

Procedura

Agentic 3D Modeling with Procedural Control

Youtian Lin¹, Yikang Yang¹, Zhanpeng Hu¹, Mengqi Zhou¹, Feihu Zhang², Xun Cao¹
Jiaheng Liu^{1,†}, Yao Yao^{1,†}

¹Nanjing University ²Envision [†]Corresponding authors



Figure 1. **Procedura** turns a text prompt into an editable procedural assembly, a parametric program whose named parts are joined by typed mates, written with a frozen LLM and no 3D training. Left: the compiled geometry. Right: further objects coloured by their named modules, a part decomposition that comes free with the representation.

Native 3D generators now recover impressive mesh geometry from a single image. However, a dense mesh stays soft where a machined object should be sharp, it carries no part decomposition, and it exposes no parameter a user could edit. To address this, we explore the paradigm of 3D shape as code, leveraging and scaling the coding ability of an LLM for 3D modeling. We introduce **Procedura**, a novel 3D modeling agent framework that writes an object as a procedural assembly, a parametric program whose named parts are joined by typed, machine-checkable mates. From a text prompt, the agent plans the object as an assembly graph and writes the program part by part, solving each placement from the mated frames rather than guessing it, and admitting a part only once compile, mate, and connectivity checks pass. A decoupled vision critic then refines the assembly one diagnosed fix at a time. Moreover, the same graph carries per-part materials and a simulator-validated articulation. We evaluate on P3D-Bench under its assembly judge, and with the same judge on MechBench-36, our hard-surface benchmark. On both, **Procedura** outperforms state-of-the-art native 3D generators and every prior 3D-code agent on judged quality, produces the sharpest edges of any method we evaluate, and is the only one whose output is an editable, part-structured program.

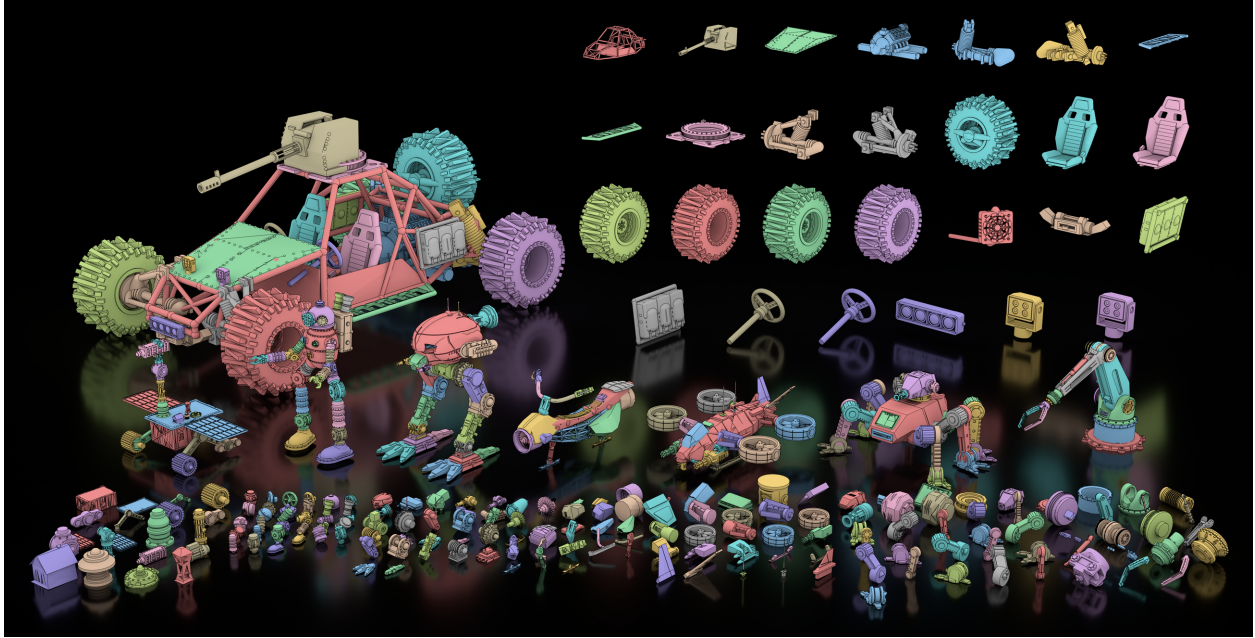


Figure 2. A shape written as code is natively part-structured. *Procedura* builds each object as a parametric program with one named module per physical part, so the model decomposes into meaningful, editable segments at no extra cost, since there is no post-hoc segmentation step. The assault buggy (left) is shown with per-part colours and fanned out into its named parts (roll cage, turret, seats, wheels, drivetrain, panels), alongside other objects and a library of individual parts. A generated mesh is a single opaque surface. In contrast, every part here is individually addressable and re-parameterizable, and the same decomposition drives the per-part materials (Sec. 3.5) and articulation (Sec. 3.6).

1 Introduction

Machines, vehicles, and tools are the objects that fill games, simulators, and factory floors, and generating them is a fundamental problem at the intersection of computer vision and graphics. Native 3D generators have advanced rapidly and now recover impressive mesh geometry from a single image [15, 37, 42, 46]. However, neither of the two dominant generator families turns this fidelity into a usable production asset. Native generators produce the shape as an SDF or a voxel grid, whereas autoregressive generators write the mesh directly, vertex by vertex and face by face [5, 27, 35]. Both are trained on large 3D datasets that are hard to collect, and both output raw triangles with no structure attached. Neither family delivers geometry that is both detailed and precise. Surfaces decoded from a grid stay soft and scan-like where a machined object demands sharp edges and exact faces, and meshes written token by token hold only as much detail as the token budget allows. Nothing in the mesh says which triangles belong to which part. A part decomposition must therefore come from a separate segmentation model applied afterwards [24, 50, 51], and even then the cuts follow surface geometry rather than meaning, so we still cannot reliably tell which piece is, say, the left finger of a robot hand. Editing is also hard, since the mesh exposes no parameter to adjust; changing the object means sculpting the surface directly, which shuts out the non-expert user. Taken together, prior methods fall short on high-precision geometry, meaningful part decomposition, and editability.

Writing the shape as code offers another route, since a program is parametric and part-structured by construction. State-of-the-art LLMs such as GPT and Gemini [9, 29] now write high-quality code, and this ability already reaches 3D, from Three.js scenes to Blender scripts [12, 26]. Yet a 3D shape that an LLM writes as code still comes out toy-like, extremely low-poly and without fine detail [48]. The gap has a cause. A single pass must commit every part, dimension, and placement at once, with no view of what its code builds. Even

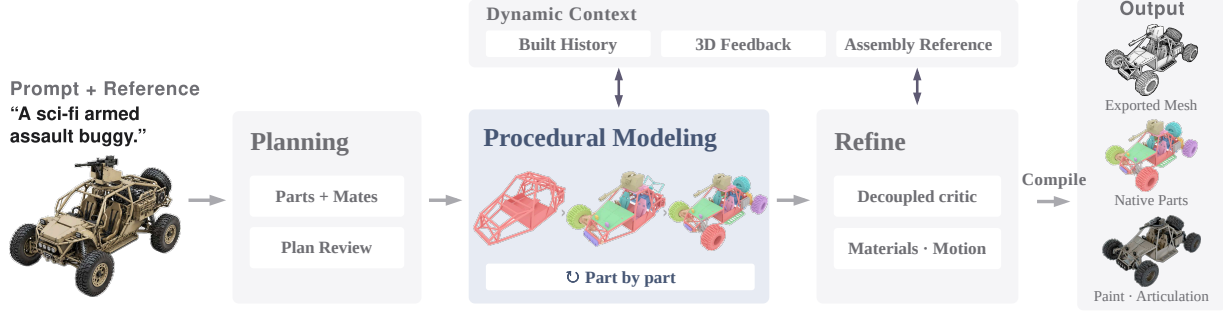


Figure 3. Overview of Procedura. From a text prompt, **Procedura** first synthesizes a reference view and then runs three stages. In Planning, it decomposes the object into an assembly graph of named parts joined by typed mates. In Procedural Modeling, the mate-driven build, it writes the 3D program part by part; each part is generated against its mates and a render of the partial build, its placement is solved from the mated frames, and a verification stack of compile, mate, and connectivity checks guards every commit. In Refine, a decoupled vision critic diagnoses the rendered model and gates one fix at a time. The same graph then completes the asset with per-part materials and an articulation validated in Isaac Sim.

recent agentic systems do not close this gap [1, 7, 55]. Can we use the coding ability of an LLM to generate detailed, high-quality 3D shapes that meet the three requirements above?

To answer this question, we design **Procedura** (Fig. 1), an agentic 3D modeling framework that leverages the code generation ability of an LLM and scales it to 3D shapes with high-grade detail and a native, meaningful part decomposition. Specifically, **Procedura** lets a frozen LLM write the shape in constructive solid geometry (CSG), where every face is an exact plane or cylinder, so edges come out sharp where a generated mesh stays soft. Our key idea is to cast the code as a procedural assembly, a parametric program whose named parts are joined by typed, machine-checkable mates such as a peg into a socket or a hinge about an axis. This structure turns one large blind generation into many small, checkable steps. Every part is a named, editable module (Fig. 2) that the model writes in a single call, each placement is solved from the mated frames rather than guessed, and compile, mate, and connectivity checks admit every commit, so the build grows into a detailed, many-part object without drifting [48]. Given a text prompt, **Procedura** synthesizes a reference view, plans the assembly graph, builds the program part by part, and refines it with a decoupled vision critic (Fig. 3).

