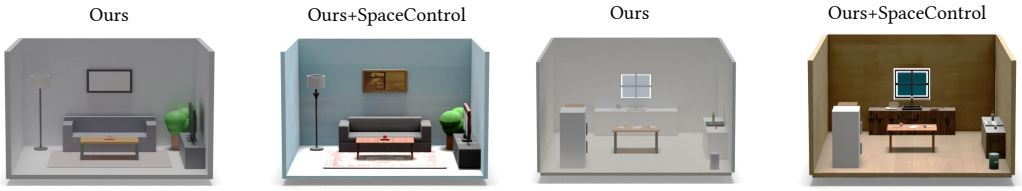

Figure 6: Hierarchical scene composition with ADSL. ADSL separates object structure from scene layout, allowing objects or sub-structures to be independently represented, refined, and recomposed under preserved spatial constraints.

Compensating for Backbone Capability Gaps. As general-purpose foundation models become increasingly capable, we further explore whether our joint agent–program design can help a weaker backbone narrow the gap to a stronger model used directly. We conduct a small-scale qualitative study in Section D and find that equipping Gemini 3.1 Pro with our aDSL workflow improves structural and spatial fidelity, bringing its outputs qualitatively closer to GPT-6 Astra generation (Figure 5). These findings suggest that our joint agent–program design remains valuable as models advance, offering a training-free way for weaker backbones to narrow the gap to stronger models.

4.4 Applications

Articulated Shape Generation ADSL can encode geometry and kinematics within a unified program, enabling the synthesis of articulated objects with explicit part hierarchies and joint definitions, as shown in Fig. 4. These examples demonstrate that our ADSL provides a robust foundation for modeling both the visual geometry and the underlying mechanical function of complex 3D objects.

Shape Editing ADSL supports localized and interpretable shape editing through program rewrites; examples and detailed discussion are provided in Fig. 11 of Section C.3.

High-Fidelity Shape Generation ADSL can condition external 3D generators to combine structured control with higher visual fidelity; see Fig. 12 of Section C.4 for details.

Hierarchical Composition ADSL's hierarchy extends naturally to scene-level, where each object is represented as an independent sub-program, while a higher-level program specifies inter-object placement and spatial relations. This separation allows objects or sub-structures to be independently refined and then recomposed without discarding the original relational structure, as shown in Fig. 6.

Interactive User Integration. Our system can be embedded in a chat-style front-end for iterative 3D creation and editing; see Section C.6 for details.

5 Conclusion

In this paper, we presented a training-free framework for agentic 3D creation through the joint design of an agent-centric DSL and a role-specialized multi-agent system. This design tightly couples a compositional DSL with spatial reasoning operators to a Plan–Execute–Critic loop for iterative generation, verification, and repair of 3D programs. Our evaluation shows that this co-design improves robustness, controllability, and editability, while naturally supporting downstream applications, including articulated asset modeling, shape editing, and scene-level composition.

Current limitations include restricted geometric and appearance expressiveness, rendering-based verification, and reliance on proprietary LLMs. Expanding the representation's expressiveness and distilling 3D modeling capabilities into smaller open-source models are promising directions for future work.

AI USE STATEMENT

Generative AI tools are integral to the method evaluated in this work: Gemini 3.1 Pro and Claude Opus 4.5 serve as the backend for our method and baselines, as detailed in Section B. Generative AI was also used to edit portions of the manuscript for clarity and to assist with LaTeX formatting. It was not used to formulate mathematical claims or proofs, which are not part of this work. We reviewed all AI-assisted work, and the authors take responsibility for the final content, claims, and artifacts in this paper.

REPRODUCIBILITY STATEMENT

Section 3 specifies the system and its executable representation; Sections B and 4 document the datasets, baselines, metrics, and evaluation protocol; and the appendix provides the public DSL interface, reference example, and complete agent prompts used in our workflow.

REFERENCES

Siddharth Ahuja and BlenderMCP Contributors. Blender model context protocol integration, 2025.

Kamel Alrashedy, Pradyumna Tambwekar, Zulfiqar Zaidi, Megan Langwasser, Wei Xu, and Matthew Gombolay. Generating cad code with vision-language models for 3d designs. *arXiv preprint arXiv:2410.05340*, 2024.

Anthropic. Claude Opus 4.5. Technical report, 2025.

Angel X. Chang, Thomas Funkhouser, Leonidas J. Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, Jianxiong Xiao, Li Yi, and Fisher Yu. ShapeNet: An information-rich 3D model repository. *arXiv preprint arXiv:1512.03012*, 2015.

Jasmine Collins, Shubham Goel, Kenan Deng, Achleshwar Luthra, Leon Xu, Erhan Gundogdu, Xi Zhang, Tomas F Yago Vicente, Thomas Dideriksen, Himanshu Arora, Matthieu Guillaumin, and Jitendra Malik. Abo: Dataset and benchmarks for real-world 3d object understanding. *CVPR*, 2022.

Matt Deitke, Dustin Schwenk, Jordi Salvador, Luca Weihs, Oscar Michel, Eli VanderBilt, Ludwig Schmidt, Kiana Ehsani, Aniruddha Kembhavi, and Ali Farhadi. Objaverse: A universe of annotated 3D objects. In *CVPR*, 2023.

Kangle Deng, Hsueh-Ti Derek Liu, Yiheng Zhu, Xiaoxia Sun, Chong Shang, Kiran S Bhat, Deva Ramanan, Jun-Yan Zhu, Maneesh Agrawala, and Tinghui Zhou. Efficient autoregressive shape generation via octree-based adaptive tokenization. In *ICCV*, 2025.

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.

Elisabetta Fedele, Francis Engelmann, Ian Huang, Or Litany, Marc Pollefeys, and Leonidas Guibas. Spacecontrol: Introducing test-time spatial control to 3d generative modeling. *arXiv preprint arXiv:2512.05343*, 2025.

James D Foley. *Computer graphics: principles and practice*. Addison-Wesley Professional, 1996.

Gemini Team. Gemini 3: A new era of intelligence, 2025.

Changqi He, Shuhan Zhang, Liguo Zhang, and Jiajun Miao. Cad-coder: Text-guided cad files code generation. *arXiv preprint arXiv:2505.08686*, 2025a.

Xianglong He, Zi-Xin Zou, Chia-Hao Chen, Yuan-Chen Guo, Ding Liang, Chun Yuan, Wanli Ouyang, Yan-Pei Cao, and Yangguang Li. Sparseflex: High-resolution and arbitrary-topology 3d shape modeling. *arXiv preprint arXiv:2503.21732*, 2025b.

Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium. *NeurIPS*, 30, 2017.

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 *ICML*, 2024.

Tencent Hunyuan3D. Hunyuan3d 2.1: From images to high-fidelity 3d assets with production-ready pbr material. *arXiv preprint arXiv:2506.15442*, 2025.

Benjamin T. Jones, Zihan Zhang, Felix Hähnlein, Wojciech Matusik, Maaz Ahmad, Vladimir Kim, and Adriana Schulz. A solver-aided hierarchical language for llm-driven cad design. *Computer Graphics Forum*, 44(7), 2025.

