VETR: EXECUTION-GROUNDED SELF-IMPROVING AGENT FOR SYMBOLIC GRAPHICS REASONING

Anonymous authors
Paper under double-blind review

ABSTRACT

Visual agents for symbolic graphics reasoning depend on executable world representations, yet continued improvement remains bottlenecked by ambiguous supervision from final renders. Recent agentic systems refine scenes through repeated interaction, but they lack mechanisms to recursively extract actionable evidence from execution traces, overlooking which proposed symbolic edits survived, disappeared, or interfered during program execution. To address this, we propose VETR (Verified Execution Trajectories for Refinement), a framework that enables *agentic recursive self-improvement* over executable world representations for symbolic graphics reasoning. Within each task, VETR retains aligned program/render records with bounded runtime observations and introduces an *execution refiner* that compiles observed writes and state changes into constrained alternative programs. The refiner validates their endpoints and preservation conditions against captured runtime state; the ordinary visual verifier then assesses goal relevance. Repair construction makes no additional language-model calls, but can require additional executions. Evaluations on BlenderGym (280 tasks) and BlenderBench (27 released tasks) with GPT-5.6-luna and Qwen3.8-27B show that VETR consistently outperforms existing refinement strategies, including BlenderAlchemy and VIGA. Controlled ablations evaluate the contributions of execution refinement and role-asymmetric evidence organization under fixed generator-interaction caps.

1 INTRODUCTION

Large Language Model (LLM) agents have made remarkable strides in code generation, visual reasoning, and interactive editing by equipping LLMs with tools, planning, and self-refinement capabilities (Madaan et al., 2023; Shinn et al., 2023; Gou et al., 2024). Building on this progress, recent systems for symbolic graphics reasoning formulate scene understanding and editing as an iterative process where a generator proposes program modifications and a verifier evaluates the rendered output (Yin et al., 2026; Kuang et al., 2026). These systems demonstrate that agentic interaction improves over one-shot generation and that executable programs can serve as world representations specifying geometry, materials, lighting, and spatial relationships (Wang et al., 2026a). However, final-render feedback alone does not expose edit causality: an image comparison need not reveal which proposed symbolic edits survived execution, which were silently overwritten, or which interfered with shared scene resources.

This motivates an execution-evidence interface alongside visual feedback (Yin et al., 2026; Kuang et al., 2026). Additional iterations can supply new proposals, but do not directly expose effects omitted from the feedback interface. Logging alone also leaves open how observed values should become alternative programs (Wang et al., 2026a). We address this with supported update operators and explicit runtime preservation checks. Observed writes supply hypotheses about alternative executions, while visual verification remains responsible for assessing those alternatives against the task goal.

To address this, we propose **VETR** (Verified Execution Trajectories for Refinement), a framework for execution-grounded, within-task program refinement. VETR retains aligned executable records and uses a private execution refiner to construct alternatives from observed writes and state transitions. The refiner compiles supported numerical changes and compatible resource bindings, then

