TreeStruct3D: Enabling Structural Editability in Agentic Procedural 3D Modeling

Anonymous 3DV submission

Paper ID 260

Abstract

Recent text-to-3D systems produce visually plausible assets, but their fixed outputs often omit the construction histories and part dependencies needed for structured editing. We introduce TreeStruct3D, a structure-conditioned extension of 3DCodeBench that generates structurally editable Blender programs. A planning VLM predicts a Part–Attachment Tree, and a code-generation VLM realizes each parent–child relation with paired, geometry-dependent anchors. After either endpoint changes, the program recomputes both anchors and realigns the child. A structure-aware validator independently edits parent and child parameters, localizes broken attachments, and returns targeted diagnostics for iterative repair. Across four VLMs, TreeStruct3D substantially improves end-to-end editability and preserves surface contact under held-out parent- and child-side edits, while changes in conventional visual and geometric fidelity remain model and metric dependent. These results move procedural generation toward assets whose part relations remain testable and editable after controlled modifications.

1. Introduction

Games, films, and virtual environments require large and diverse collections of 3D assets [5, 6, 28, 29]. Recent text- and image-to-3D systems, including Meshy, Tripo, and Tencent Hunyuan3D, demonstrate the potential of automatic creation [22, 34, 42] by directly generating holistic or part-based 3D mesh objects. However, their fixed geometric outputs typically lack explicit construction histories with semantic part relations and parameter dependencies, making them difficult to edit, validate, and adapt in a parametrized way.

Procedural code exposes the construction process of 3D objects, supporting deterministic execution and parameter-controlled variation, while recent advances in vision-language models and coding agents make its automatic generation increasingly practical [9]. General-purpose coding agents such as Claude Code, OpenAI Codex, and Gemini CLI can inspect project context, generate and execute code, invoke tools, and incorporate visual feedback [2, 11, 26]. These capabilities now extend to content-creation software [2, 4]. Complementing these general-purpose agents, 3D-focused research systems such as 3D-GPT, SceneCraft, BlenderLLM, and LL3M develop specialized pipelines for planning, executing, and visually refining procedural 3D programs [7, 14, 21, 33]. Rather than proposing another 3D-generation pipeline, 3DCodeBench provides a standardized benchmark that compares 12 frontier VLMs on the same tasks under a common execution and visual-feedback protocol [9].

Explicit structure is not new to 3D generation. StructureNet models shapes as hierarchical part graphs, ShapeAssembly generates parameterized programs with hierarchical part attachments, and Holodeck uses LLM-produced spatial constraints to arrange objects in 3D scenes [16, 23, 40]. These works show that hierarchy, attachment, and spatial relations can guide generation, but they operate in learned shape representations, a restricted assembly language, or scene-layout pipelines. Our question is narrower: whether open-ended VLM-generated Blender programs preserve specified intra-object attachments after either parent or child geometry changes. In this benchmark setting, disconnected or floating components remain common even among programs that execute and render successfully [9]. Its evaluation protocol does not explicitly require a common attachment representation, test connection preservation under controlled bidirectional variation of individual part parameters, or identify failed attachments through structure-aware geometric checks. Consequently, a program may appear plausible initially yet remain structurally unreliable.

As a motivating qualitative comparison, Fig. 1 shows one asset per model under deliberately exaggerated edits. In the selected baseline examples, attachments lose surface contact or child parts fail to follow a resized parent, whereas TreeStruct3D preserves geometric connectivity at the depicted attachments. These examples are illustrative; the cor-

Figure 1. Controlled editing examples. Each pair compares 3DCodeBench and TreeStruct3D under the default, parent-rescaled, and child-rescaled configurations. The $0.4\times$ and $1.6\times$ scaling is exaggerated for visual impact; quantitative tests use $0.8\times$ and $1.2\times$.

responding quantitative evaluation is reported in Tab. 4.

To address these limitations, we introduce TreeStruct3D, a structure-conditioned extension of the 3DCodeBench pipeline for generating structurally editable Blender programs. Given a text description, TreeStruct3D first predicts a Part–Attachment Tree and then conditions Blender code generation on both the description and the predicted tree. The code generator is instructed to implement each predicted parent–child attachment using a pair of geometry-dependent anchors. Whenever either endpoint is edited, the generated program is required to recompute the anchors and realign the child.

To further improve the quality of part–attachment relationships with more inference compute, TreeStruct3D applies structure-aware geometric checks to each generated edge: it independently scales selected parent and child parameters, then applies a deterministic method to identify part–attachment failures. It sends related diagnostic to the VLM for code revision, refining part–attachment relationships overtime.

In summary, our contributions are twofold:

- We introduce a structure-conditioned Blender code-generation approach that represents predicted object structure as a Part–Attachment Tree and realizes its parent–child dependencies through paired, geometry-derived anchors.
- We develop a dynamic attachment validation and repair protocol that independently edits parent and child parts, checks whether their encoded connections remain valid, and returns localized diagnostics to the VLM for code revision.

2. Related Work

Procedural and Agentic 3D Generation. Neural generators synthesize holistic 3D representations from text or images but do not retain an explicit construction process [27, 39, 42]. Procedural systems instead provide controllability through hand-authored rules and templates [6, 12, 28], while learned program representations expose structured construction operations [3, 16, 17, 32]. Recent language- and vision-language-model systems automate planning, code generation, execution, refinement, or parametric editing [7, 14, 21, 30, 33]. They improve controllable creation, but generally do not test whether an intra-object attachment survives an edit to either incident part.

Structured, Editable, and Evaluated 3D Assets. Structure-aware models organize objects through parts, hierarchies, and their relations [19, 23, 25, 37], with PartNet providing large-scale hierarchical annotations [24]. Work on articulated objects additionally models part geometry,

motion, and joint relations [10, 15, 20, 38]. For executable generation and editing, CADCodeVerify and BlenderGym evaluate visual correction, whereas 3DGen-Bench, VoxelCodeBench, and 3DCodeBench benchmark perceptual quality or procedural code generation [1, 9, 13, 41, 43]. None directly tests whether every specified parent–child attachment survives independent edits to either endpoint. TreeStruct3D makes these attachments explicit, validates both edit directions, and returns geometric evidence for targeted repair.

3. TreeStruct3D

TreeStruct3D extends the text-conditioned code-generation pipeline of 3DCodeBench [9] with structure-aware editability. It represents object structure as a part–attachment tree and compiles this structure into anchor-based geometric dependencies that can be validated and repaired under controlled part-level edits.

3.1. Task Formulation and Method Overview

We study text-conditioned procedural 3D generation. Given a textual object description x, our goal is to generate an executable Blender Python program P that produces an object visually consistent with x and exposes editable part-level geometry parameters. TreeStruct3D uses a predicted part–attachment tree $\hat{\mathcal{T}}$ as an operational representation of the attachments implied by x. We call P structure-aware editable if the attachment constraints encoded by $\hat{\mathcal{T}}$ remain satisfied under the designated part-level interventions.

As illustrated in Fig. 2, TreeStruct3D proceeds in four stages. First, in its structure-planning role, the VLM predicts a part–attachment tree $\hat{\mathcal{T}} = (\hat{\mathcal{V}}, \hat{\mathcal{E}})$ from x (Sec. 3.2). Second, in its code-generation role, the VLM translates x and $\hat{\mathcal{T}}$ into a Blender Python program that realizes each attachment through geometry-dependent anchors (Sec. 3.3). Third, a rule-based dynamic attachment validator executes the program in its default configuration and under controlled part-level edits (Sec. 3.4) to verify the structural correctness. Fourth, when validation fails, the validator converts its diagnostics into a repair instruction (Sec. 3.5) for the VLM. Execution failures are handled separately by the traceback-guided repair retained from 3DCodeBench [9]. The validation and repair stages may continue across multiple generation turns; the turn limits and continuation rule are described in Sec. 4.1. The validated program is then passed to the visual self-critique stage inherited from 3DCodeBench [9], and every visual revision is revalidated before acceptance.

3.2. Part–Attachment Tree

Given x, the VLM first decomposes the object into its main semantic parts. It selects one part as the root and assigns exactly one parent to every other part. Each node $v \in \hat{\mathcal{V}}$ has a stable identifier and records the part's name, semantic role, parent, quantity, importance, and coarse geometry, together with auxiliary construction metadata. For a repeated child node with quantity m, the tree stores one semantic node and one attachment rule rather than m separate nodes. During code generation, the node is instantiated m times, and the attachment rule produces one parent–child anchor pair for each instance. All instances share the same size and shape parameters and are therefore edited together.

Each directed edge $e = (p, c) \in \hat{\mathcal{E}}$ records the parent part p, the child part c, their intended attachment regions, and an anchor specification for each of the two parts. An anchor specification defines a rule for computing a point within the intended contact region based on the part's current geometric parameters. The generated program applies this rule to compute a local anchor for each endpoint, allowing the anchors to adapt automatically to changes in the part geometry (Sec. 3.3). The two anchors of an edge form an anchor pair and are required to coincide in world coordinates.

3.3. Parent–Child Dependencies with Anchors

The part–attachment tree specifies which parts should remain connected; the generated program must turn these declarative relations into executable geometric dependencies. To support part-level editing, P exposes `PART_PARAMS`, which maps each part identifier to geometry-construction parameters, including a required size scale and optional part-specific dimensions such as length, width, or radius. These parameters rebuild the part geometry rather than directly specifying its Blender translation; the attachment dependencies determine how connected parts are repositioned. Changing a geometry parameter in one entry modifies the corresponding part's geometry , while the attachment dependencies determine how connected parts must be repositioned.