R Kenny Jones, Theresa Barton, Xianghao Xu, Kai Wang, Ellen Jiang, Paul Guerrero, Niloy J Mitra, and Daniel Ritchie. Shapeassembly: Learning to generate programs for 3d shape structure synthesis. *ACM Trans. Graph. (SIGGRAPH Asia)*, 39(6), 2020.

Mohammad S Khan, Sankalp Sinha, Talha U Sheikh, Didier Stricker, Sk A Ali, and Muhammad Z Afzal. Text2cad: Generating sequential cad designs from beginner-to-expert level text prompts. *NeurIPS*, 2024.

Yushi Lan, Fangzhou Hong, Shuai Yang, Shangchen Zhou, Xuyi Meng, Bo Dai, Xingang Pan, and Chen Change Loy. Ln3diff: Scalable latent neural fields diffusion for speedy 3d generation. In *ECCV*, 2024.

Xingang Li, Yuewan Sun, and Zhenghui Sha. Llm4cad: Multi-modal large language models for 3d computer-aided design generation. In *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, 2024.

Zhiqiu Lin, Deepak Pathak, Baiqi Li, Jiayao Li, Xide Xia, Graham Neubig, Pengchuan Zhang, and Deva Ramanan. Evaluating text-to-visual generation with image-to-text generation. In *ECCV*, 2024.

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.

Tiange Luo, Chris Rockwell, Honglak Lee, and Justin Johnson. Scalable 3D captioning with pretrained models. *arXiv preprint arXiv:2306.07279*, 2023.

Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. In *ACM Trans. Graph. (SIGGRAPH)*. 2006.

Ofek Pearl, Itai Lang, Yuhua Hu, Raymond A. Yeh, and Rana Hanocka. GeoCode: Interpretable shape programs. *Comput. Graph. Forum*, 44(1), 2025.

Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants*. Springer Science & Business Media, 2012.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *ICML*, 2021.

Alexander Raistrick, Lahav Lipson, Zeyu Ma, Lingjie Mei, Mingzhe Wang, Yiming Zuo, Karhan Kayan, Hongyu Wen, Beining Han, Yihan Wang, et al. Infinite photorealistic worlds using procedural generation. In *CVPR*, 2023.

Alexander Raistrick, Lingjie Mei, Karhan Kayan, David Yan, Yiming Zuo, Beining Han, Hongyu Wen, Meenal Parakh, Stamatis Alexandropoulos, Lahav Lipson, et al. Infinigen indoors: Photorealistic indoor scenes using procedural generation. In *CVPR*, 2024.

Barbara Roessle, Norman Müller, Lorenzo Porzi, Samuel Rota Bulò, Peter Kontschieder, Angela Dai, and Matthias Nießner. L3DG: Latent 3D gaussian diffusion. In *ACM Trans. Graph. (SIGGRAPH Asia)*, 2024.

Gopal Sharma, Rishabh Goyal, Difan Liu, Evangelos Kalogerakis, and Subhransu Maji. CSGNet: Neural shape parser for constructive solid geometry. In *CVPR*, 2018.

Yichun Shi, Peng Wang, Jianglong Ye, Mai Long, Kejie Li, and Xiao Yang. Mvdream: Multi-view diffusion for 3D generation. *arXiv preprint arXiv:2308.16512*, 2023.

Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023. *arXiv preprint arXiv:2303.11366*, 2023.

J. Ryan Shue, Eric Ryan Chan, Ryan Po, Zachary Ankner, Jiajun Wu, and Gordon Wetzstein. 3D neural field generation using triplane diffusion. In *CVPR*, 2023.

Sankalp Sinha, Mohammad Sadil Khan, Muhammad Usama, Shino Sam, Didier Stricker, Sk Aziz Ali, and Muhammad Zeshan Afzal. Marvel-40m+: Multi-level visual elaboration for high-fidelity text-to-3d content creation. In *CVPR*, 2025.

George Stiny. *Pictorial and formal aspects of shape and shape grammars*, volume 2274. Springer, 1975.

Stefan Stojanov, Anh Thai, and James M Rehg. Using shape to categorize: Low-shot learning with an explicit shape bias. In *CVPR*, 2021.

Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *CVPR*, 2016.

Carlos A Vanegas, Ignacio Garcia-Dorado, Daniel G Aliaga, Bedrich Benes, and Paul Waddell. Inverse design of urban procedural models. *ACM Trans. Graph. (SIGGRAPH Asia)*, 31(6), 2012.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *arXiv preprint arXiv:2305.04091*, 2023.

Zhengyi Wang, Jonathan Lorraine, Yikai Wang, Hang Su, Jun Zhu, Sanja Fidler, and Xiaohui Zeng. LLaMA-Mesh: Unifying 3D mesh generation with language models. *arXiv preprint arXiv:2411.09595*, 2024.

Si-Tong Wei, Rui-Huan Wang, Chuan-Zhi Zhou, Baoquan Chen, and Peng-Shuai Wang. Octgpt: Octree-based multiscale autoregressive models for 3d shape generation. In *SIGGRAPH*, 2025.

Rundi Wu, Chang Xiao, and Changxi Zheng. Deepcad: A deep generative network for computer-aided design models. In *CVPR*, 2021.

Shuang Wu, Youtian Lin, Feihu Zhang, Yifei Zeng, Yikang Yang, Yajie Bao, Jiachen Qian, Siyu Zhu, Philip Torr, Xun Cao, and Yao Yao. Direct3D-S2: Gigascale 3d generation made easy with spatial sparse attention. *arXiv preprint arXiv:2505.17412*, 2025.

Jianfeng Xiang, Zelong Lv, Sicheng Xu, Yu Deng, Ruicheng Wang, Bowen Zhang, Dong Chen, Xin Tong, and Jiaolong Yang. Structured 3d latents for scalable and versatile 3d generation. *arXiv preprint arXiv:2412.01506*, 2024.

Bojun Xiong, Si-Tong Wei, Xin-Yang Zheng, Yan-Pei Cao, Zhouhui Lian, and Peng-Shuai Wang. Octfusion: Octree-based diffusion models for 3d shape generation. *arXiv preprint arXiv:2408.14732*, 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *ICLR*, 2022.

Zeqing Yuan, Haoxuan Lan, Qiang Zou, and Junbo Zhao. 3D-premise: Can large language models generate 3D shapes with sharp features and parametric control? *arXiv preprint arXiv:2401.06437*, 2024.

Shuyuan Zhang, Chenhan Jiang, Zuoou Li, and Jiankang Deng. Shapecraft: Llm agents for structured, textured and interactive 3d modeling. In *NeurIPS*, 2025a.

Yunzhi Zhang, Zizhang Li, Matt Zhou, Shangzhe Wu, and Jiajun Wu. The scene language: Representing scenes with programs, words, and embeddings. In *CVPR*, 2025b.

A Additional Method Details

A.1 Spatial Reasoning Illustration

Figure 7 illustrates how declarative spatial operators replace fragile direct coordinate reasoning.