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

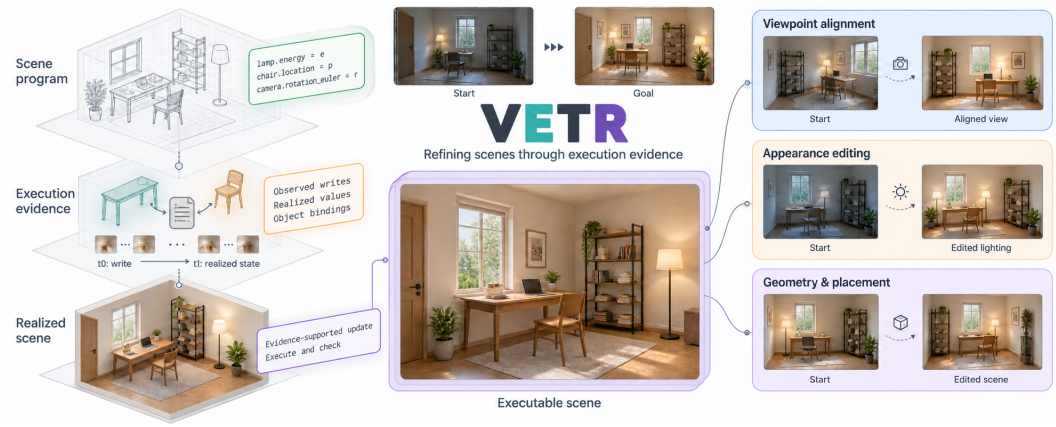

Figure 1: **Conceptual overview of VETR.** VETR refines scenes through execution-grounded evidence, bridging a scene program, its execution trace, and the realized 3D scene. Given a start and goal image, the system performs iterative updates guided by observed writes and realized bindings. This evidence-supported refinement targets viewpoint alignment, appearance (lighting/material) editing, and geometry and placement adjustments. Runtime checks test supported effects, while visual feedback guides subsequent proposals toward the goal.

checks their endpoints and preservation conditions by fresh execution, without additional language-model calls. It delivers a runtime-valid alternative when available, otherwise retaining the raw successful result or the preceding delivered record after failure. The ordinary visual verifier assesses the delivered result, and the generator conditions its next proposal on the available execution outcomes and feedback. This is the recursive program-revision loop depicted in Figure 1; Algorithms 1 and 2 specify its refiner and host loop.

We evaluate VETR on BlenderGym (280 editing tasks across five families) and BlenderBench (27 released tasks at three difficulty levels) with GPT-5.6-luna and Qwen3.8-27B. VETR is the strongest complete system on both benchmarks and both backbones, and paired comparisons confirm gains are not artifacts of completion rates. Ablations isolate the improvement to the execution refiner (disabling it reduces six-cell BlenderBench improvement from $+54.45\%$ to $+35.33\%$ on GPT and $+56.17\%$ to $+30.09\%$ on Qwen), and to role asymmetry and execution history (with GPT on BlenderGym, removing role asymmetry or limiting history raises $N\text{-}CLIP/PL$ errors by 12.7%/23.7% and 48.5%/79.0%, respectively). These results support contributions from execution refinement and evidence organization within the evaluated agent loop; fixed generator-interaction caps do not imply equal total execution cost.

Our contributions are summarized as follows:

- **Executable world representation for symbolic graphics reasoning.** We build on scene representation work to establish executable programs as the foundation for iterative scene editing, enabling direct manipulation of geometry, materials, lighting, and spatial relationships.
- **Agentic recursive self-improvement with an execution refiner.** We formulate scene refinement as a propose–execute–observe–verify cycle and design a refiner that compiles execution-trace evidence into bounded program updates under preservation constraints, without additional LLM inference (Madaan et al., 2023; Shinn et al., 2023; Gou et al., 2024).
- **Empirical validation on graphics editing benchmarks.** We evaluate on BlenderGym and BlenderBench across two backbones, showing consistent gains over VIGA (Yin et al., 2026) and BlenderAlchemy, with controlled ablations evaluating execution refinement and context organization within the host loop.

2 METHOD

VETR refines complete scene programs through a *propose–execute–observe–verify* loop. A private execution refiner constructs alternatives from bounded runtime observations and checks their real-