A fixed placement or static parent–child transform is generally insufficient for attachment-preserving edits, because neither recomputes the attachment location after a geometry edit. For example, widening a torso moves its geometry-dependent side attachment point, so an attached arm must move outward accordingly. TreeStruct3D therefore realizes every attachment edge using a pair of geometry-dependent anchors whose local coordinates are recomputed from the current part geometry.

For an attachment edge $e = (p, c)$, let θ_p and θ_c denote the current geometry parameters of the parent and child in `PART_PARAMS`. The code-generation prompt asks the VLM to implement local-anchor functions $a_e^p(\theta_p)$ and $a_e^c(\theta_c)$ in the respective part coordinate systems. Given the local-to-world transforms T_p and T_c, placement requires only that the anchor pair coincide:

$$T_p a_e^p(\theta_p) = T_c a_e^c(\theta_c). \tag{1}$$

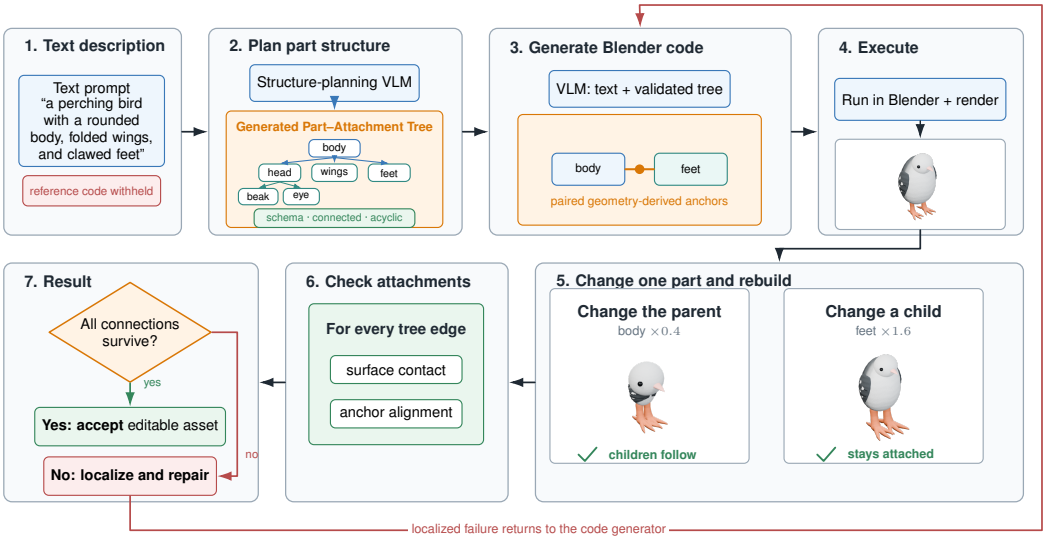

Figure 2. Overview of TreeStruct3D. A structure-planning VLM converts the text prompt into a validated Part–Attachment Tree. A code-generation VLM implements its edges using shared, geometry-derived anchors; Blender execution tests parent and child edits, and localized failures trigger repair. The bird panels show a real GPT-5.6 Sol output at default scale, with the body reduced to $0.4\times$ and the feet enlarged to $1.6\times$. Neither model receives the reference program.

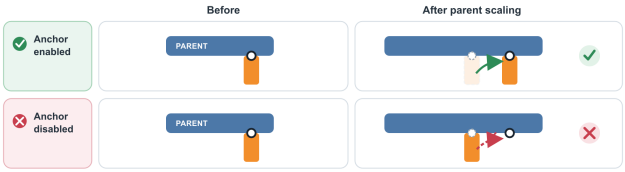

Figure 3. Effect of anchor recomputation after parent resizing. With recomputation, the child follows the updated parent anchor; without it, the child remains at its pre-edit position.

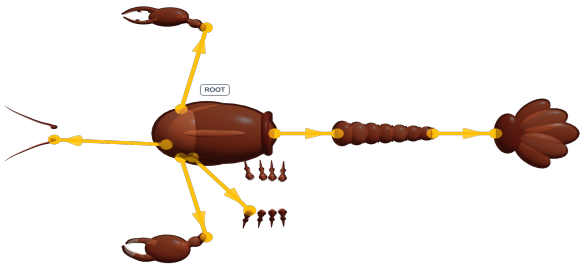

Figure 4. Exploded view of a lobster part–attachment tree. Yellow endpoints denote paired geometry-derived anchors, and arrows show the parent-to-child attachment direction.

The program enforces this constraint by evaluating both anchors and translating the child by their world-space offset.

Whenever a part is edited, the program rebuilds its geometry, recomputes every anchor defined on that part, and realigns all incident attachment edges. Blender parenting propagates these rigid transformations to all descendant parts. However, when geometry changes an edge's attachment location, its anchor must still be recomputed. Figure 4 illustrates how geometry-derived anchor pairs define the attachment hierarchy between parent and child parts.

3.4. Dynamic Attachment Validation

Anchor-aware generation alone does not guarantee structure-aware editability. A VLM-generated program may produce parts that appear connected even when their attachment locations are hard-coded and fail to update after a geometry edit. TreeStruct3D therefore applies a rule-based dynamic attachment validator in two phases: default-configuration validation and controlled-intervention validation.

Default-configuration validation The validator runs the complete program in a clean scene. Using the part IDs in $\hat{\mathcal{T}}$, it verifies that every specified part is present and that the program exposes both local anchors for every attachment edge. For each edge $e = (p, c)$, the validator maps the local anchors to world coordinates using the final part transformations and measures their Euclidean separation:

$$r_e = \left\| T_p a_e^p(\theta_p) - T_c a_e^c(\theta_c) \right\|_2 . \tag{2}$$

Here, $T_p a_e^p(\theta_p)$ and $T_c a_e^c(\theta_c)$ are respectively the world-space positions of the parent and child anchors. Thus, r_e measures how closely the alignment constraint in Eq. (1) is satisfied, with $r_e = 0$ indicating exact alignment. The check passes if $r_e \leq \epsilon_{\text{align}}$, where $\epsilon_{\text{align}} = 10^{-5}$ in

world-coordinate units. We use this as a conservative numerical tolerance for floating-point roundoff introduced by Blender's mesh-coordinate representation and local-to-world transformations, rather than requiring exact coordinate equality. The same tolerance is used for the anchor-on-mesh check because it likewise tests numerical coincidence after Blender's coordinate transformations.

To reject numerically aligned anchors that are detached from the final geometry, the validator additionally checks that each anchor lies on its associated mesh and that the two incident meshes remain in contact. These checks establish the geometric consistency of the program-declared attachments, but do not certify their semantic correctness because the validator does not independently infer semantic contact points.

Controlled-intervention validation As shown in Component 7 in Fig. 2, the validator checks two independent interventions for every attachment edge $e = (p, c)$: one on the parent p and one on the child c. Each distinct part is edited once, and the resulting run is reused to check all of its incident edges. Each intervention starts from the default parameter configuration. In each run, the validator changes the tested part's `scale` value in `PART_PARAMS` from 1.0 to 1.35, a moderate perturbation intended to produce a measurable change without excessively distorting the part, and reruns the complete program. The schema requires a `scale` value for every part, and the validator changes this value directly. An intervention passes only if the parameter produces a nontrivial geometry change that alters the part's size, shape, or number of vertices rather than only its position and every attachment incident to the edited part continues to pass the anchor-alignment, the anchor-on-mesh, and surface-contact checks.

3.5. Validation-Guided Attachment Repair

When parsing, execution, or attachment validation fails, TreeStruct3D constructs a repair instruction for the VLM. The instruction includes the original description x, the predicted part–attachment tree $\hat{\mathcal{T}}$, the current program P, and the exact validation failures. A failure may identify a missing part or attachment, an invalid `PART_PARAMS` entry, loss of surface contact, misaligned anchors, or an attachment that breaks after its parent or child is edited. The VLM is instructed to correct the reported failure while preserving unrelated parts, materials, and visual appearance.

3.6. Multi-turn editing loop

After each repair to the program produced in the preceding round, we rerun the complete program in a clean scene and repeat all parsing, execution, and attachment checks. Parsing or execution failures trigger traceback-guided repair, whereas attachment failures cause the corresponding edge-level diagnostics to be returned to the VLM. Each repair attempt, whether prompted by an execution or attachment failure, is counted as a new generation turn. The reporting horizons and continuation policy are specified in Sec. 4.1.

Compared with the executability-only safeguard in 3DCodeBench [9], TreeStruct3D revalidates every self-critique revision using both execution and attachment checks. A revision is accepted only if all checks pass; otherwise, the program is reverted to the last validated version.

4. Experiments

We evaluate whether TreeStruct3D preserves conventional generation fidelity, how well its predicted trees agree with automatic structural references, and whether the generated programs preserve reference-relative attachments under controlled edits.

4.1. Experimental Setup

Benchmark. We evaluate on the same 212 text-to-3D cases adopted by 3DCodeBench [9] from Infinigen [28]. The automatic benchmark-derived part–attachment trees described below contain 480 directed attachment edges across 172 multi-part cases; the remaining 40 cases contain a single part and no reference attachment edge.

Automatic reference trees. For each benchmark program, we derive a part–attachment tree from source code, execution traces, and final-mesh geometry using a fixed GPT-5.6 Sol annotator. We retain only outputs whose parts and edges are traceable to these inputs and whose structure is connected, acyclic, and schema-valid (Sec. A.5). The reference programs and derived trees are hidden from all generation, repair, and in-loop validation components and are used only for post-hoc evaluation. Because GPT-5.6 Sol is also evaluated, we treat these trees as reproducible comparison targets rather than human-verified ground truth. Further construction details are provided in Sec. A.