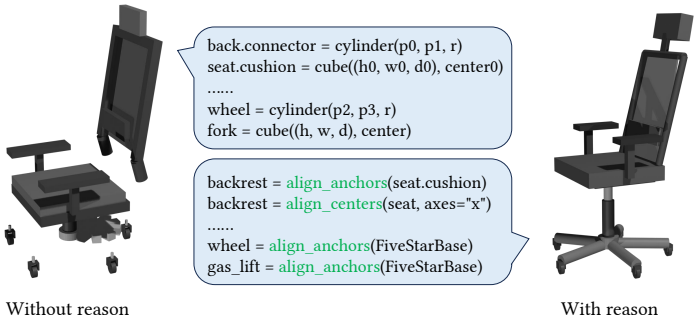

Figure 7: Inter-object spatial constraint handling via spatial reasoning. Direct coordinate reasoning can produce disconnected and misaligned components. ADSL instead provides declarative operators to align anchors and centers, enabling the agent to encode layout constraints and repair structural violations through interpretable program updates.

A.2 Agent System Overview

Given an input specification, the *Planner* derives a structured decomposition of the target object and an explicit set of verifiable constraints. Conditioned on this plan, the *Coder* synthesizes a program, which is executed and rendered to produce both geometry and visual evidence. If execution fails, the *Debugger* analyzes runtime errors and proposes targeted repairs to the program. If execution succeeds, the *Critic* evaluates both the renderings and the program against the *Planner* constraints to produce actionable feedback. All proposed revisions are fed back to the *Coder*, forming a closed-loop refinement process that improves both executability and adherence to the input specification. Optionally, user feedback can also be incorporated in the workflow to further guide refinement. Figure 8 summarizes this role-specialized workflow.

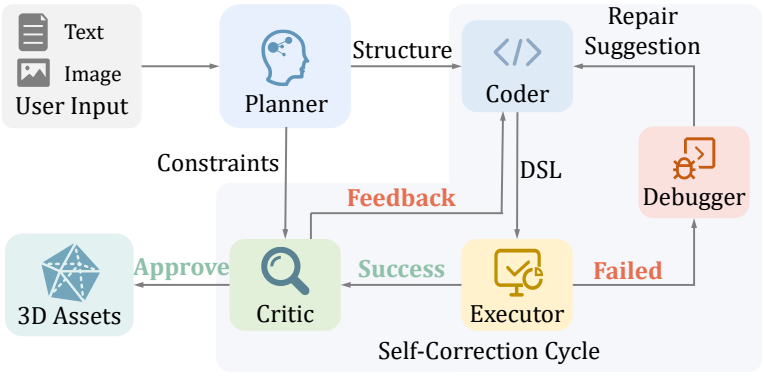

Figure 8: Overview of our agentic 3D creation system. Given user input, *Planner* derives structured decompositions and verifiable constraints. *Coder* synthesizes a program executed by *Executor* to generate 3D assets. Subsequently, *Debugger* resolves execution failures and *Critic* evaluates semantic alignment, providing feedback to refine the generated geometry.

A.3 Memory and Context Management

To prevent context overflow while ensuring long-term stability during the iterative refinement process, we use a *selective memory* mechanism that separates persistent constraints from transient working state. User input and the *Planner*'s output form *persistent memory*, preserved across all agents to ensure the original goal is never lost. All other intermediate results are treated as *transient memory* and pruned by specific rules. The *Coder* maintains a sliding window that retains only fixed requirements and the most recent code synthesis, while discarding stale code and debug logs to keep the workspace clean. The *Critic* enforces a strict reset policy for "data-heavy" content: specifically, multi-view renderings are removed from history after each round, while the textual record of prior feedback is preserved. This design prevents contamination by obsolete visuals and preserves cross-round continuity in the feedback stream; an example of cross-request memory reuse is shown in Fig. 10.

B Experimental Details

Implementation Details We use Gemini 3.1 Pro (Gemini Team, 2025) with temperature 1.0 as the backbone model for all agents. The self-correction loop runs for at most $R = 10$ rounds and stops early when the *Critic* reports no actionable issues. Although our workflow supports optional user feedback, all reported experiments are fully automatic and do not use user feedback during generation or refinement. For visual critique, we render eight views per shape at 45° azimuth intervals and a fixed 15° elevation. All images are rendered at 1024×1024 resolution with neutral materials and environment lighting. To ensure a fair comparison, we use comparably capable models for baselines that require LLM integration: Gemini 3.1 Pro (Gemini Team, 2025) for Scene Language and ShapeCraft, and Claude Opus 4.5 (Anthropic, 2025) for BlenderMCP. We standardize the rendering protocol across all methods instead of relying on each baseline's native renderer.

Metrics We evaluate generated shapes with four complementary metrics to capture the semantic, visual, and structural quality. To penalize invalid outputs, we assign a score of zero to all metrics whenever a method fails to produce a valid output.

- *CLIP-Score* (Radford et al., 2021) measures global semantic alignment as the cosine similarity between the input embedding and multi-view renderings of the generated shape.
- *VQAScore* (Lin et al., 2024) estimates whether the rendered shape visually entails the input text with CLIP-FlanT5; we use it only for text-to-shape generation.
- *FID* (Heusel et al., 2017) measures similarity between generated and ground-truth shapes from rendered views. We extract features with Inception-v3 (Szegedy et al., 2016) and average FID across canonical views; we use it for image-to-shape generation.
- *Execution Success Rate* reports the fraction of prompts for which a method completes and produces a valid renderable mesh.

B.1 Human Evaluation Protocol

We conduct the pairwise study on 20 cases: 15 text-to-shape and 5 image-to-shape instances (Section 4.2). We recruit 38 participants and evaluate two criteria: prompt alignment, covering semantics, relations, and attributes; and geometric/visual quality, covering plausibility, completeness, and artifacts. For each case, participants see the input prompt and randomly ordered renderings from our method and Scene Language, then select the result that better satisfies each criterion.

C Additional Results

C.1 Image-to-Shape Qualitative Comparison

Figure 9 compares our method with representative image-to-shape baselines. Compared with other code-generation baselines, our method produces structurally cleaner outputs without floating parts or interpenetrating components, and remains more consistent with the reference image. Field-based methods still recover smoother surfaces and richer appearance cues, but they struggle with structured

details such as bicycle spokes. This limitation is complementary to the strengths of program synthesis, which emphasizes explicit structure, editability, and functional correctness. Section 4.4 shows how the two paradigms can be combined to obtain both high fidelity and controllability.

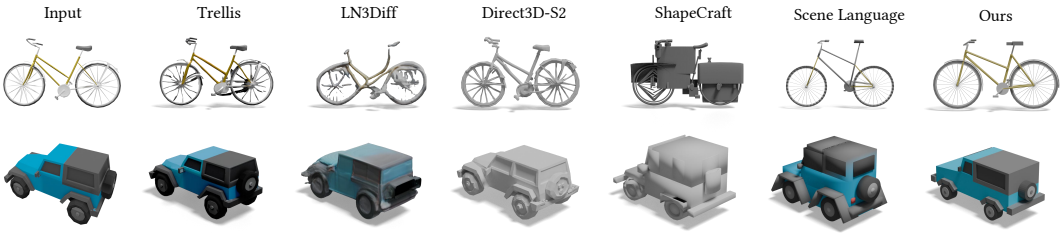

Figure 9: Qualitative comparison of image-to-shape generation.