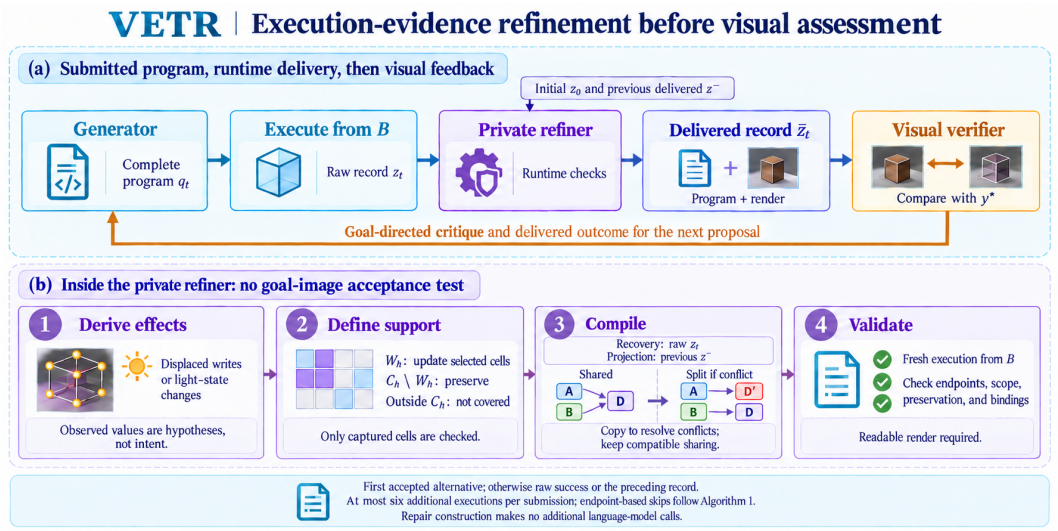

Figure 2: **Execution-evidence refinement before visual assessment.** Each complete program executes from B. The private refiner constructs alternatives, compiles supported changes and compatible bindings, and checks their realization relative to the candidate's reference record. For the numerical channel, W_h contains selected cells, $\mathcal{C}_h \setminus W_h$ contains captured cells to preserve, and properties outside $\mathcal{C}_h$ are not covered by that contract. Goal-image comparison occurs only in the ordinary visual verifier. Delivery, shortcuts, and the six-execution cap are specified in Algorithm 1. Repair construction makes no additional language-model calls.

ization before ordinary visual verification. Runtime acceptance and visual goal assessment serve different purposes: the former checks a candidate's stated effects and preservation conditions; the latter guides subsequent editing toward the goal.

2.1 Executable World Representation

A task supplies a base scene B, an editable initial program p_0, a start image r_0, and a goal image $y^\star$. Following code-based scene representations (Wang et al., 2026a), the generator submits a complete Python program q_t, conditioned on the task and its retained interaction history. Complete means executable relative to the supplied objects and assets in B, not a standalone scene constructor. Each submission and repair executes afresh from B; the initial program is not run as an extra preparation step before each submission.

The *raw execution record* $z_t = (q_t, r_t, S_t, \tau_t)$ associates the submitted program with its render, captured state, and observed writes. The captured state is $S_t = (\theta_t, \Pi_t, G_t)$, where θ_t stores supported numerical properties, Π_t stores object-to-resource references, and G_t stores captured scene structure. The bounded write trace is

$$\tau_t = \left[(o_i, k_i, v_i, \ell_i)\right]_{i=1}^{m_t}, \tag{1}$$

where an event records the owner resolved at execution time, a supported property path, its observed value, and its source location. The observer covers supported assignments in the submitted source, including local helpers; imported-library writes and unresolved aliases do not provide owner-qualified events. Unobserved effects remain unknown. Execution failures retain available observations and diagnostics, with unavailable record fields left undefined.

We distinguish z_t from the *delivered record* $\bar{z}_t$, which may contain a repaired program, the raw result, or the preceding delivered record z^-. Generated program/render pairs retain their common provenance. Initial observations (S_0, τ_0) are obtained by executing p_0 once without an additional render. We denote the initial reference by $z_0 = (p_0, r_0, S_0, \tau_0)$, marking r_0 as the supplied visual anchor, not a render from that observation pass. The refiner takes z_0 so that both initial writes and their final endpoints are available (Appendix B).

2.2 EXECUTION REFINER

The refiner examines every submitted execution, including successful proposals whose images already improve. Planning actions do not invoke it. It uses z_t, the initial observations, and z^- to construct and validate alternative complete programs without additional language-model calls.

Candidate effects from execution evidence. For a supported non-light cell (o, k) and source execution $s \in \{t, 0\}$, define its *displaced-value indices* in τ_s as the observed writes whose values differ from that source's final endpoint. The recovery endpoint is the latest such value:

$$\begin{aligned} \mathcal{I}_s(o,k) &= \{i : o_i = o,\ k_i = k,\ v_i \not\simeq \theta_s(o,k)\}, \\ \widehat{v}_s(o,k) &= v_{\max \mathcal{I}_s(o,k)} \quad \text{when } \mathcal{I}_s(o,k) \neq \varnothing. \end{aligned} \tag{2}$$

Here $\simeq$ denotes the runtime checker's numerical agreement predicate. Current and initial traces supply separate recovery hypotheses, both applied to the current complete program. The light-recovery channel similarly reconstructs observed owner-local energy/color requests displaced by later shared writes, resetting earlier requests after explicit data rebinding. These values are alternative endpoints, not inferred user intent or proof that a later assignment was erroneous.