Compared systems. 3DCodeBench provides both the 212-case benchmark and the baseline code-generation pipeline used for comparison. TreeStruct3D extends this baseline with explicit structure planning and dynamic attachment validation. We evaluate GPT-5.5, GPT-5.6 Sol, Gemini 3.1 Pro, and Gemini 3.5 Flash.

Generation and archiving. We run one trajectory per model–method–case. Each scored turn contains one code response followed by method-specific acceptance checks: 3DCodeBench requires successful execution and a valid non-empty render, whereas TreeStruct3D additionally requires all dynamic attachment checks to pass. A failure prompts a revision in the next turn. ST denotes acceptance on the first turn, and MT denotes cumulative acceptance

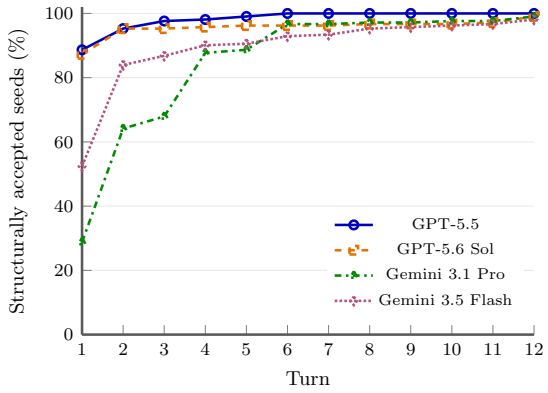

Figure 5. TreeStruct3D cumulative first-passage acceptance through 12 generation turns.

within four turns. Because the two methods use different acceptance criteria, ST and MT are process statistics rather than a common measure of executability.

Visual self-critique occurs only after method-specific acceptance; its revisions do not count toward ST or MT and are re-executed and, for TreeStruct3D, revalidated before archiving. Remaining cases continue without scoring until method-specific acceptance. The accepted program and its four-view renders are then archived for downstream evaluation. Full acceptance rates are reported in Sec. B and Tab. 8.

Figure 5 shows that GPT-5.5 reaches 100% first-passage acceptance by turn 6; by turn 12, the other models reach 98.11–99.06%. The curve measures time to passage, not the causal effect of validation feedback.

Evaluation protocol. We re-execute every archived program from an empty Blender 5.0 scene; part-name normalization and semantic and lexical matching follow Sec. 4.3. TreeStruct3D's in-loop validation applies a $1.35\times$ edit to one part parameter at a time and requires a nontrivial geometry change and passage of all incident anchor-alignment, anchor-on-mesh, and surface-contact checks (Sec. 3.4). For post-hoc evaluation, held-out $0.8\times$ and $1.2\times$ edits are applied separately to each endpoint of every eligible reference edge, testing contraction and expansion uniformly across programs. Post-hoc connectivity is measured directly from surface contact in the resulting geometry, without using program-reported anchors or in-loop decisions; thus *geometric connectivity* in Tabs. 3 and 4 denotes this surface-contact outcome rather than the full in-loop criterion.

4.2. Basic Generation Fidelity

Before evaluating structural benefits, we first examine whether explicit structure planning preserves conventional generation quality. We treat this comparison as a basic-quality sanity check; the principal structural and editability benefits are evaluated in the subsequent sections.

Paired fidelity metrics. We compare generated and reference objects using SigLIP-2 [35] for four-view image embeddings, Uni3D [44] for 3D features, and Chamfer Distance (CD) [8] for 8,192-point surface samples. For each model, we evaluate only cases with valid archived outputs from both methods: four rendered views for SigLIP-2 and a readable GLB for Uni3D and CD. This paired set may include outputs from the unscored continuation and therefore differs from the four-turn acceptance set.

We estimate uncertainty using 10,000 paired, case-level bootstrap replicates over the shared valid cases (NumPy seed 0). Let T_i^k and B_i^k denote metric k for TreeStruct3D and 3DCodeBench. We orient all differences so that positive values favor TreeStruct3D:

$$\begin{aligned} d_i^{\text{SigLIP-2}} &= T_i^{\text{SigLIP-2}} - B_i^{\text{SigLIP-2}}, \\ d_i^{\text{Uni3D}} &= T_i^{\text{Uni3D}} - B_i^{\text{Uni3D}}, \\ d_i^{\text{CD}} &= B_i^{\text{CD}} - T_i^{\text{CD}}. \end{aligned} \tag{3}$$

We report percentile-bootstrap 95% confidence intervals without making formal equivalence claims.

Table 1. Paired fidelity differences with bootstrap 95% confidence intervals. Differences are oriented so that positive values favor TreeStruct3D.

Model	Metric	Paired N	Difference	95% CI
GPT-5.5	SigLIP-2	212	−0.0049	[−0.0099, +0.0003]
	Uni3D	212	−0.0044	[−0.0166, +0.0081]
	CD	212	−0.0109	[−0.0200, −0.0029]
GPT-5.6 Sol	SigLIP-2	212	−0.0055	[−0.0098, −0.0013]
	Uni3D	212	+0.0010	[−0.0124, +0.0144]
	CD	212	−0.0024	[−0.0084, +0.0041]
Gemini 3.1 Pro	SigLIP-2	212	−0.0153	[−0.0217, −0.0089]
	Uni3D	212	+0.0237	[+0.0034, +0.0444]
	CD	212	+0.0026	[−0.0061, +0.0110]
Gemini 3.5 Flash	SigLIP-2	212	−0.0013	[−0.0071, +0.0045]
	Uni3D	212	+0.0030	[−0.0143, +0.0206]
	CD	212	−0.0060	[−0.0138, +0.0007]

Results. The fidelity tradeoff is model- and metric-dependent. As shown in Tab. 1, the confidence intervals do not establish a consistent advantage for either method, nor do they by themselves establish practical equivalence.

4.3. Agreement with Automatic Reference Trees

The 212 automatic reference trees contain 692 part nodes and 480 directed attachment edges. Because predicted and reference trees may use different part names, we normalize the names and compute separate one-to-one assignments from semantic cosine similarity and lexical overlap. Structural metrics use the semantic assignment; the edge-level

Table 2. Agreement between the predicted and reference discrete structural projections. Values are percentages except Depth MAE; Attachment F1 and Root Accuracy use 172 multi-part cases, and the other metrics use all 212.

Model	Semantic Name Similarity	Lexical Name Overlap	Attach. F1	Root Acc.	Depth MAE ↓	Exact Structural Match
GPT-5.5	42.62	21.62	51.11	70.35	0.275	20.75
GPT-5.6 Sol	46.93	26.58	48.55	69.77	0.263	**21.70**
Gemini 3.1 Pro	**57.83**	**39.82**	**53.56**	**74.42**	**0.257**	18.87
Gemini 3.5 Flash	52.55	35.39	53.55	70.93	0.319	13.21

analysis below retains only part pairs on which the two independent assignments agree. The matching equations, complete metric definitions, and aggregation rules are provided in Sec. A.4.

Semantic Name Similarity and Lexical Name Overlap are normalized assignment scores in which unmatched parts contribute zero. Attachment F1 counts recovered directed reference edges between semantically matched parts as true positives, other predicted edges as false positives, and missing reference edges as false negatives. Root Accuracy checks the matched root, Depth MAE averages level differences over matched parts, and Exact Structural Match requires equal part counts, identities, quantities, roots, and directed edge sets.

Across the four VLMs, Attachment F1 ranges from 48.55% to 53.56% and Exact Structural Match from 13.21% to 21.70%. Gemini 3.1 Pro leads most agreement metrics, while GPT-5.6 Sol has the highest Exact Structural Match. These results show that recovering the reference structure remains a substantial bottleneck, motivating the edge-level decomposition of recovery and conditional contact preservation in the next subsection.

4.4. Structure-Aware Editability

The task formulation in Sec. 3.1 defines attachment preservation relative to the predicted tree, whereas reference-relative end-to-end editability also requires recovering the relevant reference edge and exposing independent controls for both endpoints. We therefore report two complementary analyses. First, Sec. 4.4.1 diagnoses TreeStruct3D over all 480 reference edges by separating reference-edge recovery from conditional surface-contact preservation. Second, Sec. 4.4.2 compares TreeStruct3D and 3DCodeBench over the same 479 eligible reference edges, requiring both independent endpoint controls and surface-contact preservation. A separate paired test restricts the latter comparison to edges controllable by both methods. Finally, Sec. 4.4.3 directly tests the anchor-recomputation mechanism. The two tabulated edge sets are not nested because they use different correspondence and eligibility conditions, so their counts may differ in either direction.

4.4.1. Edge Recovery and Contact Preservation

We analyze all 480 directed reference edges for TreeStruct3D. A reference part is retained only when the independent semantic and lexical matchings select the same predicted part:

$$\pi_{\cap}(r) = \begin{cases} p, & \pi^*_{\text{sem}}(r) = \pi^*_{\text{lex}}(r) = p, \quad p \in P, \\ \varnothing, & \text{otherwise}, \end{cases} \tag{4}$$

where $\pi^*_{\text{sem}}(r)$ and $\pi^*_{\text{lex}}(r)$ are defined in Sec. 4.3, and E_R and E_P denote the directed edge sets of the reference and predicted trees. The recovery counts in Tab. 3 are:

$$\begin{aligned} m(r) &= \mathbf{1}[\pi_{\cap}(r) \neq \varnothing], \\ h(r_p, r_c) &= \mathbf{1}[(\pi_{\cap}(r_p), \pi_{\cap}(r_c)) \in E_P], \\ N_{\text{cons}} &= \sum_{(r_p, r_c) \in E_R} m(r_p) m(r_c), \\ N_{\text{edge}} &= \sum_{(r_p, r_c) \in E_R} m(r_p) m(r_c) h(r_p, r_c). \end{aligned} \tag{5}$$

N_{cons} counts edges whose endpoints both have consensus matches; N_{edge} additionally requires the matched directed edge in E_P. Each reference edge is counted once.