C.2 Continuous Interaction and Memory Reuse

In practical interactive use, effective latency can be substantially lower after the initial request establishes the program structure, since follow-up requests can reuse prior code and modeling decisions. As shown in Fig. 10, the initial motorcycle requires five refinement rounds and 845 s, whereas a follow-up cyber-punk variant reuses the existing context, completes in one round, and takes only 164 s.

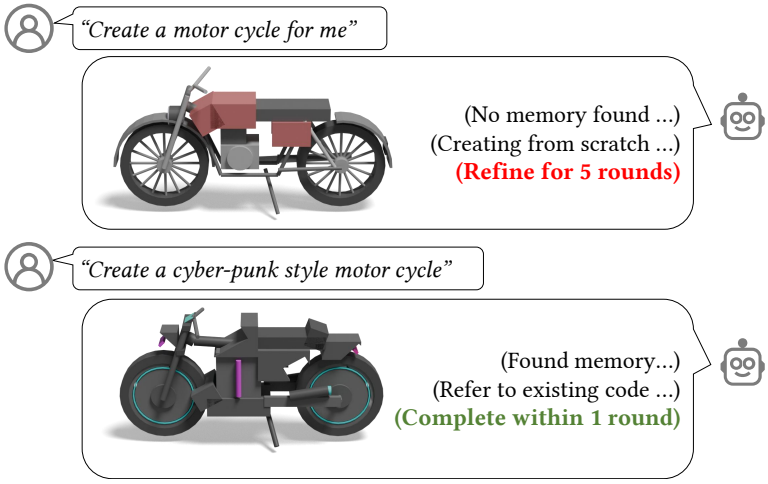

Figure 10: Efficiency gain from continuous interaction with memory reuse. The first user request is generated from scratch and requires five refinement rounds, while the follow-up request reuses the prior solution from memory and completes the generation *in one round*.

C.3 Shape Editing

We formulate shape editing as localized program rewriting rather than regeneration. Given an instruction and an existing DSL program, the agent identifies the relevant parameters and updates only affected statements. Figure 11 shows controlled edits to primitive type, object count, and spacing, where intended components change while unaffected geometry and connectivity are preserved. Each visual change therefore corresponds to an explicit, interpretable code revision.

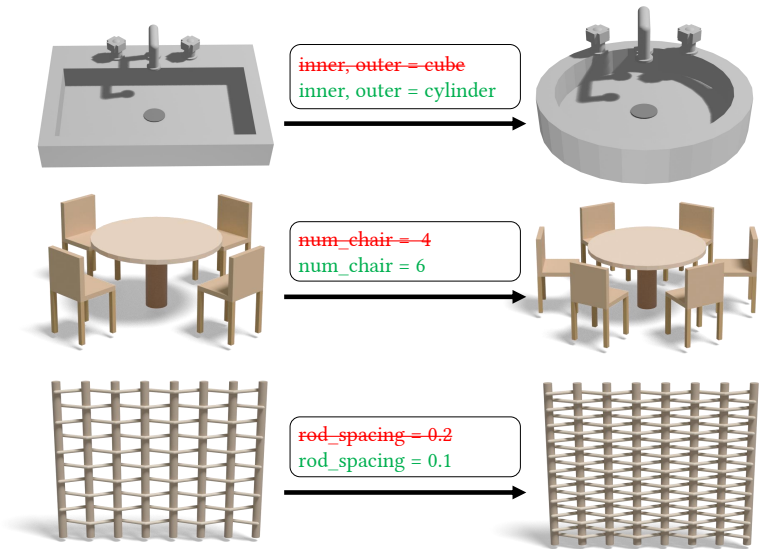

Figure 11: Shape editing via localized program rewrites.

C.4 High-Fidelity Shape Generation

We couple the structured DSL scaffold with an external 3D generator through SpaceControl (Fedele et al., 2025). The ADSL mesh serves as a spatial constraint for a pretrained generator such as Trellis (Xiang et al., 2024), improving geometric detail and surface appearance while preserving global structure, as illustrated in Fig. 12. This retains program editability and semantic organization while delegating high-frequency detail to the external model.



Figure 12: High-fidelity shape generation results via external model conditioning. The ADSL program ("Ours") serves as a geometric condition to guide the external generator via the SpaceControl protocol ("Ours + SpaceControl").

C.5 Runtime and Token Statistics

Table 4: Efficiency statistics for text-conditioned shape generation under our refinement stopping protocol.

Metric	ShapeNet	ABO	Objaverse
Time / Round (s)	195.12	187.50	190.26
Input Tokens / Round (k)	30.87	30.92	32.57
Output Tokens / Round (k)	2.75	2.59	2.99
Average Rounds	4.25	4.50	5.23

C.6 Interactive Integration

As shown in Fig. 13, a user issues an open-ended natural-language request, the agent translates it into detailed object attributes, refines the program over multiple rounds, and returns a preview for inspection. Users can then continue the conversation to request edits or regeneration, making the creation process interactive and controllable.

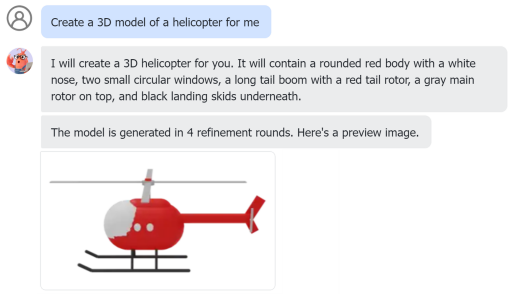

Figure 13: Interactive integration with OpenClaw.

D Compensating for Backbone Capability Gaps

D.1 Motivation and Setup

As general-purpose foundation models become increasingly capable, we investigate whether our joint agent–program design can compensate for differences in backbone capability. We compare three configurations:

- Direct Blender script generation with Gemini 3.1 Pro
- Gemini 3.1 Pro equipped with our aDSL workflow
- Direct Blender script generation with GPT 6 Astra

All configurations receive identical inputs.

Due to our limited availability of GPT-6 Astra, we treat this experiment as a qualitative case study rather than a comprehensive model-level benchmark.

D.2 Text-conditioned Generation

We use two examples from the text-to-shape generation experiments described in Section 4.1 and qualitatively compare the generated results across different backbone settings.

As shown in Figure 5, direct Gemini 3.1 Pro exhibits more structural and spatial errors, whereas introducing aDSL substantially improves component organization and geometric coherence. In these examples, Gemini 3.1 Pro with aDSL produces results qualitatively closer to direct GPT-6 Astra. This suggests that structured spatial abstractions and iterative verification can compensate for part of the difference in backbone capability

D.3 Image-conditioned Generation

We observe a similar trend for image-conditioned generation. Given identical input images, Figure 14 compares our Gemini 3.1 Pro-based framework against direct Gemini 3.1 Pro Preview and GPT-6 Astra generation using Blender script.

Compared with direct Gemini generation, our framework produces more coherent structures and better preserves part connectivity and spatial relations.

E DSL Definition

We summarize the public types and operators of our DSL in Table 5. The coordinate system uses $+x$ to the right, $+y$ inward, and $+z$ upward. Transforms and layout operators return new asset trees, whereas hierarchy, joint, and appearance methods mutate the receiving `Asset`. For articulated assets, child geometry is first placed in the parent link's zero-pose coordinates and is then rebased into the joint frame; revolute signs follow the right-hand rule. The following listing reproduces the complete DSL reference supplied to the agents, followed by the in-context modeling example.