In the ordered controller, if no recovery candidate has been accepted, the light channel considers eligible full and half effective changes between S^-, the previous delivered state, and S_t. For owners in the submitted edit's supported scope, the *projected light endpoint* is

$$u_{\alpha,o} = \phi(S^-(o)) + \alpha\big[\phi(S_t(o)) - \phi(S^-(o))\big], \qquad \alpha \in \{1, \tfrac{1}{2}\}, \tag{3}$$

where $\phi(E, c) = Ec$ is the compiler's effective-value map for light energy E and color c. Projection modifies the previous delivered program, retaining its other captured state. It therefore differs from recovery, which modifies the raw proposal. The map supplies algebraic parameter targets, not a guarantee about the rendered image.

Compatible values and resource ownership. Each candidate h specifies a reference record z_h^{ref}, required endpoints, and permitted changes. Recovery uses $z_h^{\mathrm{ref}} = z_t$; projection uses $z_h^{\mathrm{ref}} = z^-$. The compiler appends a patch to that reference program, producing q'_h. It retains reference values for other captured cells and makes permitted local resource copies when shared owners require incompatible endpoints. Compatible sharing is retained. Non-light compilation restores supported values directly; light compilation uses a bounded reconstruction of ϕ and compatible owner tuples. Because ϕ is not one-to-one, its formula alone does not specify the reconstruction rule (Appendix B).

Runtime realization and preservation. The compiled program executes afresh from B. For the numerical-property channel, let $\mathcal{C}_h$ be the reference's captured supported cells, $W_h \subseteq \mathcal{C}_h$ the selected write support, and d_h its required numerical endpoints. Fresh readback must satisfy

$$\theta'_h(c) \simeq \begin{cases} d_h(c), & c \in W_h, \\ \theta_h^{\mathrm{ref}}(c), & c \in \mathcal{C}_h \setminus W_h. \end{cases} \tag{4}$$

Additional checks cover captured node/mesh structure, protected object frames, and render context. Light candidates are checked against their captured owner tuples, effective endpoints, and permitted reference changes. The captured cells $\mathcal{C}_h \setminus W_h$ are known values to preserve; properties outside $\mathcal{C}_h$ are not covered by this numerical contract. Channel-specific checks constrain the refiner's changes relative to its reference, not every change made by the generator, and are not relaxed after failure.

Validation and delivery. The fixed order is current/initial non-light recovery, current/initial light recovery, then full-/half-effect projection. Unsupported alternatives or those lacking required evidence cannot pass the contract. Algorithm 1 makes the six-additional-execution cap explicit and returns the first accepted alternative. It also retains endpoint-based skip rules: raw reuse as full-effect projection requires that candidate's complete runtime contract relative to z^-. If the endpoint matches but those checks fail, the reported branch skips the compiled full-effect program rather than executing it. Such a skip is not proof of an equivalent rerender; Appendix B explains its implications. If no alternative is accepted, retain a successful raw result, otherwise reuse the exact preceding record. Raw fallback is not certification under a repair contract.

The ordinary visual verifier then evaluates the delivered result against $y^\star$. Passing a runtime contract neither requires nor guarantees lower image error. The recursion is within-task program revision: subsequent proposals use the delivered result and retained feedback, without a monotonic-improvement guarantee or a claim that each program encodes the entire history.

2.3 Role-Asymmetric Evidence Organization

The refiner's internal evidence and the agents' visible contexts are distinct. The refiner consumes observed writes, captured endpoints, and resource bindings. The generator retains its actions and delivered tool outcomes in a bounded multimodal history, with 24 recent messages by default and the task/planning inputs pinned separately. The visual verifier receives the current delivered comparison and returns a goal-directed critique; its default context is reset rather than sharing the generator's accumulated transcript. This organization does not require either agent to interpret the complete private trace. The three-message and shared-memory variants change these context policies (Appendix G).

Algorithm 2 places execution refinement inside the host loop and distinguishes the generator-interaction cap from additional repair executions. At width $N > 1$, each executed candidate is refined before the visual tournament selects a delivered outcome. Final evaluation uses the last successful selected render from a completed run, not the best historical image; failures do not replace its associated program. The supplied start image alone does not constitute a successful generated answer (Appendix N).