For each matched directed edge, we test surface contact in the default configuration and after separately rescaling the parent or child; each rescaling test must preserve contact at both $0.8\times$ and $1.2\times$. Geometric connectivity requires all three tests. Edge pass uses successful edges over all 480 references. A multi-part asset passes only if every reference edge is matched and passes all contact tests; extra predicted edges are ignored.

Conditional preservation among matched edges is 97.08–100%, but overall edge pass is 26.46–30.62%. Thus, contact is almost always preserved after consensus recovery; most failures arise from structural mismatch or disagreement between the semantic and lexical matchings.

4.4.2. End-to-End Editability

We reuse archived programs from Sec. 4.1 without regeneration and apply the same post-hoc contact tests to both methods. TreeStruct3D uses its `PART_PARAMS` controls. For 3DCodeBench, we search source code for distinct size-related parameters (e.g., `width`, `height`, `radius`, and `scale`) controlling the two endpoints. Each edit modifies a temporary program copy and re-executes the full program; no Blender transform is applied afterward.

Candidate parameters are ranked by surrounding code and target-part names and retained only if they cause a non-trivial target-geometry change. If the top-ranked candidates do not yield distinct effective controls for the parent and child, the search expands to all recognizable size-related parameters in the abstract syntax tree. This restricted search may miss other editing mechanisms.

Table 3. Reference-edge recovery and post-hoc surface-contact preservation under held-out scale edits for TreeStruct3D programs with anchor recomputation. A part pair is retained only when the separately computed semantic and lexical matchings agree. Values are count/denominator (%); each rescaling test must pass at both $0.8\times$ and $1.2\times$.

	Recovery (480 ref. edges)		Part-rescaling tests				Overall pass	
Model	Endpoint-consensus coverage	Directed attachment edge match	Default configuration	Parent rescaling	Child rescaling	Geometric connectivity across tested scales	Edge pass	Asset level
GPT-5.5	146/480 (30.42%)	129/480 (26.88%)	129/129 (100.00%)	129/129 (100.00%)	129/129 (100.00%)	129/129 (100.00%)	129/480 (26.88%)	41/172 (23.84%)
GPT-5.6 Sol	155/480 (32.29%)	128/480 (26.67%)	128/128 (100.00%)	128/128 (100.00%)	127/128 (99.22%)	127/128 (99.22%)	127/480 (26.46%)	35/172 (20.35%)
Gemini 3.1 Pro	167/480 (34.79%)	147/480 (30.62%)	147/147 (100.00%)	147/147 (100.00%)	147/147 (100.00%)	147/147 (100.00%)	147/480 (30.62%)	44/172 (25.58%)
Gemini 3.5 Flash	157/480 (32.71%)	137/480 (28.54%)	137/137 (100.00%)	136/137 (99.27%)	134/137 (97.81%)	133/137 (97.08%)	133/480 (27.71%)	38/172 (22.09%)

Table 4. Structure-aware editability on 479 eligible directed attachment edges. Bold indicates the better value between the two methods for each model.

Model	Method	Independent-control coverage	Conditional surface-contact preservation	End-to-end editability
GPT-5.5	3DCodeBench	37/479 (7.72%)	17/37 (45.95%)	17/479 (3.55%)
GPT-5.5	TreeStruct3D	**151/479 (31.52%)**	**132/151 (87.42%)**	**132/479 (27.56%)**
GPT-5.6 Sol	3DCodeBench	76/479 (15.87%)	40/76 (52.63%)	40/479 (8.35%)
GPT-5.6 Sol	TreeStruct3D	**136/479 (28.39%)**	**114/136 (83.82%)**	**114/479 (23.80%)**
Gemini 3.1 Pro	3DCodeBench	29/479 (6.05%)	7/29 (24.14%)	7/479 (1.46%)
Gemini 3.1 Pro	TreeStruct3D	**167/479 (34.86%)**	**147/167 (88.02%)**	**147/479 (30.69%)**
Gemini 3.5 Flash	3DCodeBench	21/479 (4.38%)	10/21 (47.62%)	10/479 (2.09%)
Gemini 3.5 Flash	TreeStruct3D	**158/479 (32.99%)**	**129/158 (81.65%)**	**129/479 (26.93%)**

Of the 480 reference edges, 479 are eligible; one armchair–ottoman relation is excluded because the ottoman is a companion object rather than a part-level attachment. An eligible edge passes only if its endpoints have independent controls and preserve surface contact in the default state and after separately scaling the parent or child to both $0.8\times$ and $1.2\times$.

Tab. 4 shows higher TreeStruct3D editability for all models via coverage and preservation, with raw-count ratios of $7.76\times$, $2.85\times$, $21.00\times$, and $12.90\times$, respectively. The maximum is $147/7 = 21.00\times$ for Gemini 3.1 Pro (147 editable edges for TreeStruct3D vs. 7 for 3DCodeBench); the paired subset shows the same trend.

4.4.3. Anchor-Recomputation Ablation

Using matched attachments from Tab. 3, we test whether recomputed geometry-dependent anchors make the child follow an edited parent, as illustrated earlier in Fig. 3. For each matched attachment, we run the same program with recomputation enabled and disabled. The enabled program recomputes the anchor pair after parent resizing; the ablation removes that function and holds the child at its pre-edit position. At parent scales $0.8\times$ and $1.2\times$, anchor-following agreement is

$$A_{\text{follow}} = 1 - \frac{\|\Delta\mathbf{c} - \Delta\mathbf{p}\|_2}{\|\Delta\mathbf{p}\|_2}, \tag{6}$$

where $\Delta\mathbf{p}$ and $\Delta\mathbf{c}$ are the parent- and child-anchor displacements. Scores of one and zero denote identical motion and a fixed child, respectively. Filters require valid, variant-consistent parent motion, a fixed disabled child, and matching default geometry.

Table 5. Anchor-recomputation ablation under parent rescaling. Counts and median agreement are identical at the two tested scales.

Model	Parent scales	Retained / candidate	Recompute on	Recompute off
GPT-5.5	$0.8\times, 1.2\times$	118/129	100%	0%
GPT-5.6 Sol	$0.8\times, 1.2\times$	112/128	100%	0%
Gemini 3.1 Pro	$0.8\times, 1.2\times$	132/147	100%	0%
Gemini 3.5 Flash	$0.8\times, 1.2\times$	117/137	100%	0%

After score-independent filtering, 112–132 attachments remain per model. Recomputation yields 100% median agreement for every model at both scales, while the fixed-child ablation yields 0% by construction. This isolates anchor-motion transfer rather than surface contact or overall asset quality.

5. Conclusion and Limitations

TreeStruct3D represents predicted part attachments as geometry-dependent anchors, recomputes them after either endpoint changes, and uses controlled edits to drive localized repair. Across four VLMs, TreeStruct3D achieves $2.85\times$–$21.00\times$ the end-to-end editability of 3DCodeBench. Among recovered reference edges, contact survives all tested states in 97.08%–100% of cases. Paired fidelity differences remain model and metric dependent, while the helper intervention confirms that recomputation transfers parent-anchor motion to the child anchor. Together, these results move procedural generation from programs that merely render toward assets whose encoded relations remain testable and editable after controlled modifications.

Limitations. The planner may omit semantic parts or attachments: conditional preservation applies only to encoded edges, whereas the overall edge metric counts omissions. The automatic references are reproducible comparison targets, not human-verified ground truth. Our tests scale one endpoint at a time, excluding simultaneous edits, rotation, topology changes, and deformation; repeated parts are evaluated as groups. Finally, anchor-motion agreement tests declared-anchor transfer, not surface contact.

References

[1] Kamel Alrashedy, Pradyumna Tambwekar, Zulfiqar Zaidi, Megan Langwasser, Wei Xu, and Matthew Gombolay. Generating CAD code with vision-language models for 3d designs. In *International Conference on Learning Representations*, 2025. 3

[2] Anthropic. Claude for creative work. https://www.anthropic.com/news/claude-for-creative-work, 2026. Accessed 2026-08-13. 1

[3] Armen Avetisyan, Christopher Xie, Henry Howard-Jenkins, Tsun-Yi Yang, Samir Aroudj, Suvam Patra, Fuyang Zhang, Duncan Frost, Luke Holland, Campbell Orme, Jakob Engel, Edward Miller, Richard Newcombe, and Vasileios Balntas. SceneScript: Reconstructing scenes with an autoregressive structured language model. In *European Conference on Computer Vision*, pages 247–263. Springer, 2024. 2

[4] Blender Foundation. Blender mcp server. https://www.blender.org/lab/mcp-server/, 2026. Accessed 2026-08-13. 1

[5] 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*, pages 5982–5994, 2022. Available at: https://proceedings.neurips.cc/paper_files/paper/2022/hash/27c546ab1e4f1d7d638e6a8dfbad9a07-Abstract-Conference.html. 1

[6] Maximilian Denninger, Martin Sundermeyer, Dominik Winkelbauer, Youssef Zidan, Dmitry Olefir, Mohamad Elbadrawy, Ahsan Lodhi, and Harinandan Katam. BlenderProc. *arXiv preprint arXiv:1911.01911*, 2019. Available at: https://arxiv.org/abs/1911.01911. 1, 2

[7] 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. Available at: https://arxiv.org/abs/2412.14203. 1, 2