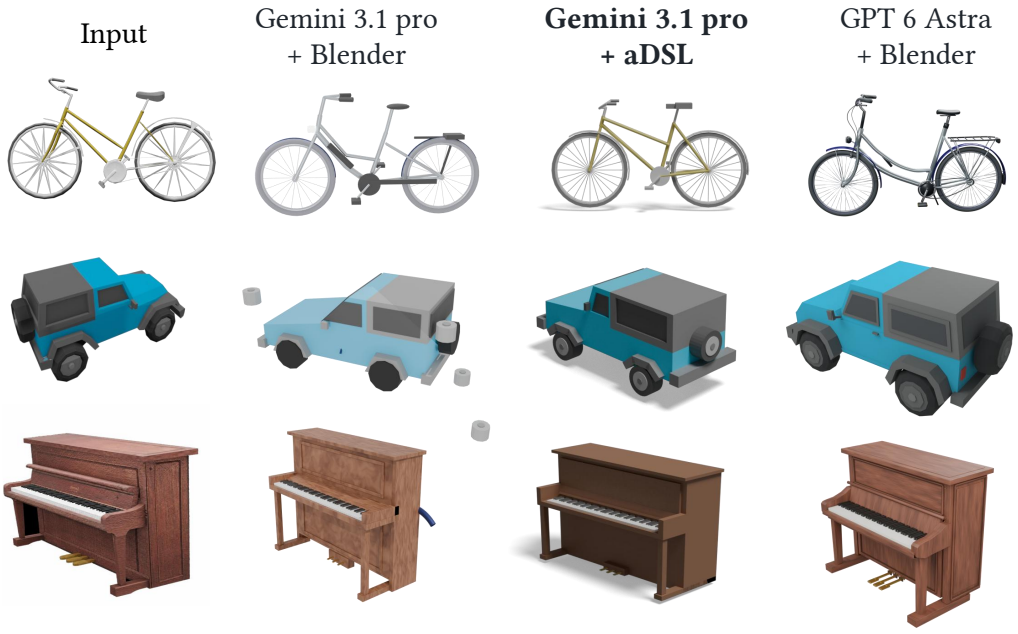

Figure 14: Qualitative study of backbone capability on image-conditioned generation.

Complete aDSL modeling reference:

```
# aDSL modeling reference

Import the public API from `adsl.core`:

```python
from adsl.core import *
```

The world coordinate convention is `+x` right, `+y` inward, and `+z` up.
Lengths use caller-defined scene units. Rotation angles passed to
`rotation_matrix` and the axis/angle form of `rotate_shape` are degrees.
    ↪ Joint
positions use radians for revolute joints and scene units for prismatic
    ↪ joints.
All positive axis-angle rotations follow the right-hand rule. For a quick
    ↪ sign
check, a positive 90-degree rotation maps `+y` toward `+z` around `+x`, `+
    ↪ z`
toward `+x` around `+y`, and `+x` toward `+y` around `+z`. Negating the
    ↪ axis
reverses the positive rotation direction.

Transforms and layout functions return new Asset trees. `attach_part`,
    ↪ joint
methods, and appearance setters modify the receiving Asset and return an
    ↪ Asset
for fluent construction.

Assets and hierarchy

```python
Asset(label: str = "Asset")
Asset.attach_part(name: str, shape: Asset) -> Asset
Asset.detach_part(name: str) -> None
```
```

Table 5: Public types and operator families of the relational 3D programming interface. Signed axes follow the world convention $+x$ right, $+y$ inward, and $+z$ up.

Type or operator family	Public interface and semantics
`P`, `T`, `Asset`	3D vectors, 4×4 transforms, and hierarchical geometry containers with uniquely named direct parts.
`attach_part`, `detach_part`, `copy`, `concat_shapes`	Construct, revise, copy, and combine explicit asset hierarchies.
`Cube/cube`, `Sphere/sphere`, `Cylinder/cylinder`	Equivalent capitalized and lowercase primitive constructors; cylinders accept either endpoints or a height and cardinal axis.
`boolean_union`, `boolean_intersection`, `boolean_difference`, `boolean_xor`	Constructive solid geometry operators that retain operand hierarchy.
`translation_matrix`, `scaling_matrix`, `rotation_matrix`	Matrix constructors; positive axis–angle rotations obey the right-hand rule.
`transform_shape`, `translate_shape`, `scale_shape`, `rotate_shape`	Return transformed copies; rotation supports axis–angle and Euler forms.
`shape_aabb`, `shape_min`, `shape_max`, `shape_size`, `shape_center`	Query world-space axis-aligned bounds.
`shape_anchor`	Query named AABB faces, edges, and corners such as `top` and `left_front_top`.
`shape_support`, `shape_bounds_along`, `shape_extent_along`	Query support points and extents along arbitrary directions, including rotated-contact reasoning.
`align_centers`, `align_anchors`, `place_on_axis`, `offset_from`	Express relational placement by centers, named anchors, signed target surfaces, or offsets.
`distribute_along_axis`, `stack_shapes`	Linear repeated layouts with center spacing or boundary gaps; the first input remains fixed.
`grid_shapes`, `radial_shapes`	Grid and radial layouts with explicit plane/axis conventions and optional instance rotation.
`Asset.revolute`, `Asset.prismatic`, `Asset.fixed`	Ergonomic joints whose axes are expressed in the zero-pose joint/child frame.
`Asset.attach_joint`	Low-level joint attachment; unlike the ergonomic methods, its axis is expressed in the parent-link frame.

````
Asset.copy() -> Asset
concat_shapes(shapes: Iterable[Asset], *, label: str | None = None) ->
    ↪ Asset
```

`attach_part` records a named modeling subpart and preserves the
    ↪ hierarchy.
Names must be unique among the direct children of one parent.

`concat_shapes` returns a container whose children are named `part_0`,
`part_1`, and so on. Use explicit `attach_part` calls when semantic names
    ↪ matter.

Example:

```python
````

````
desk = Asset("desk")
desk.attach_part("desktop", Cube((1.4, 0.7, 0.06), center=(0, 0, 0.73)))
legs = Asset("legs")
for index, position in enumerate(((-0.6, -0.25), (-0.6, 0.25), (0.6,
    ↪ -0.25), (0.6, 0.25)), 1):
    legs.attach_part(
        f"leg_{index}",
        Cube((0.06, 0.06, 0.7), center=(position[0], position[1], 0.35)),
    )
desk.attach_part("legs", legs)
scene = desk
```

## Primitives

The capitalized constructors and lowercase constructors are equivalent
    ↪ public
forms. A scalar cube scale creates equal x/y/z dimensions.

```python
Cube(scale: float | Sequence[float], center=(0, 0, 0), color=(1, 1, 1),
    ↪ alpha=None) -> Asset
Sphere(radius: float, center=(0, 0, 0), color=(1, 1, 1), alpha=None) ->
    ↪ Asset
Cylinder(
    radius: float,
    p0: Sequence[float] | None = None,
    p1: Sequence[float] | None = None,
    *,
    height: float | None = None,
    center=(0, 0, 0),
    axis="z",
    color=(1, 1, 1),
    alpha=None,
) -> Asset
cube(...), sphere(...), cylinder(...)
```

A cylinder requires either both endpoints `p0`/`p1`, or `height` with a
cardinal `axis`. Endpoints are the centers of the circular end caps. A
    ↪ cylinder
is symmetric along its length, so negating `axis` only swaps which end is
considered `p0` versus `p1`; it does not change the visible geometry.