3 Experiments

3.1 Experiment Setup

We evaluate the complete VETR system on two established test suites, BlenderGym (Gu et al., 2025) and BlenderBench (Yin et al., 2026), which evaluate program-based 3D graphics editing across diverse visual modification tasks. Throughout the main benchmark results, VETR denotes the full system with execution refinement enabled by default; the otherwise matched variant with refinement disabled is reported only in the ablation study.

Tasks and systems. BlenderGym contains 280 editing tasks: 85 blend-shape, 50 placement, 55 geometry, 45 lighting, and 45 material tasks (Gu et al., 2025). Each task supplies an initial program, start image, and goal image. BlenderBench contains 27 released tasks across three difficulty levels, with nine tasks per level. We compare One-Shot (direct generation from the initial program and visual target without iterative feedback), BlenderAlchemy, VIGA, and VETR. GPT-5.6-luna uses three separate 280-task cohorts: systems, context ablations, and scene origin/guidance. All GPT-5.6-luna and Qwen3.8-27B runs use Blender 3.6.0, and every paired task shares a runtime. VIGA/VETR allow ten generator interactions. Each ablation changes only its named setting; system comparisons retain each system's instructions and execution conventions. Table 16 maps these cohorts to the full results and configurations.

Measurements. We report negative CLIP similarity $N\text{-}CLIP = 1 - \langle \hat{f}(r), \hat{f}(y^\star) \rangle$, using unit-normalized CLIP ViT-B/32 embeddings, and photometric loss PL, the RGB mean squared error in $[0, 1]$ (Radford et al., 2021). Lower is better; tables scale both by 100. The same stored answer renders also provide PSNR, SSIM, AlexNet LPIPS, Canny edge F1, and CLIP similarity (Wang et al., 2004; Zhang et al., 2018). For PSNR, SSIM, edge F1, and CLIP similarity, higher is better, whereas for LPIPS, lower is better. PSNR is computed per image before averaging; CLIP and $N\text{-}CLIP$ are algebraically related. Edge F1 uses thresholds 100/200 on 8-bit grayscale and a two-pixel Chebyshev matching tolerance. The evaluated iterative wrapper renders Camera1; the evaluator averages matched views when more than one is available. Appendix L provides all definitions. Paired comparisons use the tasks valid for both systems. For each, we report a 95% task-level bootstrap confidence interval (5,000 resamples of tasks with replacement, seed 0) for the relative reduction in mean error, and an exact two-sided sign test on per-task wins with ties excluded; a difference is *statistically significant* when the interval excludes zero and $p < .05$ (Appendix J).

3.2 BLENDERGYM

We first evaluate the complete VETR system against one-shot generation and iterative baselines across diverse editing families. These system comparisons characterize visual quality under the reported configurations; the component ablations separately test execution refinement and evidence organization within the host loop.

Table 1 reports the complete-system results. VETR achieves the strongest overall performance on both GPT-5.6-luna and Qwen3.8-27B, and consistently outperforms One-Shot, VIGA, and BlenderAlchemy across the aggregate metrics and nearly all family-level comparisons. On GPT, VETR attains the best PL and $N\text{-}CLIP$ in four of the five editing families, with material editing as the only exception; on Qwen, it achieves the best result in every reported family–metric cell. These results are particularly notable in light of the original VIGA study (Yin et al., 2026), which reported substantial gains over a GPT-4o One-Shot baseline. In our experiments with newer and stronger backbones, however, the gap between iterative visual refinement and One-Shot becomes considerably smaller and can even reverse. A plausible explanation is that improvements in foundation-model capabilities have substantially strengthened one-shot program generation, reducing the headroom available to refinement strategies that rely primarily on rendered visual feedback. In contrast, VETR continues to improve substantially over these stronger One-Shot outputs. Rather than depending on an initially weak program that can be corrected through repeated visual inspection, VETR exploits execution evidence to identify which symbolic edits were actually realized, overwritten, or disrupted, and recursively incorporates this information into subsequent program states. These gains are statistically significant against One-Shot on both backbones: VETR lowers $N\text{-}CLIP/PL$ by 52.7%/58.8% on GPT and by 28.1%/49.6% on Qwen, and on Qwen it also improves over VIGA by 40.0%/56.3%. In each case the bootstrap interval excludes zero and the sign test gives $p \leq .0043$ (Appendix J, Table 11).