[8] Haoqiang Fan, Hao Su, and Leonidas J. Guibas. A point set generation network for 3d object reconstruction from a single image. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 605–613, 2017. 6

[9] Yipeng Gao, Lei Shu, Genzhi Ye, Xi Xiong, Ameesh Makadia, Meiqi Guo, Laurent Itti, and Jindong Chen. 3DCodeBench: Benchmarking agentic procedural 3d modeling via code. *arXiv preprint arXiv:2606.01057*, 2026. Available at: https://arxiv.org/abs/2606.01057. 1, 3, 5

[10] Haoran Geng, Helin Xu, Chengyang Zhao, Chao Xu, Li Yi, Siyuan Huang, and He Wang. GAPartNet: Cross-category domain-generalizable object perception and manipulation via generalizable and actionable parts. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7081–7091, 2023. 3

[11] Google. Gemini cli: Your open-source ai agent. https://blog.google/innovation-and-ai/technology/developers-tools/introducing-gemini-cli-open-source-ai-agent/, 2025. Accessed 2026-08-13. 1

[12] Klaus Greff, Francois Belletti, Lucas Beyer, Carl Doersch, Yilun Du, Daniel Duckworth, David J. Fleet, Dan Gnanapragasam, Florian Golemo, Charles Herrmann, Thomas Kipf, Abhijit Kundu, Dmitry Lagun, Issam Laradji, Hsueh-Ti Derek Liu, Henning Meyer, Yishu Miao, Derek Nowrouzezahrai, Cengiz Oztireli, Etienne Pot, Noha Radwan, Daniel Rebain, Sara Sabour, Mehdi S. M. Sajjadi, Matan Sela, Vincent Sitzmann, Austin Stone, Deqing Sun, Suhani Vora, Ziyu Wang, Tianhao Wu, Kwang Moo Yi, Fangcheng Zhong, and Andrea Tagliasacchi. Kubric: A scalable dataset generator. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3749–3761, 2022. 2

[13] Yunqi Gu, Ian Huang, Jihyeon Je, Guandao Yang, and Leonidas J. Guibas. BlenderGym: Benchmarking foundational model systems for graphics editing. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 18574–18583, 2025. 3

[14] 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 *Proceedings of the 41st International Conference on Machine Learning*, pages 19252–19282, 2024. Available at: https://proceedings.mlr.press/v235/hu24g.html. 1, 2

[15] Zhenyu Jiang, Cheng-Chun Hsu, and Yuke Zhu. Ditto: Building digital twins of articulated objects from interaction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5616–5626, 2022. 3

[16] 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*, 39(6):234:1–234:20, 2020. 1, 2

[17] R. Kenny Jones, Paul Guerrero, Niloy J. Mitra, and Daniel Ritchie. ShapeCoder: Discovering abstractions for visual programs from unstructured primitives. *ACM Transactions on Graphics*, 42(4):49:1–49:17, 2023. 2

[18] Harold W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1–2):83–97, 1955. 13

[19] Jun Li, Kai Xu, Siddhartha Chaudhuri, Ersin Yumer, Hao Zhang, and Leonidas Guibas. GRASS: Generative recursive autoencoders for shape structures. *ACM Transactions on Graphics*, 36(4):52:1–52:14, 2017. 2

[20] Jiayi Liu, Ali Mahdavi-Amiri, and Manolis Savva. PARIS: Part-level reconstruction and motion analysis for articulated objects. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 352–363, 2023. 3

[21] 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. Available at: https://arxiv.org/abs/2508.08228. 1, 2

[22] Meshy. 3d ai agent for conversational 3d creation. https://www.meshy.ai/features/3d-agent, 2026. Accessed 2026-08-13. 1

[23] Kaichun Mo, Paul Guerrero, Li Yi, Hao Su, Peter Wonka, Niloy J. Mitra, and Leonidas J. Guibas. StructureNet: Hierarchical graph networks for 3d shape generation. *ACM Transactions on Graphics*, 38(6):242:1–242:19, 2019. 1, 2

[24] Kaichun Mo, Shilin Zhu, Angel X. Chang, Li Yi, Subarna Tripathi, Leonidas J. Guibas, and Hao Su. PartNet: A large-scale benchmark for fine-grained and hierarchical part-level 3d object understanding. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 909–918, 2019. 2

[25] Kaichun Mo, Paul Guerrero, Li Yi, Hao Su, Peter Wonka, Niloy J. Mitra, and Leonidas J. Guibas. StructEdit: Learning structural shape variations. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8859–8868, 2020. 2

[26] OpenAI. Openai codex cli: Getting started. https://help.openai.com/en/articles/11096431, 2026. Accessed 2026-08-13. 1

[27] Ben Poole, Ajay Jain, Jonathan T. Barron, and Ben Mildenhall. DreamFusion: Text-to-3d using 2d diffusion. In *International Conference on Learning Representations*, 2023. 2

[28] 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 *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12630–12641, 2023. Available at: https://openaccess.thecvf.com/content/CVPR2023/html/Raistrick_Infinite_Photorealistic_Worlds_Using_Procedural_Generation_CVPR_2023_paper.html. 1, 2, 5

[29] Alexander Raistrick, Lingjie Mei, Karhan Kayan, David Yan, Yiming Zuo, Beining Han, Hongyu Wen, Meenal Parakh, Stamatis Alexandropoulos, Lahav Lipson, Zeyu Ma, and Jia Deng. Infinigen Indoors: Photorealistic indoor scenes using procedural generation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 21783–21794, 2024. Available at: https://openaccess.thecvf.com/content/CVPR2024/html/Raistrick_Infinigen_Indoors_Photorealistic_Indoor_Scenes_using_Procedural_Generation_CVPR_2024_paper.html. 1

[30] Fadlullah Raji, Stefano Petrangeli, Matheus Gadelha, Yu Shen, Uttaran Bhattacharya, and Gang Wu. Proc3D: Procedural 3d generation and parametric editing of 3d shapes with large language models. *arXiv preprint arXiv:2601.12234*, 2026. 2

[31] Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence embeddings using siamese BERT-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*, pages 3982–3992. Association for Computational Linguistics, 2019. 13

[32] Gopal Sharma, Rishabh Goyal, Difan Liu, Evangelos Kalogerakis, and Subhransu Maji. CSGNet: Neural shape parser for constructive solid geometry. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5515–5523, 2018. 2

[33] Chunyi Sun, Junlin Han, Weijian Deng, Xinlong Wang, Zishan Qin, and Stephen Gould. 3D-GPT: Procedural 3d modeling with large language models. *arXiv preprint arXiv:2310.12945*, 2023. Available at: https://arxiv.org/abs/2310.12945. 1, 2

[34] Tripo AI. Tripo developers: 3d generation api. https://developers.tripo3d.ai/en/, 2026. Accessed 2026-08-13. 1

[35] Michael Tschannen, Alexey Gritsenko, Xiao Wang, Muhammad Ferjad Naeem, Ibrahim Alabdulmohsin, Nikhil Parthasarathy, Talfan Evans, Lucas Beyer, Ye Xia, Basil Mustafa, Olivier J. Hénaff, Jeremiah Harmsen, Andreas Steiner, and Xiaohua Zhai. SigLIP 2: Multilingual vision-language encoders with improved semantic understanding, localization, and dense features. *arXiv preprint arXiv:2502.14786*, 2025. 6

[36] Wenhui Wang, Furu Wei, Li Dong, Hangbo Bao, Nan Yang, and Ming Zhou. MiniLM: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. In *Advances in Neural Information Processing Systems*, pages 5776–5788, 2020. 13

[37] Rundi Wu, Yixin Zhuang, Kai Xu, Hao Zhang, and Baoquan Chen. PQ-NET: A generative part seq2seq network for 3d shapes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 829–838, 2020. 2

[38] Fanbo Xiang, Yuzhe Qin, Kaichun Mo, Yikuan Xia, Hao Zhu, Fangchen Liu, Minghua Liu, Hanxiao Jiang, Yifu Yuan, He Wang, Li Yi, Angel X. Chang, Leonidas J. Guibas, and Hao Su. SAPIEN: A simulated part-based interactive environment. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11097–11107, 2020. 3

[39] 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. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 21469–21480, 2025. 2

[40] 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 *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16227–16237, 2024. 1

[41] Yuhan Zhang, Mengchen Zhang, Tong Wu, Tengfei Wang, Gordon Wetzstein, Dahua Lin, and Ziwei Liu. 3DGen-Bench: Comprehensive benchmark suite for 3d generative models. *arXiv preprint arXiv:2503.21745*, 2025. 3

[42] Zibo Zhao, Zeqiang Lai, Qingxiang Lin, et al. Hunyuan3D 2.0: Scaling diffusion models for high resolution textured 3d assets generation. *arXiv preprint arXiv:2501.12202*, 2025. Available at: https://arxiv.org/abs/2501.12202. 1, 2

[43] Yan Zheng and Florian Bordes. VoxelCodeBench: Benchmarking 3d world modeling through code generation. *arXiv preprint arXiv:2604.02580*, 2026. 3

[44] Junsheng Zhou, Jinsheng Wang, Baorui Ma, Yu-Shen Liu, Tiejun Huang, and Xinlong Wang. Uni3D: Exploring unified 3d representation at scale. In *International Conference on Learning Representations*, 2024. 6

A. Automatically Derived Structural Reference

A.1. Motivation and Evaluation Role

The Part–Attachment Tree used for generation is predicted from text and can therefore be incomplete or semantically wrong even when it is internally valid. Dynamic validation determines whether a generated program preserves the connections specified by its own tree, but cannot determine whether that tree captures the parts and attachments expressed by the benchmark asset. We therefore automatically derive a fixed structural reference from each of the 212 reference programs provided by 3DCodeBench and use it only after generation to evaluate the predicted tree.