We evaluate **Procedura** on P3D-Bench under its assembly judge, and with the same judge on MechBench-36, our hard-surface benchmark, against native generators, prior 3D-code agents, and single-shot LLM baselines. On both benchmarks, **Procedura** outperforms all of them on judged quality. Our contributions are as follows:

- **A 3D-shape-as-code modeling agent.** We explore the paradigm of 3D shape as code and introduce **Procedura**, an agentic framework that leverages and scales the coding ability of a frozen LLM for 3D modeling, yielding detailed, many-part 3D shapes from a text prompt with no 3D training.
- **Procedural assembly with typed mates.** We represent an object as a parametric CSG program whose named parts are joined by typed, machine-checkable mates. Placement is solved from the mated frames and every commit is verified, so the build stays connected and precise and every part remains editable by name. The same graph further carries per-part materials and a simulator-validated articulation.
- **Benchmark and state-of-the-art results.** On P3D-Bench and on MechBench-36, **Procedura** outperforms state-of-the-art native 3D generators and all prior 3D-code agents, with the sharpest edges and the only editable, part-structured output.

2 Related Work

Native 3D generation. A large body of work synthesizes a 3D shape directly as continuous geometry. Optimization-based methods distill a 2D diffusion prior through score distillation [23, 30, 38], reaching high fidelity at the cost of a long per-object optimization. Feed-forward methods map a prompt or a single image to a point cloud, an implicit field, or a set of Gaussians in one pass [11, 18, 28, 36, 54], and native 3D diffusion models now recover mesh geometry of striking fidelity and broad coverage [15, 37, 42, 43, 46]. In parallel, autoregressive models emit the mesh itself token by token [5, 27, 35]. However, these models optimize fidelity to the input view alone. Trained on scanned and artist-made surfaces, they emit dense, soft, scan-like meshes with no named parts, no editable dimensions, and rounded edges where a machined object needs sharp ones. Part structure, when needed, comes from a post-hoc segmentation model [24, 50, 51], which cuts the surface after the fact and attaches no dimensions or relations to the parts it names. In contrast, we represent an object as the assembly program that constructs it, which makes parts, dimensions, and relations explicit while requiring no 3D training.

Program-based 3D generation. Representing a shape as the program that builds it yields an inherently editable, parametric model. Early work induces constructive solid geometry (CSG) trees [19, 32] or structured assembly programs [17], and CAD-native methods learn sketch-extrude command sequences from large engineering datasets [40, 41, 47]. More recently, large language models write parametric 3D programs directly or translate text into construction-command sequences [20, 26]; semantic commenting then recovers part labels for an existing program [53]. Production CAD treats assembly itself as a first-class object, where a graph of typed mates between part-local connectors drives placement and kinematics alike, and design-for-assembly practice prescribes how good parts join: self-locating features, matched male-female pairs sharing one nominal dimension, and explicit fits [4]. Learned-assembly research brings that structure to machine learning, predicting mates or joints for existing human-made parts [16, 39] from large CAD datasets. However, no generative system authors this structure. The program-writing line remains single-shot, producing the code once with no visual feedback, so many-part objects come out with wrong part counts and misplaced parts [48], and nothing in the emitted program states how two parts relate. **Procedura** generates the parts and their mate graph jointly, building part by part against a render of what is already placed, so the structure that assembly research assumes as input becomes our output.

Agentic 3D-code generation. Closest to our setting, recent systems wrap a general LLM in a draft-execute-revise loop that writes 3D code, runs the compiler, and revises from rendered or programmatic feedback [1, 3, 7, 34, 55], and domain-tuned coders fine-tune a model to emit CAD code [10, 21]. Beyond static shape, agentic pipelines author articulated assets by generating code for parts and their kinematics [34, 55], vision-language systems articulate an existing mesh by predicting its joints [22], and LLM scene systems arrange retrieved assets by predicted layout or sample procedural environments [6, 8, 12, 31, 49]. These systems confirm that a coding agent can turn a prompt into an editable program, yet they share two limits. Parts are glued by raw transforms the model estimates from a picture, and kinematics is a stage that follows geometry, so a joint axis is fitted to a finished surface rather than read off the structure that placed the part. **Procedura** answers both. Every relation is a typed mate checked on the compiled mesh, so placement is solved rather than guessed and connectivity is enforced during generation instead of patched afterward [2, 14]; kinematic mates live in the same graph, so a hinge is the very relation that seated its two parts. Moreover, an independent vision critic, separate from the model that wrote the code, diagnoses every build and gates each edit. Self-refinement instead lets the same model critique its own output [25, 33, 52], and such a reviewer often fails to find its own errors [13].

Mate Library



Figure 4. The mate library. The nine static mating features a part may declare, each rendered from the compiler macros the build emits. Teal is the male half added to the new part and grey the female half cut from its partner; both derive from one shared nominal dimension and a signed fit offset.

3 Method

Given a text prompt, our goal is a 3D model that meets the three requirements of Sec. 1: high-precision geometry, a meaningful part decomposition, and editability. **Procedura** therefore generates the program that builds the mesh rather than the mesh itself. We define this representation, a procedural assembly whose parts are joined by typed mates (Sec. 3.1). The agent authors it in three stages (Fig. 3): it plans the assembly graph (Sec. 3.2), builds the program one part at a time under solved placement and deterministic verification (Sec. 3.3), and refines it through a decoupled vision critic (Sec. 3.4). The same graph also completes the asset with per-part materials (Sec. 3.5) and a simulator-validated articulation (Sec. 3.6). One frozen LLM drives every reasoning step, so **Procedura** needs no 3D training; every other component in the pipeline is deterministic.

3.1 Procedural Assembly Representation

Formulation. **Procedura** represents an object as a procedural assembly $A = (P, C)$, where $P = \{p_1, \dots, p_n\}$ is an ordered set of parts, each a named parametric CSG module, and C is a set of mates, each a typed relation between the mate frames of two parts. The shipped geometry is the mesh $M = \text{COMPILE}(A)$ that the compiler produces. This representation answers the three requirements by construction. Geometry is precise since a program’s faces are planes and its bores are cylinders, so its edges stay sharp at any resolution. The part decomposition is the module list itself, so it is exact and needs no separate segmentation model. Editing is a parameter change, since every dimension is a named parameter bound to a named part. The representation also records relations. A mate states how two parts join, so placement can be solved and verified, and the same relation later drives articulation.

Typed mates. A mate $c = (i, j, \text{type}, F_i, F_j, d, \phi)$ joins a frame F_j published by the new part p_j to a frame F_i published by an earlier part p_i . Its type is drawn from a closed vocabulary of nine static mating features (bolt-pattern, peg-socket, seat-face, flange, tab-slot, press-fit, lip-rabbet, snap-tab, and key; Fig. 4) and three kinematic ones (revolute, prismatic, and spherical). The two halves share one nominal dimension d , and a fit class $\phi \in \{\text{clearance}, \text{location}, \text{press}, \text{snap}\}$ resolves to a signed per-side offset in millimetres. Two rules follow. First, both halves of a mate derive from the same parameter d plus a signed fit offset, never from two literals, so an edit to one side can never orphan the other. Second, each type fixes a degree-of-freedom signature, so a static mate fully locates its part while a kinematic mate leaves exactly the intended motion free.

3.2 Assembly-Graph Planning

Part plan with mates. Given the prompt, **Procedura** first synthesizes a single reference view, which grounds every later stage. A single vision call then decomposes the object into an ordered list of named

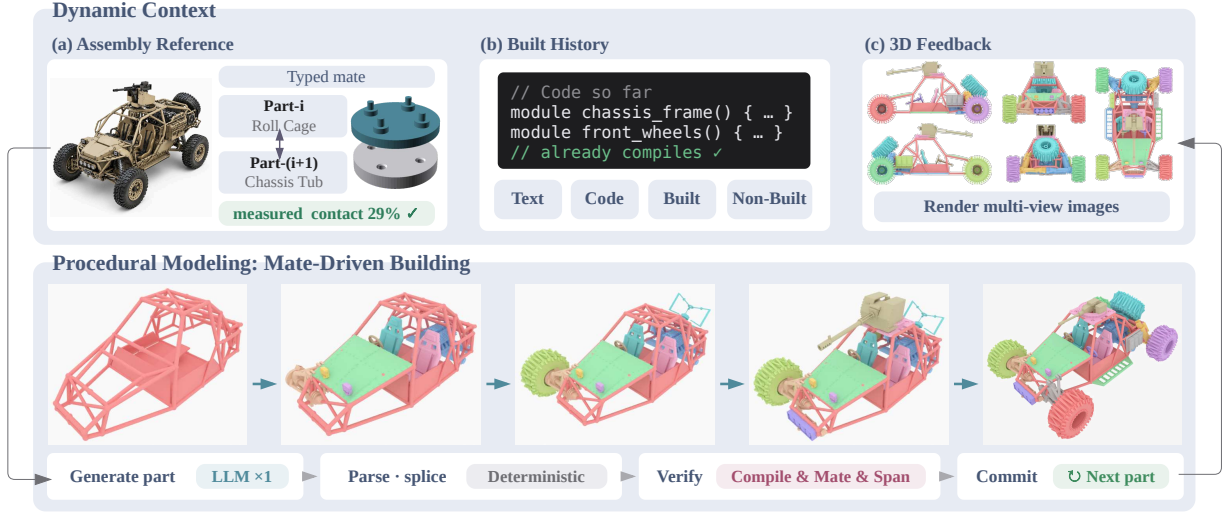


Figure 5. Mate-driven building. Top: for each planned part, the dynamic context packs three grounded sources into one generation message: (a) the assembly reference, namely the reference image, the prompt, and the part’s mate, shown here beside the verdict its mesh check returned; (b) the built history, the full and already-compiling program; and (c) 3D feedback, parts-colour renders of the build so far. Bottom: the program grows one part per mate, shown as five snapshots of the assault buggy at 2, 10, 18, 24, and 28 parts. Each part is generated in one LLM call and parsed into five blocks. The splice is deterministic and the placement is solved from the mated frames, and the compile, mate, and span gates must all pass before the part is committed. Every commit re-renders the 3D feedback for the next part.

parts. Each part carries a `snake_case` module name, a coarse detail level (silhouette, major feature, or sub-feature), and a short description that fixes its relative size, aspect, and pose against its neighbours. Each part also carries its interface declarations, the typed mates that bind it to parts placed earlier, each naming the partner, the mate type, and the shared nominal. The list is ordered so that every part attaches to one placed earlier. It is therefore a topological order of the assembly graph, which lets the build seat each module on geometry that already exists. Mirror pairs, such as a left and a right arm, are kept as separate parts, since a hand-assembled object is rarely perfectly symmetric and later stages edit each side independently.