Table 2 complements the primary error metrics with pixel, structural, perceptual, and semantic similarity measures, reporting VETR at $N = 4$. On GPT-5.6-luna, its improvements over One-Shot satisfy the paired significance criterion for all five metrics on 214 jointly valid tasks (sign tests $p \leq 2.2 \times 10^{-12}$). Against VIGA, edge F1 and LPIPS improve by 5.6% and 18.5%, respectively, on 279 jointly valid tasks; both pass the criterion ($p = .0030$ and $.0227$). Thus the gains extend beyond photometric error to edge agreement and learned perceptual similarity. CLIP provides a similarity-scale view of the primary $N\text{-}CLIP$ score, rather than an independent metric (Appendix L, Table 14).

On Qwen3.8-27B, VETR has the best reported means for PSNR, SSIM, LPIPS, and CLIP, and ties One-Shot on edge F1 at the displayed precision. These means use each system's valid tasks, so we assess significance on matched subsets (Table 14). At $N = 4$, all five metrics improve significantly over VIGA on 264 paired tasks; against One-Shot, SSIM, LPIPS, and CLIP meet the criterion on 250 tasks, whereas PSNR and edge F1 do not. No extended metric establishes an advantage of $N = 4$ over $N = 1$, separating the evidence for refinement from that for increased candidate breadth.

3.3 BLENDERBENCH

BlenderBench is a more demanding testbed than BlenderGym, with camera adjustment, multi-step editing, and compositional tasks that introduce stronger dependencies between successive scene modifications. These settings create more opportunities for intermediate errors to propagate, making them particularly suitable for evaluating whether VETR can preserve reliable refinement under increasingly entangled execution trajectories.

Across the 27 released BlenderBench tasks, VETR achieves the best result in every level–metric cell on both GPT-5.6-luna and Qwen3.8-27B, outperforming One-Shot and VIGA throughout the benchmark. Relative to VIGA on the same tasks, VETR reduces $N\text{-}CLIP/PL$ by 55.0%/67.7% on GPT and 27.6%/48.6% on Qwen, showing that its advantage remains substantial even as task complexity increases.

The strongest separation appears on GPT Level 3, the most compositional setting, where VETR lowers $N\text{-}CLIP/PL$ by 72.4%/75.2% relative to VIGA. This is precisely the regime in which execution-grounded refinement is most useful: when multiple edits interact, a failure may arise not only from an incorrect visual judgment but also from symbolic writes that are overwritten, only partially realized, or otherwise altered during execution. By retaining the execution trace and converting

Table 1: **Visual editing performance on BlenderGym across editing families.** PL and $N\text{-}CLIP$ are scaled by 100 (↓). Parentheses indicate negative improvement.

Model	Setting	All		Shape		Placement		Geometry		Lighting		Material		Impr. (%)↑
		$PL\downarrow$	$N\text{-}CLIP\downarrow$	$PL\downarrow$	$N\text{-}CLIP\downarrow$	$PL\downarrow$	$N\text{-}CLIP\downarrow$	$PL\downarrow$	$N\text{-}CLIP\downarrow$	$PL\downarrow$	$N\text{-}CLIP\downarrow$	$PL\downarrow$	$N\text{-}CLIP\downarrow$	
GPT-5.6-luna	One-Shot	1.389	2.583	0.460	1.629	0.955	2.966	4.350	5.754	1.906	1.555	2.109	**5.237**	–
	BlenderAlchemy	5.675	9.032	1.176	3.567	0.895	4.993	10.506	14.553	9.969	11.421	11.429	17.588	(236.65)
	VIGA	1.354	4.017	0.265	0.864	0.551	1.608	4.242	10.977	0.916	1.161	**1.276**	7.155	16.92
	VETR	**1.034**	**2.397**	**0.246**	**0.775**	**0.527**	**1.270**	**2.914**	**4.660**	**0.495**	**0.681**	1.326	5.661	**41.22**
Qwen3.8-27B	One-Shot	2.160	3.520	0.522	1.849	0.923	2.820	3.956	6.201	1.748	1.697	5.978	7.412	–
	BlenderAlchemy	2.370	4.000	0.654	2.132	0.815	2.380	6.462	7.450	2.292	2.366	2.429	6.794	(9.94)
	VIGA	2.950	4.750	0.804	1.938	0.760	2.390	8.055	11.528	2.755	2.605	4.585	8.192	(31.35)
	VETR	**1.330**	**2.910**	**0.452**	**1.375**	**0.686**	**2.115**	**3.557**	**5.666**	**0.747**	**1.080**	**1.630**	**5.225**	**30.44**