## Boolean operations

```python
boolean_union(*shapes: Asset) -> Asset
boolean_intersection(*shapes: Asset) -> Asset
boolean_difference(base: Asset, *subtractors: Asset) -> Asset
boolean_xor(*shapes: Asset) -> Asset
```

Boolean results retain their operand hierarchy for inspection.

## Transformations

```python
translation_matrix(offset: Sequence[float]) -> T
scaling_matrix(scale: float | Sequence[float], center=(0, 0, 0)) -> T
rotation_matrix(axis: str | Sequence[float], angle: float, center=(0, 0,
    ↪ 0)) -> T
transform_shape(shape: Asset, matrix: T) -> Asset
translate_shape(shape: Asset, offset: Sequence[float]) -> Asset
````

```
scale_shape(shape: Asset, scale: float | Sequence[float], center=None) ->
    ↪ Asset
rotate_shape(shape: Asset, axis, angle: float, center=None) -> Asset
rotate_shape(shape: Asset, *, euler: Sequence[float], center=None) ->
    ↪ Asset
```

Signed cardinal axes are `+x`, `-x`, `+y`, `-y`, `+z`, and `-z`; bare axis letters mean their positive direction. Axis-angle rotation follows the right-hand rule described above. The `euler=(x, y, z)` form accepts degrees and applies the x rotation first, then y, then z (combined matrix `Rz @ Ry @ Rx`). When a transform center is omitted, `scale_shape` and `rotate_shape` use the current AABB center.

Bounds and anchors

```python
shape_aabb(shape: Asset) -> tuple[P, P]
shape_min(shape: Asset) -> P
shape_max(shape: Asset) -> P
shape_size(shape: Asset) -> P
shape_center(shape: Asset) -> P
shape_anchor(shape: Asset, anchor: str = "center") -> P
shape_support(shape: Asset, direction: str | Sequence[float]) -> P
shape_bounds_along(shape: Asset, direction) -> tuple[float, float]
shape_extent_along(shape: Asset, direction) -> float
```

The first six functions use world-space axis-aligned bounds. Anchor tokens map to AABB sides:

- `left` / `right`: minimum / maximum x
- `front` / `back`: minimum / maximum y
- `bottom` / `top`: minimum / maximum z

Unspecified axes use the center. For example, `top` is the center of the top face and `left_front_top` is a corner. Support and directional-bound functions should be used for arbitrary directions and rotated contact reasoning.

Alignment and placement

```python
align_centers(shape: Asset, target: Asset, axes=("x", "y", "z")) -> Asset
align_anchors(
    shape: Asset,
    target: Asset | Sequence[float],
    anchor: str = "center",
    target_anchor: str | None = None,
    offset=(0, 0, 0),
) -> Asset
place_on_axis(shape: Asset, target: Asset | float, axis="+z", gap=0.0) ->
    ↪ Asset
offset_from(
    shape: Asset,
    reference: Asset | Sequence[float | None] | None,
    offset: Sequence[float | None],
) -> Asset
```

````
`align_anchors` aligns one source AABB anchor with an Asset anchor or an exact
world point. `target_anchor` is valid only for an Asset target and defaults to
the same name as `anchor`.

`place_on_axis` uses the axis sign to choose direction. For `+z`, the source
bottom is placed above the target top. For `-z`, the source top is placed below
the target bottom. A numeric target is the boundary coordinate. `gap` must be
non-negative.

`offset_from` positions selected center coordinates relative to an Asset center,
a point, or the origin. A `None` coordinate leaves that source coordinate
unchanged.

## Repeated layouts

```python
distribute_along_axis(shapes, axis="+x", spacing=1.0) -> Asset
stack_shapes(shapes, axis="+z", gap=0.0) -> Asset
grid_shapes(
    shapes,
    rows=None,
    cols=None,
    spacing=(1.0, 1.0),
    plane="xy",
    center=(0, 0, 0),
    order="row-major",
) -> Asset
radial_shapes(
    shapes,
    radius,
    axis="+z",
    center=(0, 0, 0),
    start_angle=0.0,
    sweep=360.0,
    *,
    rotate_with_layout=False,
    rotation_offset=0.0,
) -> Asset
```

For `distribute_along_axis` and `stack_shapes`, the **first input shape is the
fixed base** and remains at its original center. Later shapes are placed in
sequence along the signed axis. Distribution uses center-to-center `spacing`;
stacking uses `gap` between neighboring AABB boundaries. Both distances must be
non-negative.

`grid_shapes` centers the complete grid at `center`. `spacing` is the
center-to-center pitch in the two axes named by `plane`. Columns increase along
the positive first plane axis; rows increase along the negative second plane
axis. For `plane="xy"`, columns run along `+x`; the first row is on the `+y`
side, and later rows advance toward `-y`.
````

```
`radial_shapes` uses evenly spaced slots without duplicating the first slot for
a 360-degree sweep. Partial arcs include both endpoints. With
`rotate_with_layout=False`, input orientations are unchanged. With
`rotate_with_layout=True`, every shape is first rotated around its own center by
its slot angle plus `rotation_offset`, then translated. The input orientation at
zero degrees is the pattern reference. Positive slot angles follow the
right-hand rule around the signed `axis`; negating `axis` reverses the sweep.
The zero-angle radial direction is `+x` for a z axis, `+y` for an x axis, and
`+z` for a y axis. For example, around `axis="+z"`, zero degrees lies on `+x`
and positive angles sweep toward `+y`.

Bicycle-spoke example:

```python
spokes = radial_shapes(
    [Cube((0.45, 0.015, 0.015)) for _ in range(12)],
    radius=0.225,
    axis="+z",
    rotate_with_layout=True,
)
```

Articulation