The reference program is never shown to the structure planner or the Blender code generator. Offline, our pipeline first traces its construction operations and normalizes code, runtime, and geometric evidence. A fixed semantic annotator then converts this evidence into a neutral reference tree whose parts and attachments remain bound to their source evidence. This reference is automatic rather than human verified; consequently, it supports reproducible reference-agreement measurements but is not treated as an absolute semantic Ground Truth. The separation lets us measure two different properties: agreement between the predicted and reference structures, and preservation of generated attachments under controlled edits.

Table 6. Seven construction strategies found in the benchmark reference programs and the rules used to normalize their implementation-level structural evidence.

Construction strategy	Cases	How the reference structure is recovered
Objects joined into one mesh	113	Join inputs, per-source point provenance, and measured geometric relations
Single directly built mesh	71	One surviving authored mesh; no unsupported child attachment is introduced
Explicit parent–child objects	8	Executed parent or common-parent events and observed relative transforms
Final mesh with temporary helpers	13	One surviving core; deleted cutters and helpers remain integrated operations
Parts merged through BMesh	3	Source-to-final point provenance and exact vertex conservation
Parts assembled with Geometry Nodes	2	Template provenance and realized instance counts
Multiple separate final objects	2	Returned or visible root objects and their observed relative placement

A.2. Reference-Program Evidence Extraction

Reference programs rarely expose a complete hierarchy through Blender's `parent` field. We therefore combine three sources of evidence. Static AST analysis records functions, calls, and construction sites such as Join, explicit parenting, modifiers, and Geometry Nodes. Instrumented execution assigns stable object identifiers and records each object's creation, copy source, modification, deletion, and participation in Join operations. Finally, evaluated meshes provide the geometry after all modifiers have been applied, including point provenance and measured contact, containment, or relative placement.

Every recovered relation can be traced back to source code, runtime events, and geometric measurements. Traces are bound to the reference source by a SHA-256 hash, and unresolved operations remain explicitly unresolved rather than being guessed. This stage uses neither an LLM nor per-instance manual rules; it produces auditable implementation-level evidence rather than the final semantic reference by itself.

A.3. Construction-Specific Evidence Normalization

The 212 reference programs use seven distinct construction patterns. A single conversion rule would either miss valid geometry contributions or promote temporary tools to semantic parts. We therefore route each program by its observed construction pattern and apply the corresponding evidence contract, summarized in Tab. 6. All routes produce the same implementation-level evidence format and fail closed when required evidence is missing. This stage records how reference geometry was assembled; semantic grouping is performed only in the next stage.

A.4. Automatic Semantic Reference and Matching

A fixed GPT-5.6 Sol annotator receives the asset description, reference program, and normalized objective evidence. Under a neutral schema, it outputs only semantic parts, their quantities, one root, and directed parent–child relations; it is not given TreeStruct3D's shared-anchor, recomputation, or repair requirements. Every proposed part and relation must cite corresponding code or runtime evidence. We retain only connected, acyclic, schema-valid outputs and freeze the complete set before evaluating any structure-planning model.

The predicted and automatically derived structures share the same rooted Part–Attachment Tree schema. Because independently produced identifiers need not use identical strings, we establish one-to-one correspondence from normalized part names only. Semantic cosine similarity and lexical Jaccard similarity are assigned independently with zero-score dummy nodes; they are not mixed, and root role, depth, quantity, and neighboring relations do not affect matching. A predicted attachment is correct only when both semantically mapped endpoints and the parent-to-child direction agree. Semantic and Lexical Assignment Agreement, Attachment F1, Root Accuracy, Depth MAE, and Exact Tree therefore measure automatic structural-reference agreement; the dynamic tests in Sec. 3.4 separately evaluate whether the generated program preserves its specified attachments after parameter changes.

Part correspondence. We normalize part names by splitting identifier boundaries, removing punctuation, and lowercasing. Matching uses names only; semantic role, root status, depth, quantity, and neighboring edges are excluded. For each reference–prediction pair (r, p), `all-MiniLM-L6-v2` [31, 36] produces embeddings e_r and e_p, while tokenization produces word sets T_r and T_p:

$$c_{\text{sem}}(r,p) = \cos(e_r, e_p),$$
$$c_{\text{lex}}(r,p) = \frac{|T_r \cap T_p|}{|T_r \cup T_p|}. \tag{7}$$

We optimize the two scores independently using separate Hungarian assignments [18]. Let Π denote the set of partial one-to-one matchings and $v \in \{\text{sem}, \text{lex}\}$:

$$\pi_v^* = \arg\max_{\pi \in \Pi} \sum_{(r,p)\in\pi} c_v(r,p). \tag{8}$$

Pairs with nonpositive similarity are discarded using a fixed, untuned zero cutoff, so either side may remain unmatched. The normalized matching score is

$$A_v = \frac{\sum_{(r,p)\in\pi_v^*} c_v(r,p)}{\max(|R|,|P|)}, \tag{9}$$

where unmatched parts contribute zero. We report $100A_{\text{sem}}$ as Semantic Name Similarity and $100A_{\text{lex}}$ as Lexical Name Overlap.

Structural metrics. We evaluate part identities and quantities, the root, depth, and directed attachments, excluding semantic roles, coarse geometry, attachment regions, and anchors. For Attachment F1, recovered directed reference edges between semantically matched parts are true positives, other predicted edges are false positives, and missing reference edges are false negatives. Root Accuracy checks the root match; Depth MAE averages level differences over matched parts; and Exact Structural Match requires equal part counts, identities, quantities, root, and directed edge sets. All metrics are averaged equally across cases. Semantic Name Similarity, Lexical Name Overlap, Depth MAE, and Exact Structural Match use all 212 cases; Attachment F1 and Root Accuracy use the 172 multi-part cases.

A.5. Dataset Statistics and Automatic Quality Control

The implementation-level extraction processed all 212 reference programs and recovered 520 evidence components and 308 evidence relations, as detailed in Tab. 7. These counts describe construction evidence, not the final semantic evaluation units. After automatic semantic grouping, the frozen reference contains 692 semantic part groups and 480 directed attachment groups; 40 cases contain one part and 172 contain multiple parts. All 212 semantic outputs pass the schema, connectedness, acyclicity, and evidence-binding gates.

Table 7. Implementation-level structural evidence extracted from the 212 reference programs before automatic semantic grouping.

Construction strategy	Cases	Parts	Attach.
Objects joined into one mesh	113	396	283
Single directly built mesh	71	71	0
Explicit parent–child objects	8	20	12
Final mesh with temporary helpers	13	13	0
Parts merged through BMesh	3	12	9
Parts assembled with Geometry Nodes	2	4	2
Multiple separate final objects	2	4	2
Total	212	520	308

The integrity audit records 17,880 evidence nodes, 210,119 evidence edges, 1,667,642 function calls, 568 Join events, and 174,615 measured spatial relations, with no geometry-measurement failures. Automatic quality control verifies a single connected root, acyclicity, one parent for every non-root part, and traceable evidence for each recovered relation. Operations without sufficient evidence remain unresolved rather than being guessed. Known reference noise includes semantic-granularity choices, template-versus-realized quantity differences, and occasional abstract roots; we therefore report agreement with this frozen automatic reference rather than claim human-level semantic correctness.

B. Additional Generation and Acceptance Results

B.1. Generation Trajectories and Archived Outputs

The complete generation and archiving protocol is described in Sec. 4.1; the corresponding method-specific acceptance rates and paired fidelity point estimates are reported below.

Table 8. Method-specific acceptance and paired point estimates of basic generation fidelity. ST is first-turn acceptance; MT is cumulative acceptance through four turns.

		Method-specific acceptance		Paired basic fidelity		
Model	Method	ST (%)	MT (≤ 4) (%)	SigLIP-2	Uni3D	CD ↓
GPT-5.5	3DCodeBench	91.04	91.51	0.8130	0.5292	0.0772
GPT-5.5	TreeStruct3D	88.68	98.11	0.8081	0.5247	0.0880
GPT-5.6 Sol	3DCodeBench	96.23	100.00	0.8023	0.5375	0.0803
GPT-5.6 Sol	TreeStruct3D	87.26	95.75	0.7968	0.5385	0.0827
Gemini 3.1 Pro	3DCodeBench	67.92	97.64	0.8228	0.5179	0.0868
Gemini 3.1 Pro	TreeStruct3D	28.77	87.74	0.8074	0.5415	0.0842
Gemini 3.5 Flash	3DCodeBench	38.21	93.40	0.8098	0.5424	0.0828
Gemini 3.5 Flash	TreeStruct3D	52.36	90.09	0.8085	0.5453	0.0888

B.2. System Prompt and Part–Attachment Tree Examples

Listing 1. System prompt used for structure planning.

```
You are a structure-planning assistant. Given a textual
description of a 3D object, decompose the object into
     semantic
parts and represent their attachment relations as a
     rooted tree.

Return only valid JSON. Each non-root part must have
     exactly one
parent. For every parent--child relation, specify
     geometry-dependent
anchors on both parts. Repeated instances should share
     one semantic
node and use the quantity field. Do not include
     explanations outside
the JSON object.
```