One-direction plan review. A second vision call reviews the plan against the reference, and we allow it to revise in only one direction. It may add a missing part or sharpen a description or mate, but it may never merge, remove, rename, or reorder. The rule matters because each part is exactly one generation call in the build, so merging two parts into one removes detail the builder would otherwise produce. We enforce it in code instead of trusting the prompt, so every original part survives with its name and position, and only a genuinely new part is admitted, inserted after the part it attaches to. The reviewed graph is the plan the build consumes.

3.3 Mate-Driven Building

One part per call. A single LLM call can write the whole program for a simple shape. However, a detailed, many-part object forces that one call to fix every part, count, and placement at once, and the model loses track, leaving misplaced sub-assemblies and floaters. To address this, the build grows the program one module at a time, in plan order, so each call solves the small problem of adding a single part to a model that already exists. The call emits only the new part, in a strict five-block format: new parameters, helper modules, the part body, its published interface frames, and its placement intent. A simpler one-shot mode, which writes the entire program in a single call, is the control our experiments compare against (Sec. 4).

Dynamic context. Each generation call sees three grounded sources (Fig. 5). The assembly reference is the reference image, the text prompt, and the plan slice, namely the part to add, its mates, the parts not yet built, and a mandate to match the reference’s relative size, aspect, and pose. The built history is the full accumulated program, passed as text; because it already compiles, the model reads exact parameters and frames instead of guessing the current scale. The 3D feedback is a parts-colour render of the build from up to twelve viewpoints, six orthographic faces and six isometric corners. Each committed part carries a distinct colour, so the model can see where its part attaches. The first part, with nothing yet built, receives the reference and plan alone.

Solved placement. The model authors the part’s geometry and its mate frames but not the assembly transform. For a static mate joining the new part’s frame F_j to the placed partner frame F_i , the placement is the rigid transform that aligns them:

$$T_j = F_i \cdot \Delta(\phi) \cdot F_j^{-1}, \quad (1)$$

where $\Delta(\phi)$ denotes the signed fit offset along the mate axis. Solving T_j in code removes the largest single source of drift in prior 3D-code systems, since the model never emits a raw transform it estimated from a render. A part with several static mates is placed by the first and checked against the rest.

Deterministic splice. The five blocks are spliced into the accumulating program at fixed marker comments. Parameters are appended and de-duplicated, a module whose name collides is renamed, and the solved placement is inserted into the assembly. The LLM authors only the new part, so the program grows without the generator ever re-emitting or corrupting what earlier calls produced.

Verification stack. A spliced part must clear three deterministic checks before it joins the build. First, a compile gate compiles the program; on failure the compiler’s error text is fed back and the part is regenerated. Second, a mate gate measures each of the part’s static mates on the compiled mesh, testing the registration area $a(c)$ of the mated surfaces and the penetration depth $\delta(c)$ between the two bodies:

$$a(c) \geq \tau_a d^2 \quad \text{and} \quad \delta(c) \leq \delta_{\max}(\phi), \quad (2)$$

where d denotes the mate’s nominal and $\delta_{\max}$ the tolerance its fit class allows. A part that merely hovers near its partner, or gouges into it, is therefore rejected, with the measured values returned in the feedback. Third, a delta-aware connectivity gate reuses the span measure of Sec. 3.4 and rejects the part only when it increases the count of visible floaters over the running baseline, so a part that legitimately connects late is never blocked. Each part gets up to three quality attempts. When it exhausts them it is committed with a warning rather than dropped, since completion matters more than a defect the refine loop can still remove; a part that never compiles is skipped. When the plan is exhausted, the markers are stripped, yielding the draft assembly the refine loop takes up.

3.4 Decoupled-Critic Refinement

The draft assembly is complete but imperfect, so a refine loop polishes it. Each cycle renders the current build, has a critic diagnose it against the reference, and applies the top defect, followed by a verify compile. The defining choice is to decouple the critic from the fixer, so the agent repairs what a separate reviewer reports rather than what it judges of its own work.

Decoupled critic. The reviewer is a separate, single-turn vision call. It runs the same underlying model with its own system prompt and context, never as a turn of the agent that writes code. It sees the reference, the views the agent just rendered, the parts-colour legend, and the current program, and returns a one-line SUMMARY and a prioritized ISSUES list, each issue tagged [HIGH|MED|LOW] and naming the responsible module with a one-sentence fix direction. We decouple the two for three reasons. First, a model that judges its own

output carries the very assumptions that produced the error, so an independent reviewer is more likely to catch it. Second, the diagnosis becomes a persisted, first-class artifact, whereas in a freeform loop the critique lives only in hidden reasoning. Third, the critic is history-aware, receiving a trail of what prior cycles flagged, so it verifies that earlier fixes landed and flags regressions (Fig. 11).

Diagnosis-gated edits. The ordering is a hard invariant, enforced in code. A successful `diagnose` sets a one-shot latch that exactly one subsequent edit consumes. The edit tools refuse unless the latch is set, and `diagnose` itself refuses if the rendered views are stale with respect to the current program. Without the gate the agent edits repeatedly off a stale review; an ungated variant issued 15 edits against 2 diagnoses, whereas the gated loop holds the diagnose-to-edit ratio at $\approx 1:1$.

Agent-chosen context. The agent selects which views to render and how many (1 to 20) from a fixed catalog of orthographic faces, isometric corners, eye-level diagonals, and front tilts. On a fresh model it surveys broadly so no side hides a defect; once a problem is localized it renders the focused subset that best shows it. A refine step is bounded by a single module edit, and only edits count against the budget. After the edit, the agent compiles and, when the compile flags floaters, checks connectivity within the same cycle, so a broken edit is caught before it wastes the next review. The agent ends with `ok` when the critic reports no `HIGH` issue, or `give_up` when several edits fail to shrink the issue list.

Connectivity as a hard constraint. A faithful-looking render does not guarantee a printable object, since parts that merely abut compile to several disconnected solids even though they look joined. The mate gate of Sec. 3.3 prevents most of these by construction; the refine loop closes the rest with a terminal check on the mesh that actually ships.

Span-based gate. Let M be the compiled mesh. We weld coincident vertices and partition its triangles into connected components by union-find over shared edges:

$$C = \{c_1, \dots, c_K\} = \text{COMPONENTS}(M), \quad \rho_i = \frac{\text{span}(c_i)}{\text{span}(M)}, \quad (3)$$

where $\text{span}(\cdot)$ is the largest side of the axis-aligned bounding box. Let the body be the largest component, $b = \arg \max_i \text{vol}(c_i)$. A component c_i with $i \neq b$ is a visible floater if its span fraction ρ_i reaches a threshold τ , and the gate passes only when none remain:

$$N_{\text{vis}} = |\{i \neq b : \rho_i \geq \tau\}|, \quad \text{GATE}(M) = [N_{\text{vis}} = 0], \quad (4)$$

with $\tau = 0.01$.

A volume-based criterion instead misses the floaters that occur in practice. A thin, flat panel has near-zero enclosed volume yet a large span, so it passes a volume check while remaining clearly detached (Fig. 6). We measure by span so the metric tracks what a viewer and the critic actually see.

Gated finish. The terminal action `finish(ok)` is refused while $N_{\text{vis}} > 0$. Since the agent may edit after its last compile, the committed program is recompiled when stale, so the gate grades the mesh the user

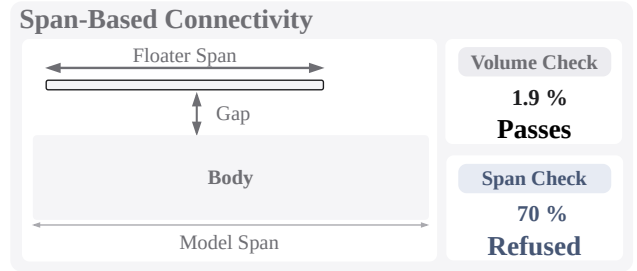


Figure 6. Span-based connectivity. A thin detached panel has near-zero enclosed volume, so a volume check passes and the floater ships; its bounding-box span covers most of the model and the defect is plainly visible. **Procedura** thresholds on span (Eq. (4)). Drawn to the worst case from our audit of a pre-gate pipeline: 70% span, 1.9% volume.