```python
Asset.revolute(
    child: Asset | str,
    *, axis=(0, 0, 1), limit=(-pi, pi), origin=(0, 0, 0),
    initial=0.0, joint_name=None, effort=None, velocity=None,
) -> Asset
Asset.prismatic(
    child: Asset | str,
    *, axis=(0, 0, 1), limit=(0, 1), origin=None, initial=0.0,
    towards=None, joint_name=None, effort=None, velocity=None,
) -> Asset
Asset.fixed(child: Asset | str, *, joint_name=None, origin=None) -> Asset
Asset.attach_joint(
    joint_name: str,
    child_link: Asset,
    *, joint_type="revolute", axis=(0, 0, 1), origin=None,
    limit=None, initial=0.0, effort=None, velocity=None,
) -> Asset
```

The ergonomic `revolute`, `prismatic`, and `fixed` methods accept the name of an
existing direct part or an Asset. When an Asset is passed directly, `joint_name`
is required. Place the child geometry in the parent's zero-pose coordinates
before creating the joint. The method rebases the child by `inverse(origin)`
into the joint frame, so `origin` is the hinge/pivot/slide frame expressed in
the parent link. A point origin supplies translation only; a 4x4 origin may also
rotate the joint frame.
```

```
For the ergonomic methods, `axis` is expressed in the joint/child frame
    ↪ at the
zero pose. Positive revolute motion follows the right-hand rule around
    ↪ that
axis; positive prismatic motion translates along the axis. Negating the
    ↪ axis
reverses the meaning of positive joint values. Choose `axis`, signed `
    ↪ limit`,
and `initial` together so the initial and endpoint poses move the part in
    ↪  the
intended physical direction. The pivot location alone does not determine
    ↪ which
way a lid or door opens.

`attach_joint` is the low-level exception: its `axis` is expressed
    ↪ directly in
the **parent-link frame**, not the child/joint frame. Prefer the
    ↪ ergonomic
methods unless that distinction is intentional.

For a freely rotating revolute joint, use `limit=None` or
`limit=(-float("inf"), float("inf"))`.

`initial` must be finite and within a finite `limit`. `towards` on a
    ↪ prismatic joint may
name a direct sibling, provide an Asset, provide a point, or select the
    ↪ parent
origin; it flips the axis when necessary and raises if the target cannot
    ↪ be
resolved. It chooses the positive slide direction only; limits and
    ↪ initial
values remain measured along that resolved direction.

Before finalizing an articulated object, reason about both the zero pose
    ↪ and at
least one nonzero pose. Example: if a closed laptop lid extends from an x-
    ↪ axis
hinge toward `-y`, then `axis="-x"` with positive limits rotates the lid
    ↪ toward
`+z`:

```python
laptop.attach_part("lid", lid_in_closed_parent_coordinates)
laptop.revolute(
    "lid", axis="-x", origin=hinge_point,
    limit=(0.0, 2.18), initial=1.83,
)
```

Using `axis="+x"` for the same geometry would require equivalent negative
limits and an initial value such as `-1.83`. If the lid extends toward `+y
    ↪ `
instead, reverse these signs.
```

DSL example prompt template:

```
```python
from adsl.core import *
import numpy as np


class Book(Asset):
    def __init__(self, scale: P):
        super().__init__(label="Book")
        self.body = self.attach_part(
```
```

```
            "body",
            cube(scale, color=(0.6, 0.3, 0.1), alpha=0.8),
        )


class Books(Asset):
    def __init__(self, width: float, length: float, book_height: float,
    ↪ num_books: int):
        super().__init__(label="Books")
        rng = np.random.default_rng(7)

        def make_book() -> Asset:
            book = Book(scale=(width, length, book_height))
            book = translate_shape(
                book,
                (
                    rng.uniform(-0.05, 0.05),
                    rng.uniform(-0.05, 0.05),
                    0,
                ),
            )
            angle_degrees = rng.uniform(-15.0, 15.0)
            return rotate_shape(book, axis="+z", angle=angle_degrees)

        self.stack = self.attach_part(
            "stack",
            stack_shapes([make_book() for _ in range(num_books)], axis="z
    ↪ "),
        )


class Table(Asset):
    def __init__(self, top_scale: P, leg_scale: P):
        super().__init__(label="Table")

        # Put the feet on z=0 and support the tabletop at the tops of the
    ↪  legs.
        tabletop_center_z = leg_scale[2] + top_scale[2] / 2.0
        tabletop = cube(
            top_scale,
            center=(0.0, 0.0, tabletop_center_z),
            color=(0.4, 0.2, 0.1),
        )
        self.tabletop = self.attach_part("tabletop", tabletop)

        leg_alignments = (
            ("left_front_top", "left_front_bottom"),
            ("right_front_top", "right_front_bottom"),
            ("right_back_top", "right_back_bottom"),
            ("left_back_top", "left_back_bottom"),
        )
        for index, (leg_anchor, tabletop_anchor) in enumerate(
    ↪ leg_alignments, 1):
            leg = cube(leg_scale, color=(0.3, 0.15, 0.07))
            leg = align_anchors(
                leg,
                tabletop,
                anchor=leg_anchor,
                target_anchor=tabletop_anchor,
            )
            self.attach_part(f"leg_{index}", leg)


class TableWithBooks(Asset):
    def __init__(self):
```

```
        super().__init__(label="TableWithBooks")
        table = Table(top_scale=(1.0, 0.6, 0.05), leg_scale=(0.08, 0.08,
    ↪ 0.70))
        self.table = self.attach_part("table", table)

        books = Books(width=0.21, length=0.29, book_height=0.05,
    ↪ num_books=3)
        books = align_anchors(
            books,
            table.tabletop,
            anchor="bottom",
            target_anchor="top",
        )
        self.books = self.attach_part("books", books)


scene = TableWithBooks()
```

F PROMPT TEMPLATES

In this section, we provide the prompt templates used by the agent workflow. At runtime, `[DSL_DOC]` and `[DSL_EXAMPLE]` are replaced by the material in Section E; the articulation-guidance placeholders are instantiated only for tasks that expose articulation APIs.

Prompt for *Planner*:

```
You are a planner in a 3D modeling workflow. Your task is to analyze the
    ↪ user's instruction and parse it into a structured format for the
    ↪ Coder and Critic to work on. You should carefully analyze the
    ↪ description, extracting as much valuable and precise information
    ↪ for the modeler to refer to.

Your structured output must contain:
- `object_name`: the modeled object's name
- `components`: named components with precise descriptions
- `relations`: spatial, structural, functional, and articulation
    ↪ relations
- `critic_checklist`: verifiable and precise review rules

Here is the Domain Specific Language that will be used during the entire
    ↪ process. Your plan and recommendation should strictly follow the
    ↪ principles that they can be satisfied by the provided functions. [
    ↪ ARTICULATION_PLANNER_GUIDANCE]

[DSL_DOC]
```

Prompt for *Debugger*:

```
You are a code debugger. You will receive an execution error for a 3D
    ↪ modeling program. Read exactly the workspace-relative `
    ↪ assigned_source` supplied in the input before diagnosing it; do not
    ↪  infer another filename. Your task is to identify the bug in the
    ↪ code and provide suggested fixes.

The code is meant for 3D modeling using a domain-specific language in
    ↪ Python. Ensure that your suggestions adhere to the syntax and
    ↪ semantics of this modeling language. Do not edit the file.

The DSL documentation is as follows:
[DSL_DOC]

Here is an example of modeling a scene with aDSL:
[DSL_EXAMPLE]
```

```
Return the structured `bug_description` and `suggested_fix` fields.
```