Table 2: **Complementary image-similarity metrics.** Higher is better for PSNR, SSIM, Edge F1, and CLIP; lower is better for LPIPS.

GPT-5.6-luna

System	PSNR↑	SSIM↑	Edge F1↑	LPIPS↓	CLIP↑
One-Shot	25.17	0.884	0.773	0.127	0.974
BlenderAlchemy	19.00	0.787	0.614	0.266	0.928
VIGA	28.18	0.911	0.812	0.087	0.978
VETR	**30.85**	**0.924**	**0.840**	**0.069**	**0.988**

Qwen3.8-27B

System	PSNR↑	SSIM↑	Edge F1↑	LPIPS↓	CLIP↑
One-Shot	23.84	0.838	**0.729**	0.171	0.965
BlenderAlchemy	22.84	0.817	0.692	0.192	0.960
VIGA	22.05	0.795	0.683	0.207	0.952
VETR	**24.48**	**0.841**	**0.729**	**0.151**	**0.971**

witnessed state changes into bounded program updates, VETR can correct these failures before they cascade into later steps. In contrast, render-only refinement has less direct evidence about which intermediate edits actually survived execution.

Every iterative row completes all 27 tasks; GPT One-Shot renders 21/27, and its unrendered tasks are excluded from its reported means.

Table 3: **Visual editing on native BlenderBench by level.** Improvement is the mean relative reduction from the same-backbone One-Shot row over the six level–metric cells, as defined in Eq. (5).

Model	Setting	Level 1		Level 2		Level 3		Impr.
		$PL\downarrow$	$N\text{-}CLIP\downarrow$	$PL\downarrow$	$N\text{-}CLIP\downarrow$	$PL\downarrow$	$N\text{-}CLIP\downarrow$	↑
GPT-5.6-luna	One-Shot	10.9403	14.8074	2.7780	4.4044	14.3415	26.3542	—
	VIGA	11.2304	14.7821	5.3481	5.0353	21.7647	22.8812	(24.65%)
	VETR	**6.3523**	**10.1947**	**0.6519**	**2.7021**	**5.3922**	**6.3201**	**+54.45%**
Qwen3.8-27B	One-Shot	12.9096	19.3977	3.9448	5.1091	18.5005	36.1053	—
	VIGA	9.4124	14.2022	4.5780	4.8665	13.6108	15.1571	+21.17%
	VETR	**7.7325**	**11.6530**	**0.2585**	**3.9672**	**6.1927**	**9.1418**	**+56.17%**

The improvement column averages the six level–metric cells against the same-backbone One-Shot row. Let $e_{b,\ell,m}$ denote the error for baseline b, level ℓ, and metric m, and let $e_{s,\ell,m}$ denote the corresponding system error. We report

$$\text{Improvement}(s) = \frac{100}{6} \sum_{\ell=1}^{3} \sum_{m \in \{N\text{-}CLIP, PL\}} \left(1 - \frac{e_{s,\ell,m}}{e_{b,\ell,m}}\right) \%, \tag{5}$$

which weights the three levels and two metrics equally. Each method's level means use only the tasks it rendered.

Qualitative comparison. Figure 3 compares three selected final outputs, not intermediate repair interventions. Values here follow the figure's $PL \times 100$ scale. On the displayed light task, VETR has error < 0.0001 versus VIGA's 0.0387; on geometry, < 0.0001 versus 0.7820; on shape, 0.0801 versus 0.0938. The marked regions illustrate differences in illumination, placement, and facial shape. These examples support the stated final-image comparisons, but do not identify which repair produced a difference or independently verify preservation of scene properties. The refiner's contribution is evaluated by the controlled on/off comparison in Section 3.4.