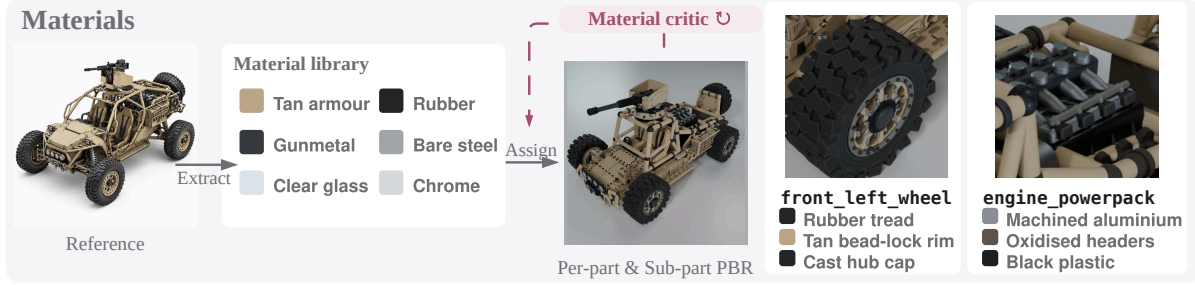


Figure 7. Per-part PBR materials from the reference. **Procedura** extracts a compact material library from the reference (25 entries for the assault buggy, six shown), assigns one entry to each of the 41 named parts, and a decoupled material critic corrects the assignment against the reference. A sub-part pass then splits a module into sibling colour blocks where one material is too coarse. Renders are Blender-Cycles PBR.

actually receives. When it refuses, it names the worst offenders with their bounding box so the next cycle has a concrete target. The `give_up` verdict is never gated, allowing a genuinely intentional separation to terminate.

3.5 Materials

Material library and assignment. Geometry alone does not make a production asset, so the same graph also carries a material for every part. Paint runs once the geometry is frozen and changes only colour and surface; the whole stage is four one-shot vision calls with no agent loop. **Procedura** first reads the reference image and catalogues every distinct surface as a compact material library, typically one to two dozen physically-based-rendering (PBR) entries with albedo, class, roughness, metalness, and wear. It then assigns one library material to each named part, so the palette is grounded in the reference and shared across parts made of the same thing. As with geometry, a separate vision critic compares a render of the painted model to the reference and returns a corrected assignment.

Sub-part colour blocks. One material per module is often too coarse, since one wheel module carries a rubber tread, a painted rim, and a cast hub. **Procedura** therefore rewrites a module into sibling colour blocks so a sub-feature takes its own material, recompiling each rewrite and rejecting any whose triangle count moves beyond a fixed tolerance, so colour can never alter the shape. The stage emits a colour-tagged program, a multi-group `OBJ+MTL` with per-part PBR fields, and a Blender-Cycles studio render (Fig. 7); every artifact stays parts-addressable.

3.6 Articulation

Turning the same parts into a jointed, physically validated asset requires no new structure, since the mate network is already the kinematic graph. Static mates group parts into rigid links. Each kinematic mate, whether revolute, prismatic, or spherical, names its joint, axis, and anchor, and is either authored at plan time or promoted later from geometric evidence.

Geometric evidence. **Procedura** measures the geometry before any reasoning about motion. For each part it computes two kinds of evidence: rotational-symmetry axes, taken from the area-weighted covariance of the part’s surface triangles, and contact regions between touching parts. Both are deterministic and yield concrete axis and anchor candidates, so a wheel’s spin axis or a hinge line starts from measured geometry.

Joint planning and validation. A vision call then completes the articulation. It groups modules into links,

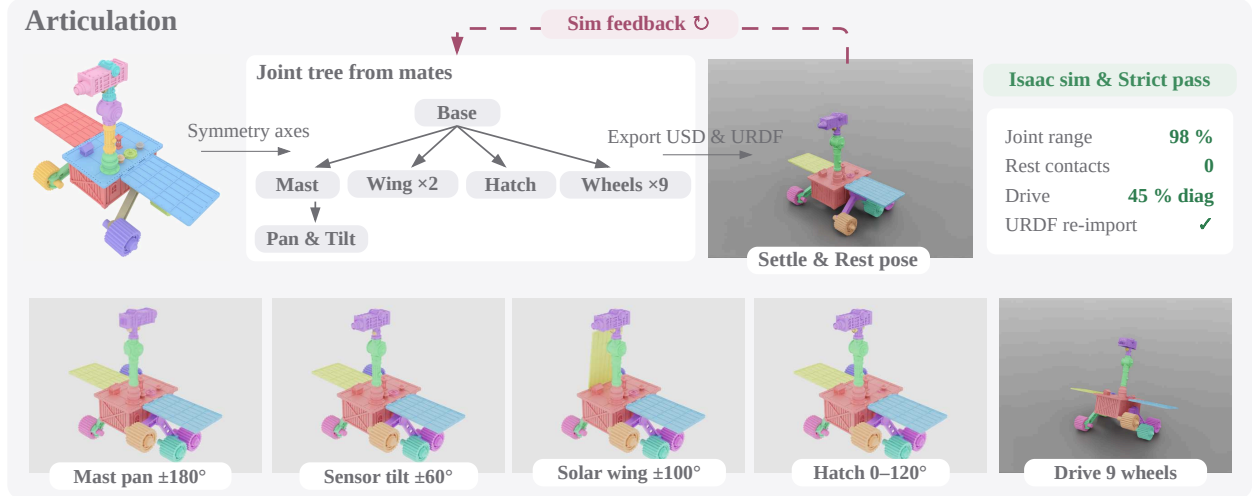


Figure 8. A sim-ready articulation validated in Isaac Sim. From the rover’s named parts, **Procedura** measures geometric evidence (the rotational-symmetry axes and contact regions) and plans the joint tree over the kinematic mates (17 links, 16 joints). It exports an OpenUSD articulation and a URDF and validates them headless in Isaac Sim; a failed verdict feeds rendered simulation frames back to the planner for one refine. Every panel is a real Isaac frame in per-link colours: the settled rest pose, four per-joint actuation sweeps, and the rover after a nine-wheel drive.

builds the parent–child tree over the kinematic mates, and assigns each joint its axes, limits, and drives. The plan is refined against deterministic feedback and exported as an OpenUSD articulation and a URDF, so the same object drops into a physics simulator or a robotics stack. **Procedura** then validates the plan headless in Isaac Sim through a fixed battery: a dynamic settle, per-joint actuation sweeps, a rest-contact scan, a wheeled mobility test, and a URDF re-import. A flagged fault feeds up to eight rendered simulation frames back to the planner for one refine, so the critic can see a wrong axis (Fig. 8).

4 Experiments

Our experiments answer three questions. Does mate-driven shape-as-code make a frozen LLM a state-of-the-art 3D generator, measured against native generators, prior 3D-code agents, and single-shot prompting of the same models (Secs. 4.2 and 4.3)? What do the pipeline’s stages contribute (Sec. 4.4)? And does the program deliver what a mesh cannot, namely a full production asset (Sec. 4.5)? Throughout we report intervals and paired tests rather than single numbers.

4.1 Setup and Protocol

Benchmarks. We evaluate on two benchmarks. The first is MechBench-36, our hard-surface benchmark, a frozen set of 36 deliberately many-part objects, rovers, mechs, cranes, excavators, engines, and landing gear, chosen because they stress part counts, placement, and connectivity far harder than a single-sub-assembly shape (Sec. 4.2). The second is the assembly task of P3D-Bench [48], 203 text-and-image cases scored by the benchmark’s own assembly judge (Sec. 4.3). **Procedura** normally synthesizes its own reference view from the prompt; for evaluation we lock both the prompt and that view and hand the same pair to every method, so image synthesis cannot confound the comparison.

Blind judging protocol. Judging generated 3D by eye invites bias, so the protocol removes each confound in turn. Every shipped model is re-rendered from one fixed view set under one grey-clay rig, assigned an

Method	Judge				Edge structure		
	Geo↑	Aes↑	Sem↑	Overall↑	CLIP↑	Shrp ₆₀ ↑	Dih ₉₅ ↑
Procedura (Gemini 3.7 Flash)	0.799	0.827	0.858	0.828	82.67	158.08	97.97
Procedura (GPT-5.6-sol)	0.756	0.793	0.843	0.797	81.76	185.18	105.28
<i>Agentic 3D-code</i>							
Adam CAD (GPT-5.6-sol) [1]	0.778	0.759	0.861	0.799	81.26	57.35	90.41
ArtiCraft (GPT-5.6-sol) [55]	0.682	0.596	0.827	0.702	79.12	41.31	94.74
CAD-Coder [10]	0.099	0.210	0.096	0.135	60.62	7.63	92.91
cadrille [21]	0.086	0.105	0.083	0.092	50.65	19.58	97.18
<i>Single-shot LLM</i>							
GPT-5.6-sol	0.790	0.735	0.852	0.792	80.91	60.39	89.76
Gemini 3.7 Flash	0.719	0.722	0.849	0.763	80.86	63.54	91.35
Gemini 3.1 Pro	0.698	0.673	0.824	0.731	79.94	—	—
<i>Native 3D generation</i>							
TRELLIS.2 [45]	0.781	0.787	0.861	0.810	77.51	134.02	73.11
UltraShape [15]	0.731	0.694	0.818	0.748	75.04	8.58	33.06
Hunyuan3D [37]	0.685	0.590	0.815	0.697	74.62	6.41	35.12
Direct3D-S2 [43]	0.583	0.559	0.775	0.639	74.90	35.17	52.78