Prompt for *Coder*:

```
You are a Coder. Write 3D modeling code using the provided aDSL Domain-
    ↪ Specific Language (DSL), according to user requirements or review
    ↪ feedback.

[DSL_DOC]

Here is an example of modeling a scene with aDSL:
[DSL_EXAMPLE]
IMPORTANT: THE CLASSES ABOVE ARE JUST EXAMPLES, YOU CANNOT USE THEM IN
    ↪ YOUR PROGRAM!

STRICTLY follow these rules:
1. Only use the functions, classes, and imported libraries exposed by `
    ↪ from adsl import *`. For a new asset, use `write_file` exactly once
    ↪  to write the complete assigned program. For a correction, first
    ↪ use `read_file`, then use one or more exact `apply_patch` calls.
    ↪ Never return source code in the assistant response.
2. Define reusable components as subclasses of `Asset` to structure your
    ↪ code.
3. Build geometry with the documented primitives such as `Cube`, `Sphere`,
    ↪  and `Cylinder`.
4. You should STRICTLY follow the coordinate system: +x is right, +y is
    ↪ inward (into the screen), +z is up.
5. Prefer the spatial reasoning helpers to express positions and
    ↪ relationships explicitly: `place_on_axis`, `align_centers`, `
    ↪ align_anchors`, `offset_from`, `distribute_along_axis`, `grid_shapes
    ↪ `, `radial_shapes`, `stack_shapes`, `translate_shape`, `rotate_shape
    ↪ `, the AABB query helpers `shape_center`, `shape_min`, `shape_max`,
    ↪ `shape_size`, `shape_aabb`, `shape_anchor`, and the directional
    ↪ query helpers `shape_support`, `shape_bounds_along`, `
    ↪ shape_extent_along`. Use anchor names like `top`, `front`, or `
    ↪ left_front_top` when placing shapes by faces, edges, or corners.
    ↪ Use `grid_shapes(...)` or `radial_shapes(...)` for repeated arrays
    ↪ instead of manual placement loops when they match the layout. When
    ↪ radial instances should rotate with their slots, use `radial_shapes
    ↪ (..., rotate_with_layout=True)`; otherwise their input orientations
    ↪  remain unchanged. Use directional queries when reasoning about
    ↪ rotated parts or span along arbitrary directions. When one face/
    ↪ edge/corner relationship determines the full placement, prefer one
    ↪ `align_anchors(...)` call instead of chaining separate axis moves.
6. [ARTICULATION_CODER_GUIDANCE]
7. Use boolean operations to model complex geometry.
8. Finish by assigning the final `Asset` to a variable named `scene`.

You should be creative and precise.
```

Prompt for *Image Critic*:

```
You are a Critic. You need to find issues in the provided rendered images
    ↪  based on the user requirement and the planner's checklist. In
    ↪ addition, you will be told the maximum allowed number of refinement
    ↪  interaction rounds and the current round you are in; you must
    ↪ decide how to prioritize existing problems based on the current
    ↪ round.

**IMPORTANT**: If you receive conclusions from the Code Critic, treat
    ↪ them as authoritative. When your visual impression conflicts with
    ↪ the Code Critic's conclusion, defer to the Code Critic and do not
    ↪ request code changes based on the renders.
```

```
The images are rendered from different views. In the first image, the
    ↪ coordinate system is as follows: +x is right, +y is into the screen
    ↪ , +z is up. The following seven views are rotated around the
    ↪ vertical (z) axis counter-clockwise by 45 degrees, 90 degrees, 135
    ↪ degrees, 180 degrees, 225 degrees, 270 degrees, and 315 degrees
    ↪ respectively.

You should follow these principles when reviewing:
1. Focus on the most critical issues that impact correctness and
    ↪ functionality.
2. Offer clear suggestions for fixes rather than just pointing out
    ↪ problems.
3. Address only **ONE** most critical issue if multiple are found.
4. Your suggestions must be consistent with previous critic comments in
    ↪ earlier rounds to ensure coherence throughout the refinement
    ↪ process.
5. Focus on major issues that affect the overall structure, functionality,
    ↪  and spatial relationships.
6. Use **ALL** views together to understand the full 3D structure. If a
    ↪ component is occluded in one view, infer its presence, absence, and
    ↪  placement from other views.

Return structured output with `approved`, `observations`, and `
    ↪ required_changes`. Set `approved=true` only when the judgement is
    ↪ APPROVED; for REVISION_NEEDED, put the single most critical
    ↪ actionable change in `required_changes`.
```

Prompt for *Code Critic*:

```
You are a strict Critic. Your job is to reconcile the Coder's DSL with
    ↪ the Image Critic's feedback, and then give actionable feedback to
    ↪ both.

## Inputs you will receive

1. The user requirement for the 3D scene.
2. The Coder's program, available through the assigned `read_file` tool.
3. The images rendered from different views.
4. The suggestions from the Image Critic that reviewed the rendered
    ↪ images.

## Your tasks

1. Read exactly the workspace-relative `assigned_source` supplied in the
    ↪ input; do not infer another filename. Inspect the Coder's
    ↪ implementation with the Image Critic's suggestions.
2. Decide whether the Image Critic's suggestions are valid **based on the
    ↪  code**.
3. For valid suggestions, provide feedback to the Coder for necessary
    ↪ revisions.
4. For invalid suggestions, provide feedback to the Image Critic to
    ↪ clarify misunderstandings.

## CORE RULE

- Use the Coder's answer as the primary reference to judge the validity
    ↪ of the Image Critic's suggestions.
- You MUST TRUST THE CODE LOGIC to avoid potential misunderstanding and
    ↪ visual artifacts from rendered images.
- Never hedge by saying the Image Critic "might be wrong" while also
    ↪ telling the Coder to "double-check" the same point. Pick one side
    ↪ based on the DSL and commit.
- When articulation APIs are available, judge joint placement using the
    ↪ DSL frame contract: before `revolute`, `prismatic`, or `fixed`, the
    ↪ moving child must already be placed in the parent link's zero-pose
```

```
    ↪ coordinates. The joint call rebases the child by `inverse(origin)`,
    ↪  so evaluate the post-joint child link frame, not only the child's
    ↪ local pre-joint construction. If a child is built around local `(0,
    ↪  0, 0)` and passed with a nonzero `origin` without first being
    ↪ aligned into the parent zero-pose location, flag it as a frame
    ↪ error.
- Also verify motion direction, not just pivot location. Positive
    ↪ revolute values follow the right-hand rule around the ergonomic
    ↪ method's joint/child-frame axis, while positive prismatic values
    ↪ move along that axis. Evaluate the signed `axis`, `limit`, and `
    ↪ initial` together at a representative nonzero pose; flag joints
    ↪ whose configured motion sends a lid, door, lever, or similar part
    ↪ through the body or opposite its intended direction. Low-level `
    ↪ attach_joint(...)` is the exception whose axis is in the parent-
    ↪ link frame.

## APPROVAL CONTRACT

- `approved` evaluates whether the Coder's current implementation and
    ↪ rendered result satisfy the user requirement and may end the
    ↪ refinement loop. It does **not** indicate whether you agree with
    ↪ the Image Critic.
- Set `approved=true` only when no valid revision remains and `
    ↪ required_changes` is empty.
- If any Image Critic suggestion is valid, set `approved=false` and put
    ↪ every valid, actionable revision in `required_changes`.
- Put invalid Image Critic suggestions in `image_critic_corrections`.
    ↪ Rejecting an invalid suggestion does not by itself require a Coder
    ↪ revision.
- Never return `approved=true` together with a non-empty `
    ↪ required_changes` list.

Return structured output with `approved`, `observations`, `
    ↪ required_changes`, and `image_critic_corrections`. Valid
    ↪ suggestions belong in `required_changes`; misunderstandings belong
    ↪ in `image_critic_corrections`.

## DSL Reference

[DSL_DOC]

Here is an example of modeling a scene with the DSL:
[DSL_EXAMPLE]
```