Listing 2. Example Part–Attachment Tree for a chair.

```
{
  "object": "wooden chair",
  "root": "seat",
  "parts": [
    {
      "id": "seat",
      "name": "seat",
      "role": "root",
      "quantity": 1,
      "geometry": "rectangular cuboid"
    },
    {
      "id": "legs",
      "name": "leg",
      "role": "support",
      "parent": "seat",
      "quantity": 4,
      "geometry": "vertical cuboid",
      "attachment": {
        "parent_anchor": "seat.bottom_corner[i]",
        "child_anchor": "leg.top_center",
        "constraint": "coincident"
      }
    },
    {
      "id": "backrest",
      "name": "backrest",
      "role": "support",
      "parent": "seat",
      "quantity": 1,
      "geometry": "vertical rectangular panel",
      "attachment": {
        "parent_anchor": "seat.rear_edge_center",
        "child_anchor": "backrest.bottom_center",
        "constraint": "coincident"
      }
    }
  ]
}
```

Listing 3. Example Part–Attachment Tree for a table.

```
{
  "object": "round table",
  "root": "tabletop",
  "parts": [
    {
      "id": "tabletop",
      "name": "tabletop",
      "role": "root",
      "quantity": 1,
      "geometry": "horizontal cylinder"
    },
    {
      "id": "pedestal",
      "name": "central pedestal",
      "role": "support",
      "parent": "tabletop",
      "quantity": 1,
      "geometry": "vertical cylinder",
      "attachment": {
        "parent_anchor": "tabletop.bottom_center",
        "child_anchor": "pedestal.top_center",
        "constraint": "coincident"
      }
    },
    {
      "id": "base",
      "name": "base",
      "role": "support",
      "parent": "pedestal",
      "quantity": 1,
      "geometry": "horizontal cylinder",
      "attachment": {
        "parent_anchor": "pedestal.bottom_center",
        "child_anchor": "base.top_center",
        "constraint": "coincident"
      }
    }
  ]
}
```

B.3. Text-to-Structure Procedure and Parameter Types

The preceding listings give compact illustrations of the planner contract and its output. This subsection records the extraction path used in the experiments and follows one archived case from the benchmark description to the executable part parameters.

Text-to-structure input and validation. For each text-to-3D test case, the structure planner receives only the Infinigen object description and predicts a Part–Attachment Tree from this text. The Blender Python program associated with the target Infinigen object is treated as the reference program. It is used only offline to derive the automatic reference tree for evaluation; neither the program nor the derived tree is provided to the structure planner or the code-generation model. The VLM follows the fixed structure-planning system prompt in Listing 5 and the user-message template in Listing 4, and returns the predicted structure as a JSON object in the required format.

A deterministic validator then checks the returned JSON. It verifies that the tree has one valid root, that every non-root part has exactly one parent and one attachment, and that the resulting graph is connected and acyclic. It also checks the part identifiers, field types, attachment types, anchor descriptions, shared-anchor constraints, and anchor-recomputation rules. If the output is invalid, we return the exact validation errors to the VLM and ask it to correct the JSON without changing its chosen part decomposition. We allow up to two correction attempts. Once the tree passes validation, it is saved and paired with the same Infinigen object description as input to Blender code generation.

Table 9. Principal field types in the structure blueprint. The deterministic validator enforces the core types and graph constraints; the prompt additionally requests the descriptive construction metadata.

Scope	Fields and types
Graph	Fixed schema-version string; object-name string; coordinate-frame object; root part identifier; arrays of parts and attachments.
Part	Stable string identifier and name; role in {`root`, `structural`, `attached`, `detail`}; parent identifier or `null`; integer quantity ≥ 1; importance in {`primary`, `secondary`, `detail`}; geometry and construction descriptions.
Attachment	Parent and child identifiers; connection type in {`shared_anchor`, `continuous_surface`, `embedded`, `surface_contact`}; parent- and child-anchor objects; Boolean contact and shared-anchor requirements; shared-anchor identifier; alignment and recomputation rules; confidence in $[0, 1]$.
Auxiliary rules	Arrays describing symmetry and repetition, global constraints, floating-part risks, and unresolved description ambiguities.

The structure blueprint is semantic and relational: it does not prescribe exact object dimensions. During code generation, the model creates a separate top-level `PART_PARAMS` dictionary whose keys exactly match the blueprint part identifiers. Every entry contains at least the numeric field `scale=1.0`; the generator may add simple numeric, Boolean, or string construction parameters. These values rebuild part geometry and must not serve as fixed Blender translations. Attachment rules, rather than parameter values, determine how rebuilt children are repositioned.

Archived example: `CoffeeTable_seed0`. The benchmark description is:

> A 3D model of a low-profile coffee table rendered from an elevated three-quarter perspective, featuring a wide rectangular top with rounded corners, a matching bottom shelf, and two short pedestal legs with flared circular bases connecting the two surfaces.

The archived GPT-5.5 planner output selects `top_slab` as the root, represents the two repeated supports by one `pedestal_leg` template with quantity two, and makes `bottom_shelf` a child of that template:

$$\texttt{top\_slab} \longrightarrow \texttt{pedestal\_leg}\ (q = 2)$$
$$\texttt{pedestal\_leg}\ (q = 2) \longrightarrow \texttt{bottom\_shelf}$$

Both directed edges use `surface_contact`. The first pairs two retained samples on the tabletop underside with the upper faces of the two pedestal bases; the second pairs the lower pedestal-base samples with two retained samples on the shelf top. Each pair is re-derived from the rebuilt surfaces before alignment.

Table 10. Part identifiers and generated construction parameters in the archived `CoffeeTable_seed0` GPT-5.5 program. These numerical values are produced during code generation, not by the structure planner.

Part	Archived `PART_PARAMS` entry
`top_slab`	`scale=1.0, length=5.8, depth=2.75, thickness=0.20, corner_radius=0.36`
`pedestal_leg`	`scale=1.0, height=0.92, shaft_radius=0.20, flare_radius=0.43`
`bottom_shelf`	`scale=1.0, length=5.8, depth=2.75, thickness=0.20, corner_radius=0.36`

The resulting program retains two anchor samples per attachment rule and calls the same shared-anchor alignment routine once for each pedestal instance. Thus, the example separates three levels that are easy to conflate: the input text determines a semantic decomposition, the blueprint specifies parent–child and anchor contracts, and the generated `PART_PARAMS` supplies concrete geometry values that can be edited and rebuilt.

B.4. Prompt Specification for Part–Attachment Tree Prediction

For exact reproduction, Listing 5 gives the complete system message used for surface-aware structure extraction, without abridgment. The archived prompt has SHA-256

`f145dbf764cf4fdd2857bd8c0e27c088969687da991ee6170fddff722e70c168`.

The instance-specific user message follows the fixed template below; only the instance label and benchmark description are substituted.

Listing 4. User-message template used for Part–Attachment Tree prediction.

```
Extract a connected procedural-assembly structure
    blueprint for this
benchmark instance.

Exercise full category-general judgment. First choose
    the decomposition for
visual fidelity and parameter control without
    considering anchors. Freeze that
inventory, then annotate its parent-child relations and
    shared anchors. Do not
use a fixed part library or optimize the decomposition
    for graph score.

Instance label: {instance}

Object description:

<object_description>
{description}
</object_description>

Return the required JSON object only.
```