Table 1. MechBench-36 (Figs. 9 and 10). Every method is re-rendered through one rig and scored by the same judge and CLIP backbone, so the only difference between rows is the method. Geo, Aes, and Sem are the judge’s geometry, aesthetics, and semantic axes, mapped per case to 0–1 by $\text{clamp}((v - 1)/9)$ as in Tab. 2; aesthetics rates the object as an engineered part, and Overall is their mean. CLIP is prompt-to-render alignment on the shared renders. Shrp₆₀ is total sharp-edge length above a 60° dihedral and Dih₉₅ the 95th-percentile dihedral, both measured on the shipped mesh. The 60° threshold keeps only genuine machined creases, since a shallower one also counts the ridges of a regressed surface. Cases where a method shipped no mesh score zero and stay in the mean, which is why CAD-Coder and cadrille sit low. Single-shot rows are one call with no plan, build, or refine stage. Best per column in bold.

opaque hash, and judged blind, so method names never reach the judge and render settings cannot skew a row. A vision LLM [44] scores each render with one shared prompt on three axes, semantic fidelity, geometric quality, and aesthetics, the last rating the object as an engineered part. On MechBench-36 the judge sees no ground truth, since none exists for a prompt; on P3D-Bench it follows the benchmark’s own protocol. Alongside the judge we report CLIP, the image-text alignment between the prompt and the shared grey-clay renders, and two edge measurements on the shipped mesh, sharp-edge length and the 95th-percentile dihedral (Tab. 1). All are computed under the one shared rig, so they are comparable across rows but not across papers.

Models and implementation. Unless noted, planning, building, and the critic all use one frozen deep-reasoning LLM, Gemini 3.7 Flash, with no 3D-specific training; Tab. 2 additionally reports a GPT-5.6-sol variant of the full pipeline. The judge is Gemini 3.7 Flash in its high-reasoning mode, blinded by hash. **Procedura** is built on an in-house LLM harness; the program is realized as OpenSCAD and compiled by the OpenSCAD engine, with module parsing, splicing, the mesh parser, and the union-find connectivity analysis in process. Beyond the compiler, the only external binaries are Blender for renders and a headless Isaac Sim for physical validation.

4.2 Results on MechBench-36

MechBench-36 concentrates the many-part regime that motivates **Procedura**. As shown in Tab. 1, **Procedura** takes the top judge composite at 0.828 with Gemini 3.7 Flash, leads geometry, aesthetics, and CLIP, and ships a mesh for every case. TRELLIS.2 follows at 0.810; Adam CAD at 0.799 and our GPT-5.6-sol arm at 0.797 are effectively tied.

Axis-level analysis. The judge’s axes do not separate the field equally. Semantic spans only 0.775 to 0.861 across the methods that work at all, since that axis mostly asks whether the object is of the right kind; Adam CAD and TRELLIS.2 share its top at 0.861, a whisker above our 0.858. Geometry and aesthetics separate the methods far more, spanning 0.583 to 0.799 and 0.105 to 0.827, and **Procedura** tops both.

Comparison with native generators. Native generators map an image to a dense surface whose creases come out rounded. The strongest of them, TRELLIS.2, trails by 5.2 CLIP points, and Direct3D-S2 and Hunyuan3D fall behind on aesthetics by 0.27 and 0.24. In Fig. 10, the natives reproduce the silhouette faithfully but blur bolt heads, vents, and panel lines into the surrounding body, whereas our compiled CSG keeps every feature crisp and every part separable.

Comparison with prior 3D-code agents. The strongest prior agent, Adam CAD on GPT-5.6-sol, reaches 0.759 aesthetics and 81.26 CLIP. The domain-tuned coders CAD-Coder and cadrille collapse on many-part objects, scoring 0.210 and 0.105 and shipping no mesh on 1 and 5 cases, since a model trained to emit one sketch-extrude body cannot express an assembly. The same failure is visible in Fig. 9, where the prior agents keep the outline of each machine but simplify its mechanism into a few large blocks.

Edge structure. The two right-hand columns measure the edge structure each representation produces, and **Procedura** leads both. Our programs carry 158.1 units of sharp edge above a 60° dihedral with Gemini 3.7 Flash and 185.2 with GPT-5.6-sol, $2.6\times$ and $3.1\times$ the next code-emitting method, at 95th-percentile dihedrals of 98.0° and 105.3° . The native generators sit at $33\text{--}73^\circ$, since a surface regressed from samples has no notion of a face or a bore. TRELLIS.2 posts the only comparable sharp-edge length at 134.0, yet its 73.1° dihedral shows those creases are far shallower than ours. The gap follows from the representation, since an edge in compiled CSG is the exact intersection of two analytic surfaces.

Paired gain over the base model. Every row uses a frozen model, so the pipeline can be ablated against a single call to the same model on the same case. Paired per object over the whole benchmark, the pipeline improves aesthetics by +0.105 on Gemini 3.7 Flash (20 wins, 7 losses, sign test $p = 0.019$); on GPT-5.6-sol the mean gain is +0.059 but the split is 15 wins to 11 losses ($p = 0.56$), so we do not claim significance there. CLIP improves by +1.80 ($p = 0.011$) and +0.85 respectively. The gain concentrates on aesthetics, the axis the mate-driven build targets.

4.3 Results on P3D-Bench

P3D-Bench’s judge scores each case on the three axes of Sec. 4.1, and its composite is the mean of the three after normalization. We re-render and re-judge every method under the shared harness, so every row of Tab. 2 is directly comparable.

As shown in Tab. 2, **Procedura** solves all 203 cases with either base model and leads every column. The composite reaches 0.590 on Gemini 3.7 Flash and 0.575 on GPT-5.6-sol, against 0.524 for the strongest entry that also covers the full benchmark, a margin several times the 95% intervals. Notably, the lead is largest on geometry, 0.490 against 0.392, which is exactly where the mate-driven build acts, since solved placement and verified connectivity are geometric properties.

Gain over single-shot prompting. Prompting our two base models once reaches 0.566 and 0.563 on the reduced sets those runs cover, above every published entry, so one call to a frontier model already clears the published leaderboard. Part of any margin over the published rows therefore reflects base-model strength rather than method; the agent supplies the remaining margin. A per-case paired comparison puts the full pipeline above the Gemini 3.7 Flash control on the cases it covers, and the pipeline also solves the cases the control does not.

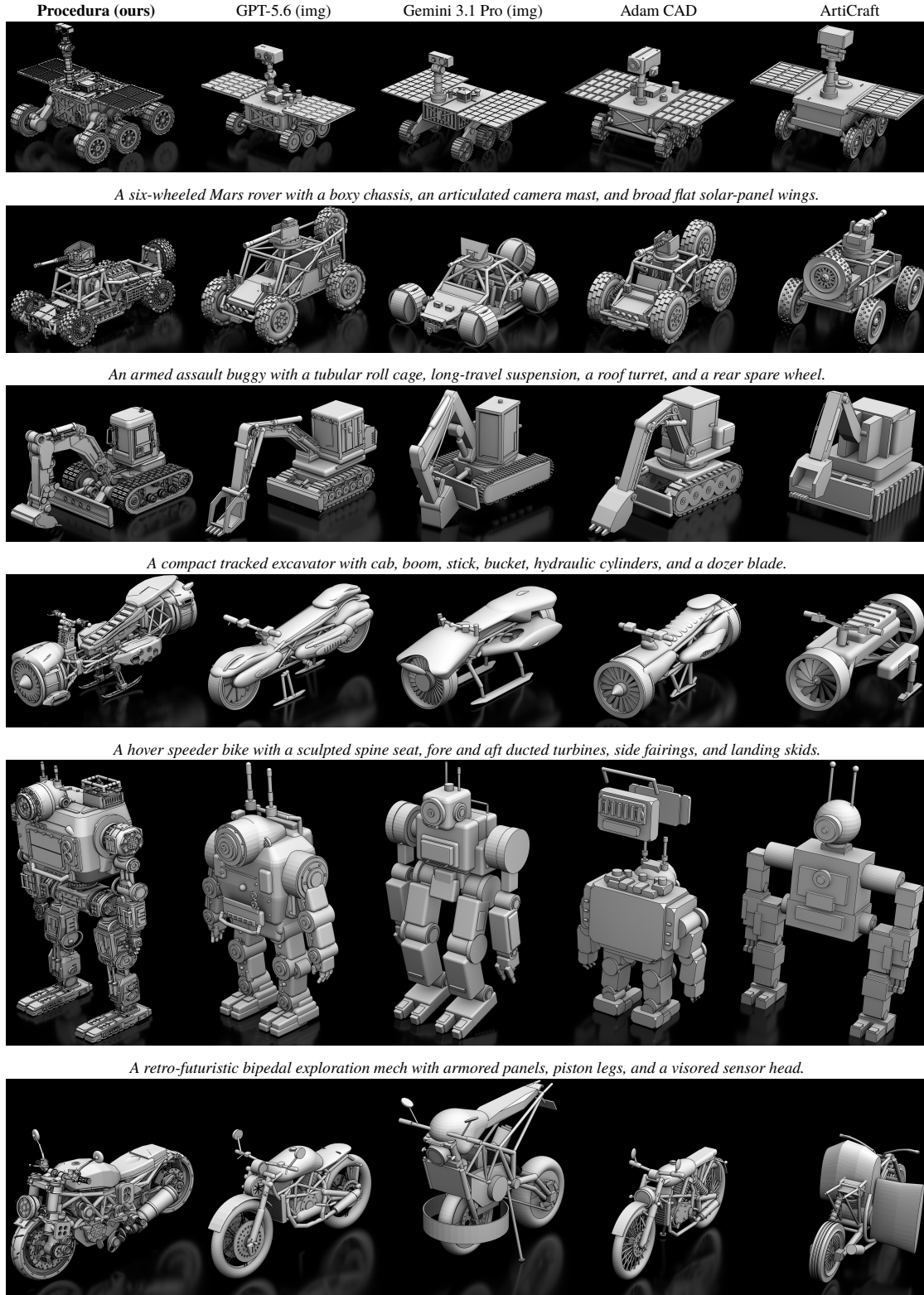


Figure 9. Qualitative comparison against LLM-driven 3D generation. Six benchmark objects, one per row, each generated from the prompt printed beneath it; every method receives the same reference image and render rig. Prior agents recover the rough silhouette but collapse the mechanism into a handful of boxes, whereas **Procedura** resolves each part as its own solid.