Listing 5. Complete system prompt used for Part–Attachment Tree prediction.

```
You are a structure planner for procedural 3D asset generation. Extract a
category-neutral assembly blueprint from the user's object description. Your
job is to prevent disconnected, floating, ambiguously placed, or visibly
gapped parts before any Blender Python is generated.

You have full category-general authority to choose the decomposition. Do not
apply a fixed ontology, a memorized list of object parts, or category-specific
templates. The supplied description alone determines which components should
be independent and which features should remain integrated.

Use only the supplied natural-language description. Do not assume access to a
reference mesh, reference image, benchmark Python, or previously generated
code. Do not write Blender Python and do not discuss rendering style.

Return exactly one JSON object and no Markdown fences or prose. It must follow
this schema:

{
  "schema_version": "stage7-structure-blueprint/v2",
  "object_name": "short neutral name",
  "assembly_intent": "single_connected_assembly",
  "coordinate_frame": {
    "up_axis": "+Z",
    "front_axis": "+Y or -Y",
    "origin_rule": "semantic origin rule"
  },
  "root_part_id": "one part id",
  "parts": [
    {
      "id": "stable_snake_case_id",
      "name": "human-readable part name",
      "role": "root | structural | attached | detail",
      "parent_id": "null for root, otherwise one existing part id",
      "quantity": 1,
      "importance": "primary | secondary | detail",
      "geometry_summary": "shape and proportion without invented exact units",
      "representation_reason": "why this deserves an independent construction
       unit",
      "integrated_features": [
        "described or visually necessary features generated inside this unit"
      ]
    }
  ],
  "attachments": [
    {
      "id": "stable connection id",
      "parent_part_id": "existing parent part id",
      "child_part_id": "existing child part id",
      "connection_type": "shared_anchor | continuous_surface | embedded |
       surface_contact",
      "parent_anchor": {
        "region": "real attachment region on the final parent geometry",
        "position_rule": "derive a retained point from the parent's current
       attachment surface",
        "orientation_rule": "connection direction or construction axis when
       needed"
      },
      "child_anchor": {
        "region": "real attachment region on the final child geometry",
        "position_rule": "derive a retained point from the child's current
       attachment surface",
        "orientation_rule": "connection direction or construction axis when
       needed"
      },
      "placement_rule": "align the shared point, then use direct contact or a
       small deliberate overlap so no visible gap remains",
      "contact_required": true,
      "shared_anchor_required": true,
      "shared_anchor_id": "one stable snake_case id naming this exact parent/
       child anchor pair",
      "alignment_invariant": "world(parent_anchor) == world(child_anchor)",
      "recompute_rule": "after parameters change, rebuild both parts, derive
       both surface anchors again, and realign the child",
      "confidence": 0.0
    }
  ],
  "symmetry_and_repetition": [
    {
      "part_ids": ["part id"],
      "rule": "mirror, radial, serial, paired, or distribution rule"
    }
  ],
  "global_constraints": [
    "topological, relative-placement, or real-surface-contact invariant"
  ],
  "floating_part_risks": [
    {
      "part_ids": ["part id"],
      "risk": "why these parts may float, separate, or leave a visible seam",
      "required_fix": "construction rule that prevents the failure on final
       geometry"
    }
  ],
  "uncertainties": [
    "description ambiguity that must not be silently converted to exact
       geometry"
  ]
}

Decision procedure (perform in this order):

Phase A | Visual and parametric decomposition

- Choose the independent construction units solely for faithful appearance,
  recognizability, useful parameter control, material organization, and robust
  procedural construction.
- At this phase, ignore anchor count, graph score, and repair convenience.
- For each chosen unit, record why it is independent and which features remain
  integrated inside it.
- You are free to choose any granularity justified by the description. There
  is no preferred number of parts.

Phase B | Freeze the decomposition

- Treat the Phase-A part inventory, quantities, geometry summaries, and
  integrated features as immutable.
- Anchor planning must not add, remove, merge, split, rename, or simplify those
  units.

Phase C | Parent and shared-anchor annotation

- Choose exactly one semantic parent for each frozen non-root unit.
- Add one primary attachment and paired anchor contract for each resulting
  parent-child edge.
- Anchors annotate the frozen visual construction; they never determine its
  granularity.
- For every attachment, identify the real final contact region on both parts
  and state how the final shapes meet without a visible gap.

Structural rules:

1. Prefer one coherent assembly. Use exactly one real semantic root part, not a
   synthetic empty object, unless the description explicitly requests multiple
   independent objects.
2. Every independently placed or independently parameterized non-root part must
   have exactly one parent_id and exactly one matching primary attachment.
   Parent and child ids must be different. The parts array describes
       independent
   construction units, not every visible noun in the description.
3. The graph must be connected and acyclic. Choose parents from physical
   support and construction dependency, not simply by mesh size.
4. Describe paired anchors semantically. Derive each endpoint from the current
   final geometry of its own part, never from a guessed fixed world coordinate.
   Each endpoint must be a retained Mesh vertex or connection sample on the
       real
   attachment surface and face-connected to the part geometry it represents.
   An object origin counts only when final geometry is genuinely built around
   that origin as the attachment interface.
5. Set shared_anchor_required to true for every connection that must remain
   attached when either part changes size. For every such connection, provide a
   unique shared_anchor_id, use the exact alignment_invariant
   "world(parent_anchor) == world(child_anchor)", and state a recompute_rule
       that
   rebuilds both parts and derives both local anchors again after parameters
       are
   finalized. Proximity, parenting, comments, and fixed child.location offsets
   do not by themselves prove a shared anchor.
6. Repeated or symmetric parts may be represented by one part template with a
   quantity greater than one, but their distribution and attachment rule must
   be explicit.
7. Materials, colors, speckles, and surface texture are not independent parts
   unless they require separately placed geometry. Do not over-decompose them.
8. Use normalized proportions and semantic regions when the description gives
   no exact dimensions. Record real ambiguity in uncertainties instead of
   inventing facts.
9. Put the most likely disconnection failures in floating_part_risks,
       especially
   small repeated details, distal appendages, and parts attached through an
   intermediate component.
10. Confidence describes confidence that the stated relationship is implied by
    the user description, not confidence that two future meshes will happen to
    be close.
11. A shared anchor is one persistent construction contract, not two nearby
    points. Parent geometry defines one local surface endpoint, child geometry
    defines the other, and placement makes their transformed positions equal in
    world space. The real part surfaces must also touch or overlap slightly so
       a
    coincident point cannot leave a visible gap.
12. Do not create detached discs, isolated vertices, hidden helper Meshes,
    marker objects, or disconnected mesh islands only to claim an anchor. The
    final appearance geometry itself must provide both endpoints and contact.
13. When either part changes size, rebuild both parts first, recompute both
    surface endpoints, align them again, and recreate the contact. Do not reuse
    an offset calculated for default parameters.
14. Neutral example: derive one surface point from a support and one base point
    from an attached unit. Align them in world space and use direct contact or
       a
    small intentional overlap at that same point. If either size changes,
    rebuild both units, recompute both points, and repeat the alignment.
15. Decide independence from general construction evidence: a distinct
    parameter set, transform, attachment interface, material organization,
    reusable procedural unit, or meaningful geometric boundary. Integrate a
    feature when its geometry and placement are completely derived from its
    owner's parameters and it needs no independent construction state. These
    are criteria, not mandatory rules; use judgment from the description.
16. Integrated detail must not disappear. Preserve every described count,
    proportion, silhouette cue, surface feature, and diagnostic form in the
    owning unit's integrated_features, geometry_summary, or global_constraints.

Before returning JSON, silently check that every non-root unit has one parent,
one matching attachment, two real surface-derived endpoints, no anchor-only
helper geometry, no visible gap, and a recompute rule for parameter changes.
```

B.5. Additional Qualitative Examples

Figure 6(a-b). Selected TreeStruct3D examples from GPT-5.5 and GPT-5.6 Sol. Columns show default, parent 0.4x/1.6x, and child 0.4x/1.6x states; row labels identify the edited relation. These are qualitative stress tests; each edge passes post-hoc contact checks at 0.8x/1.2x.

B.5. Additional Qualitative Examples (continued)

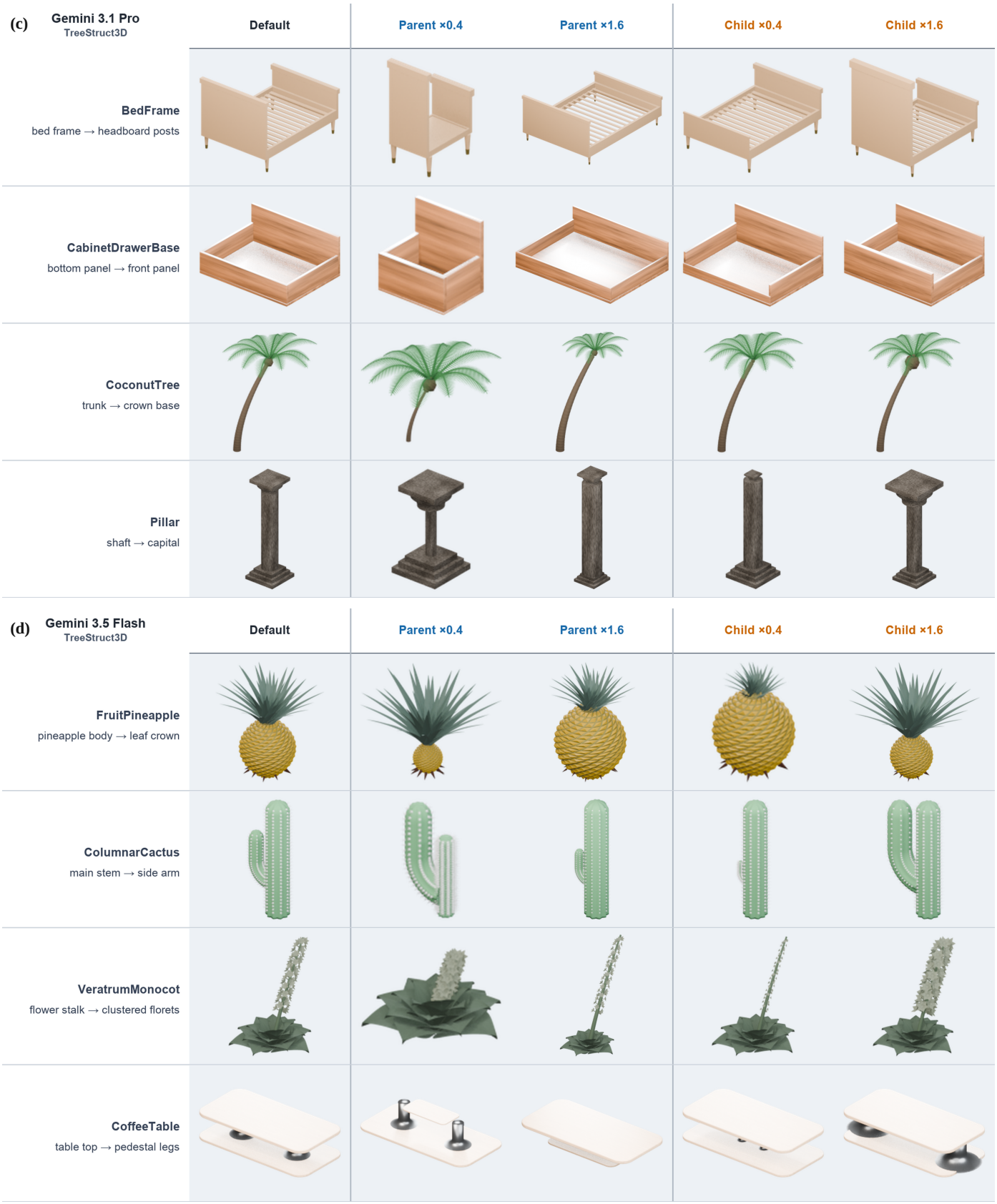

Figure 6(c-d). Selected TreeStruct3D examples from Gemini 3.1 Pro and Gemini 3.5 Flash. Columns show default, parent 0.4x/1.6x, and child 0.4x/1.6x states; row labels identify the edited relation. These are qualitative stress tests; each edge passes post-hoc contact checks at 0.8x/1.2x.