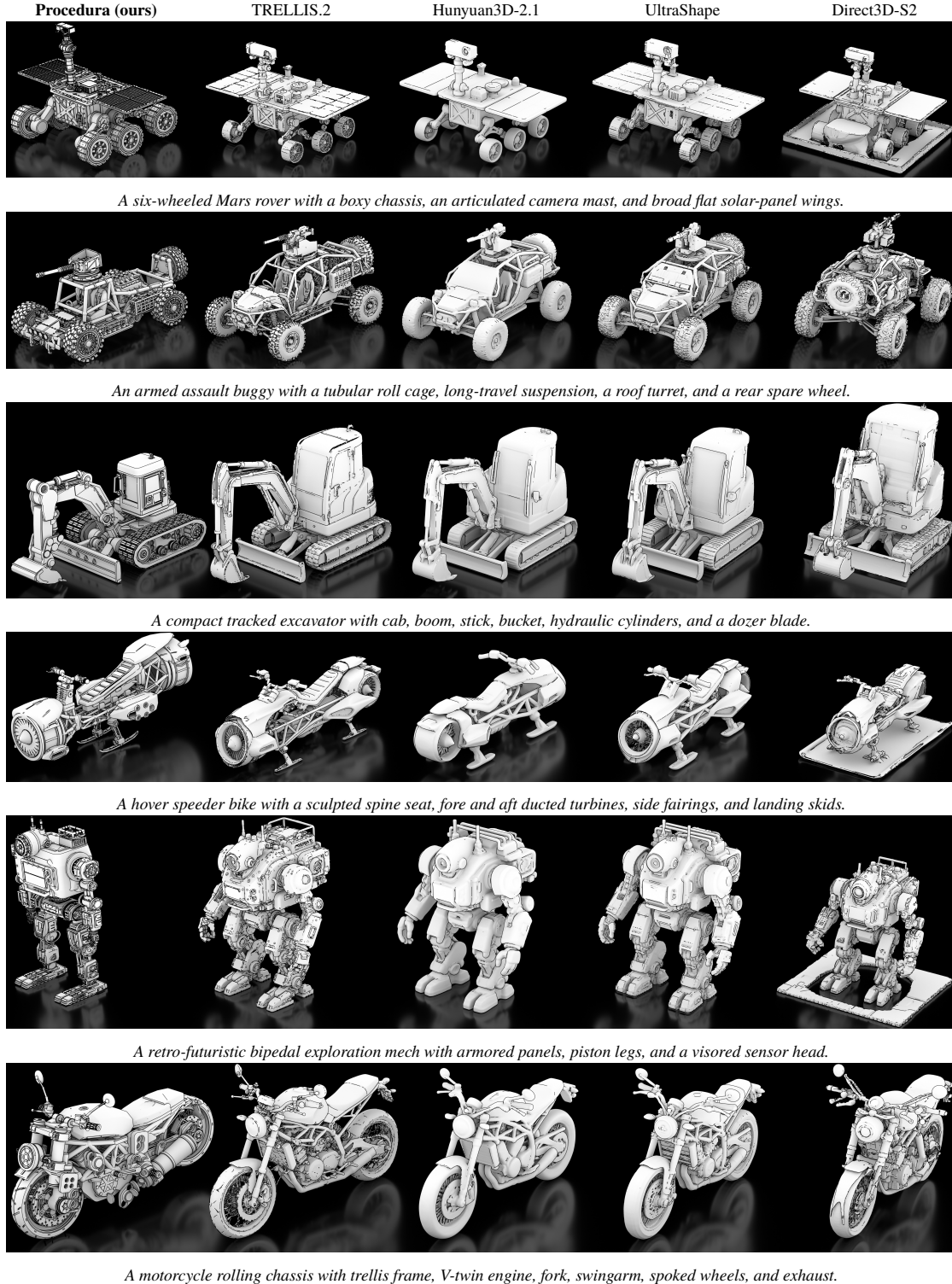


Figure 10. Qualitative comparison against native 3D generators. The same six objects, one per row with its prompt printed beneath it, generated by four native 3D diffusion models and by **Procedura** (left column), all conditioned on the same locked reference image and re-rendered from the same camera and shading rig. The natives reproduce the silhouette faithfully, since they are trained end-to-end to do so, but their surfaces are soft, so bolt heads, vents, and panel lines fuse into one skin and no part boundary survives. **Procedura** compiles exact CSG solids, so edges stay sharp and every part remains separable.

Method	Semantic $\uparrow$	Geometry $\uparrow$	Aesthetics $\uparrow$	Judge (Overall) $\uparrow$
Procedura (Gemini 3.7 Flash)	0.666	0.490	0.616	0.590 ± 0.009
Procedura (GPT-5.6-sol)	0.661	0.460	0.604	0.575 ± 0.010
Gemini 3.7 Flash	0.662	0.458	0.577	0.566 ± 0.016
GPT-5.6-sol	0.662	0.444	0.584	0.563 ± 0.015
GPT-5.5	0.643	0.392	0.537	0.524 ± 0.012
Gemini 3.1 Pro	0.639	0.390	0.508	0.513 ± 0.012
Opus 4.6	0.572	0.327	0.446	0.448 ± 0.016
Kimi K2.6	0.523	0.294	0.372	0.396 ± 0.014
Qwen 3.6	0.451	0.264	0.322	0.346 ± 0.014
Doubao	0.408	0.253	0.302	0.321 ± 0.014
GLM-5V Turbo	0.402	0.249	0.300	0.317 ± 0.013
MiMo Omni	0.379	0.242	0.294	0.305 ± 0.012

Table 2. P3D-Bench assembly results. Every method is re-rendered from the same rig and scored by the benchmark’s assembly judge on its three axes, semantic fidelity, geometric quality, and aesthetics, normalized to 0–1 by the benchmark’s mapping; Judge (Overall) is their mean, the benchmark’s composite, with 95% intervals. Baselines are the benchmark’s entries, re-run under the same harness, plus single-shot prompting of the two base models **Procedura** uses.

Variant	Geo $\uparrow$	Aes $\uparrow$	Sem $\uparrow$	Judge $\uparrow$	CLIP $\uparrow$
Procedura (full)	0.799	0.827	0.858	0.828	82.67
w/o refine	0.762	0.793	0.846	0.800	81.84
w/o planning	0.756	0.775	0.843	0.791	82.36
w/o 3D feedback	0.790	0.815	0.852	0.819	81.96

Table 3. Ablations on MechBench-36. All variants use the same base model and are judged in the same single session as Tab. 1, with the same normalization ($n = 36$). Ablated variants are evaluated at the draft stage, before refine, so the planning and 3D-feedback deltas also include the absence of the refine stage.

4.4 Ablation Study

Tab. 3 removes the refine loop, the assembly planning, or the per-part 3D feedback and re-judges the 36 cases under the same regime as Tab. 1. Every removal costs quality on every column, and the full pipeline is the best variant on all five. Dropping the refine loop lowers the composite from 0.828 to 0.800, the cleanest read of what the decoupled critic’s gated edits add, since that variant differs from the full pipeline by refine alone. Dropping the planning stage costs the most, 0.791 overall with the largest aesthetics drop, so the assembly graph the build consumes is load-bearing. Dropping the per-part 3D feedback lowers every judge axis and CLIP as well. The deficit grows with how much of the design is removed, consistent with each stage contributing to the final quality.

Refinement case study. Fig. 11 traces the refine loop on one of the hardest benchmark cases. The critic’s complaints are quantitative and structural. It reports the front winch $1.25\times$ a road-tire diameter ahead of the front axle against roughly $0.6\text{--}0.7\times$ in the reference, and a gun receiver at $0.19\times$ the tire diameter against $0.30\text{--}0.35\times$, and the agent answers each with a single tool call on the named modules. Over three cycles the `HIGH` list shrinks monotonically, and the history-aware critic carries context forward, with cycle 3 opening on the rear equipment stack that “remains too low”. The case ends on `max-steps` rather than converging, since one gated fix per fresh diagnosis makes progress linear in the cycle budget. One of the three edits is a `snap_floaters` call that closes sub-millimetre gaps, changing connectivity without changing the silhouette,

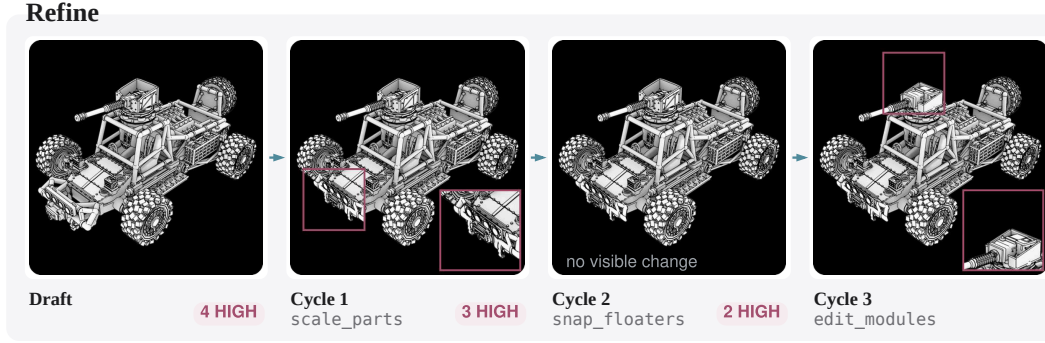


Figure 11. What each gated edit changes. The assault buggy, refined from the draft the build stage produced. Every frame is rendered from one fixed camera, so a difference between frames is a difference in geometry; the inset magnifies the densest region of pixels that cycle’s edit changed, and the outline on the model marks where the inset is taken from. Each cycle is one fresh diagnosis followed by one gated edit; under every state we print the tool that produced it and the number of HIGH issues the critic reported for that state. The count falls $4 \rightarrow 3 \rightarrow 2$; the final state carries no count because the cycle budget ends before a further diagnosis. Cycle 2 is a sub-millimetre contact snap that changes connectivity but nothing a reader could see, and the frame says so.

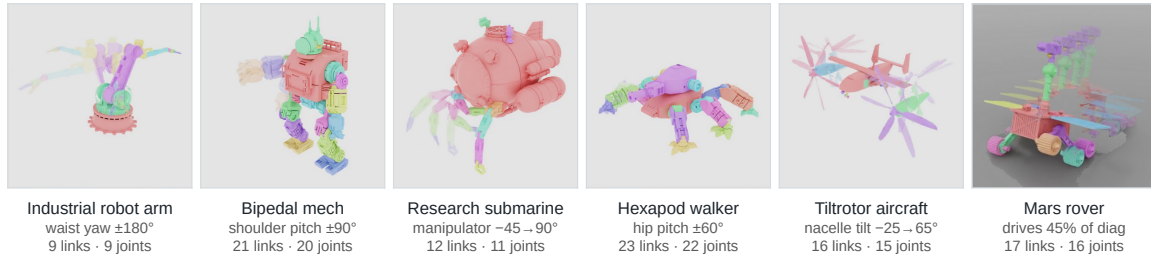


Figure 12. Articulation across the benchmark. Six objects, each planned, exported to OpenUSD and URDF, and validated headless in Isaac Sim. Every panel is composited from that object’s own simulation clip, with the swept poses ghosted and the final pose solid, so a still carries the motion. The first five hold one commanded joint traversing its range; the last shows the wheeled base driving on the ground plane. All six pass every dynamics phase: each position-driven joint traverses $\geq 97\%$ of its commanded range, no non-adjacent links touch at rest, and the URDF re-imports. Five also clear the asset rules outright; the tiltrotor trips the advisory drive-gain rule, the same rule two earlier rover plans tripped.

which is why the cycle-2 frame looks unchanged and why we report the issue list rather than a pixel difference as the outcome.

4.5 Materials and Articulation

Materials. Because every part is a named module, a physically-based material is a per-part attribute the program simply carries. Fig. 7 shows an object painted from its reference alone, where the extract, assign, critic, and sub-part passes recover distinct metals, rubber, and glass, and Fig. 13 shows the same stage across the benchmark. We report no paint metric, since there is no ground-truth material map and a colour distance against a generated reference would score the reference rather than the model.

Articulation. Articulation, in contrast, can be checked objectively, since the physics simulator is an independent oracle. Across the 18 objects we have articulated and validated, 14 clear every dynamics phase – settle, per-joint actuation, rest-contact scan, and URDF re-import – and 9 of those also clear the advisory asset rules; the four that fail outright are one asset that would not load and three whose actuators never traverse (Fig. 12).



Figure 13. Texturing across the benchmark. Ten of the 36 benchmark objects, geometry and materials both authored by **Procedura**. Every image is a Blender-Cycles render of the shipped program, with per-part materials assigned to the named modules.

5 Limitations

Procedura produces editable procedural assemblies that surpass native generators on judged quality, yet three limitations remain. First, the agent perceives the build only through rendered images. A rendered view conveys silhouette and placement well, but occluded interiors and fine contact geometry can hide from every camera; depth maps, cross-sections, or direct mesh queries would strengthen the loop. Second, CSG captures mechanical, hard-surface objects precisely, whereas organic, freeform surfaces are expressed less naturally than in a neural field. Third, the mate vocabulary covers rigid joints only, and a mate the plan never declares cannot be verified.

6 Conclusion

We presented **Procedura**, a procedural assembly agent that generates a 3D object as the program that builds it, a parametric assembly whose named parts are joined by typed, machine-checkable mates. A frozen LLM authors this representation with no 3D training, planning the assembly graph, building the program part by part with placement solved from the mated frames and every commit verified, and refining the result through a decoupled vision critic. The same graph then carries per-part materials and a simulator-validated articulation. On P3D-Bench and on MechBench-36, judged blind, **Procedura** leads every native generator and prior 3D-code agent on judged quality, carries the sharpest edges of any method we measure, and is the only system whose output re-opens as an editable, part-structured program. Its principles carry to program-based generation at large: casting shape generation as coding, joining parts by verifiable mates, separating the critic from the fixer, and validating every claim in a compiler or a simulator.

References

- [1] Adam CAD. Adam: An AI cad copilot for hardware teams. <https://adam.new/>, 2026. Commercial product; accessed 2026-07-19.
- [2] Marco Attene, Marcel Campen, and Leif Kobbelt. Polygon mesh repairing: An application perspective. *ACM Computing Surveys*, 45(2), 2013.
- [3] Jesse Barkley, Rumi Loghmani, and Amir Barati Farimani. CadSmith: Multi-agent cad generation with programmatic geometric validation. *arXiv preprint arXiv:2603.26512*, 2026.
- [4] Geoffrey Boothroyd, Peter Dewhurst, and Winston A. Knight. *Product Design for Manufacture and Assembly*. CRC Press, 3rd edition, 2010.
- [5] Yiwen Chen, Tong He, Di Huang, Weicai Ye, Sijin Chen, Jiaxiang Tang, Xin Chen, Zhongang Cai, Lei Yang, Gang Yu, Guosheng Lin, and Chi Zhang. MeshAnything: Artist-created mesh generation with autoregressive transformers. In *International Conference on Learning Representations (ICLR)*, 2025.
- [6] Matt Deitke, Eli VanderBilt, Alvaro Herrasti, Luca Weihs, Jordi Salvador, Kiana Ehsani, Winson Han, Eric Kolve, Ali Farhadi, Aniruddha Kembhavi, and Roozbeh Mottaghi. ProcTHOR: Large-scale embodied AI using procedural generation. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.
- [7] Fengxiao Fan, Jingzhe Ni, Xiaolong Yin, Sirui Wang, Xingyu Lu, Qiang Zou, Ruofeng Tong, Min Tang, and Peng Du. CADDesigner: Conceptual cad model generation with a general-purpose agent. *arXiv preprint arXiv:2508.01031*, 2025.
- [8] Weixi Feng, Wanrong Zhu, Tsu-Jui Fu, Varun Jampani, Arjun Akula, Xuehai He, Sugato Basu, Xin Eric Wang, and William Yang Wang. LayoutGPT: Compositional visual planning and generation with large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.
- [9] Gemini Team. Gemini: A family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.
- [10] 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. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.
- [11] Yicong Hong, Kai Zhang, Jiuxiang Gu, Sai Bi, Yang Zhou, Difan Liu, Feng Liu, Kalyan Sunkavalli, Trung Bui, and Hao Tan. LRM: Large reconstruction model for single image to 3d. In *International Conference on Learning Representations (ICLR)*, 2024.
- [12] 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 *International Conference on Machine Learning (ICML)*, 2024.
- [13] Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *International Conference on Learning Representations (ICLR)*, 2024.
- [14] Jingwei Huang, Yichao Zhou, and Leonidas Guibas. ManifoldPlus: A robust and scalable watertight manifold surface generation method for triangle soups. *arXiv preprint arXiv:2005.11621*, 2020.
- [15] Tanghui Jia, Dongyu Yan, Dehao Hao, Yang Li, Kaiyi Zhang, Xianyi He, Lanjiong Li, Yuhan Wang, Jinnan Chen, Lutao Jiang, Qishen Yin, Long Quan, Ying-Cong Chen, and Li Yuan. UltraShape 1.0: High-fidelity 3d shape generation via scalable geometric refinement. *arXiv preprint arXiv:2512.21185*, 2025.
- [16] Benjamin Jones, Dalton Hildreth, Duowen Chen, Ilya Baran, Vladimir G. Kim, and Adriana Schulz. Automate: A dataset and learned model for fixturing and mating in CAD assemblies. *ACM Transactions on Graphics (SIGGRAPH Asia)*, 40(6), 2021.

-
- [17] 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 Transactions on Graphics (SIGGRAPH Asia)*, 39(6), 2020.
 - [18] Heewoo Jun and Alex Nichol. Shap-E: Generating conditional 3d implicit functions. *arXiv preprint arXiv:2305.02463*, 2023.
 - [19] Kacper Kania, Maciej Zięba, and Tomasz Kajdanowicz. UCSG-Net: Unsupervised discovering of constructive solid geometry tree. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.
 - [20] Mohammad Sadil Khan, Sankalp Sinha, Talha Uddin Sheikh, Didier Stricker, Sk Aziz Ali, and Muhammad Zeshan Afzal. Text2CAD: Generating sequential cad designs from beginner-to-expert level text prompts. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.
 - [21] Maksim Kolodiaznyi, Denis Tarasov, Dmitrii Zhemchuzhnikov, Alexander Nikulin, Ilya Zisman, Anna Vorontsova, Anton Konushin, Vladislav Kurenkov, and Danila Rukhovich. cadrille: Multi-modal cad reconstruction with reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2026.
 - [22] Long Le, Jason Xie, William Liang, Hung-Ju Wang, Yue Yang, Yecheng Jason Ma, Kyle Vedder, Arjun Krishna, Dinesh Jayaraman, and Eric Eaton. Articulate-anything: Automatic modeling of articulated objects via a vision-language foundation model. In *International Conference on Learning Representations (ICLR)*, 2025.
 - [23] Chen-Hsuan Lin, Jun Gao, Luming Tang, Towaki Takikawa, Xiaohui Zeng, Xun Huang, Karsten Kreis, Sanja Fidler, Ming-Yu Liu, and Tsung-Yi Lin. Magic3D: High-resolution text-to-3d content creation. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023.
 - [24] Minghua Liu, Mikaela Angelina Uy, Donglai Xiang, Hao Su, Sanja Fidler, Nicholas Sharp, and Jun Gao. Part-Field: Learning 3d feature fields for part segmentation and beyond. *arXiv preprint arXiv:2504.11451*, 2025.
 - [25] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-Refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.
 - [26] Liane Makatura, Michael Foshey, Bohan Wang, Felix Hähnlein, Pingchuan Ma, Bolei Deng, Megan Tjandra-suwita, Andrew Spielberg, Crystal Elaine Owens, Peter Yichen Chen, et al. How can large language models help humans in design and manufacturing? *arXiv preprint arXiv:2307.14377*, 2023.
 - [27] Charlie Nash, Yaroslav Ganin, S. M. Ali Eslami, and Peter W. Battaglia. PolyGen: An autoregressive generative model of 3d meshes. In *International Conference on Machine Learning (ICML)*, 2020.
 - [28] Alex Nichol, Heewoo Jun, Prafulla Dhariwal, Pamela Mishkin, and Mark Chen. Point-E: A system for generating 3d point clouds from complex prompts. *arXiv preprint arXiv:2212.08751*, 2022.
 - [29] OpenAI. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.
 - [30] Ben Poole, Ajay Jain, Jonathan T. Barron, and Ben Mildenhall. DreamFusion: Text-to-3d using 2d diffusion. In *International Conference on Learning Representations (ICLR)*, 2023.
 - [31] 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. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023.
 - [32] Gopal Sharma, Rishabh Goyal, Difan Liu, Evangelos Kalogerakis, and Subhansu Maji. CSGNet: Neural shape parser for constructive solid geometry. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.
 - [33] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

-
- [34] Yuan Shui, Yandong Guan, Zhanwei Zhang, Juncheng Hu, Jing Zhang, Dong Xu, and Qian Yu. ArtiCAD: Articulated cad assembly design via multi-agent code generation. *arXiv preprint arXiv:2604.10992*, 2026.
 - [35] Yawar Siddiqui, Antonio Alliegro, Alexey Artemov, Tatiana Tommasi, Daniele Sirigatti, Vladislav Rosov, Angela Dai, and Matthias Nießner. MeshGPT: Generating triangle meshes with decoder-only transformers. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024.
 - [36] Jiaxiang Tang, Zhaoxi Chen, Xiaokang Chen, Tengfei Wang, Gang Zeng, and Ziwei Liu. LGM: Large multi-view gaussian model for high-resolution 3d content creation. In *European Conference on Computer Vision (ECCV)*, 2024.
 - [37] Team Hunyuan3D. Hunyuan3D 2.1: From images to high-fidelity 3d assets with production-ready PBR material. *arXiv preprint arXiv:2506.15442*, 2025.
 - [38] Zhengyi Wang, Cheng Lu, Yikai Wang, Fan Bao, Chongxuan Li, Hang Su, and Jun Zhu. ProlificDreamer: High-fidelity and diverse text-to-3d generation with variational score distillation. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.
 - [39] Karl D. D. Willis, Pradeep Kumar Jayaraman, Hang Chu, Yunsheng Tian, Yifei Li, Daniele Grandi, Aditya Sanghi, Linh Tran, Joseph G. Lambourne, Armando Solar-Lezama, and Wojciech Matusik. JoinABLE: Learning bottom-up assembly of parametric CAD joints. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.
 - [40] Karl D.D. Willis, Yewen Pu, Jieliang Luo, Hang Chu, Tao Du, Joseph G. Lambourne, Armando Solar-Lezama, and Wojciech Matusik. Fusion 360 Gallery: A dataset and environment for programmatic cad construction from human design sequences. *ACM Transactions on Graphics (SIGGRAPH)*, 40(4), 2021.
 - [41] Rundi Wu, Chang Xiao, and Changxi Zheng. DeepCAD: A deep generative network for computer-aided design models. In *IEEE/CVF International Conference on Computer Vision (ICCV)*, 2021.
 - [42] Shuang Wu, Youtian Lin, Feihu Zhang, Yifei Zeng, Jingxi Xu, Philip Torr, Xun Cao, and Yao Yao. Direct3D: Scalable image-to-3d generation via 3d latent diffusion transformer. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.
 - [43] Shuang Wu, Youtian Lin, Feihu Zhang, Yifei Zeng, Yikang Yang, Yajie Bao, Jiachen Qian, Siyu Zhu, Xun Cao, Philip Torr, and Yao Yao. Direct3D-S2: Gigascale 3d generation made easy with spatial sparse attention. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.
 - [44] Tong Wu, Guandao Yang, Zhibing Li, Kai Zhang, Ziwei Liu, Leonidas Guibas, Dahua Lin, and Gordon Wetzstein. GPT-4V(ision) is a human-aligned evaluator for text-to-3d generation. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024.
 - [45] Jianfeng Xiang, Xiaoxue Chen, Sicheng Xu, Ruicheng Wang, Zelong Lv, Yu Deng, Hongyuan Zhu, Yue Dong, Hao Zhao, Nicholas Jing Yuan, and Jiaolong Yang. TRELLIS.2: Native and compact structured latents for 3d generation. *arXiv preprint arXiv:2512.14692*, 2025.
 - [46] Jianfeng Xiang, Zelong Lv, Sicheng Xu, Yu Deng, Ruicheng Wang, Bowen Zhang, Dong Chen, Xin Tong, and Jiaolong Yang. TRELLIS: Structured 3d latents for scalable and versatile 3d generation. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2025.
 - [47] Xiang Xu, Karl D.D. Willis, Joseph G. Lambourne, Chin-Yi Cheng, Pradeep Kumar Jayaraman, and Yasutaka Furukawa. SkexGen: Autoregressive generation of cad construction sequences with disentangled codebooks. In *International Conference on Machine Learning (ICML)*, 2022.
 - [48] Yikang Yang, Zhanpeng Hu, Youtian Lin, Mengqi Zhou, Jingxi Xu, Feihu Zhang, Jiaheng Liu, and Yao Yao. P3D-Bench: Benchmarking MLLMs for parametric 3d generation and structural reasoning. *arXiv preprint arXiv:2606.11152*, 2026.

-
- [49] Yue Yang, Fan-Yun Sun, Luca Weihs, Eli VanderBilt, Alvaro Herrasti, Winson Han, Jiajun Wu, Nick Haber, Ranjay Krishna, Lingjie Liu, Chris Callison-Burch, Mark Yatskar, Aniruddha Kembhavi, and Christopher Clark. Holodeck: Language guided generation of 3D embodied AI environments. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024.
 - [50] Yunhan Yang, Yuan-Chen Guo, Yukun Huang, Zi-Xin Zou, Zhipeng Yu, Yangguang Li, Yan-Pei Cao, and Xihui Liu. HoloPart: Generative 3d part amodal segmentation. *arXiv preprint arXiv:2504.07943*, 2025.
 - [51] Yunhan Yang, Yukun Huang, Yuan-Chen Guo, Liangjun Lu, Xiaoyang Wu, Edmund Y. Lam, Yan-Pei Cao, and Xihui Liu. SAMPart3D: Segment any part in 3d objects. *arXiv preprint arXiv:2411.07184*, 2024.
 - [52] Zhengyuan Yang, Jianfeng Wang, Linjie Li, Kevin Lin, Chung-Ching Lin, Zicheng Liu, and Lijuan Wang. Idea2Img: Iterative self-refinement with GPT-4V for automatic image design and generation. In *European Conference on Computer Vision (ECCV)*, 2024.
 - [53] Haocheng Yuan, Jing Xu, Hao Pan, Adrien Bousseau, Niloy J. Mitra, and Changjian Li. CADTalk: An algorithm and benchmark for semantic commenting of cad programs. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2024.
 - [54] Zibo Zhao, Wen Liu, Xin Chen, Xianfang Zeng, Rui Wang, Pei Cheng, Bin Fu, Tao Chen, Gang Yu, and Shenghua Gao. Michelangelo: Conditional 3d shape generation based on shape-image-text aligned latent representation. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.
 - [55] 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.