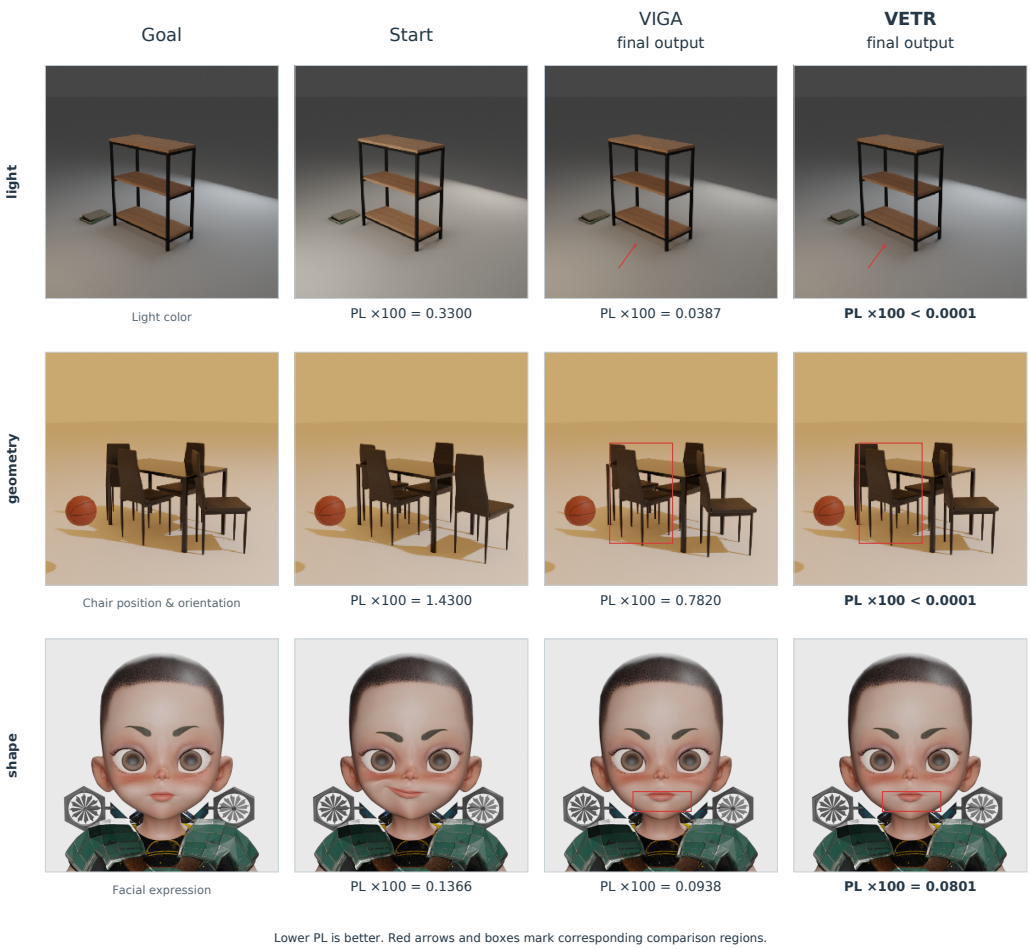

Figure 3: **Qualitative comparison of visual editing.** Columns display the goal, start, VIGA final output, and VETR final output across three editing categories: light (color), geometry (chair position & orientation), and shape (facial expression). Red arrows and boxes highlight local regions of comparison. Reported values denote PL $\times 100$ (lower is better). VETR achieves near-zero PL in the displayed light and geometry tasks and lower error in the displayed shape task. These output comparisons illustrate visual fidelity, not an independent verification of the refiner's runtime mechanism.

3.4 Ablation Study

The ablation study examines three design choices: (i) adding the execution refiner within a fixed generator-interaction cap; (ii) using separate generator and verifier contexts rather than shared memory; and (iii) retaining a longer generator history. These interventions test the refiner and its surrounding evidence interface without assuming that both agents consume the private runtime trace.

Execution refinement. We first isolate the execution-refiner stage, which converts observed writes and state transitions into runtime-validated updates. Disabling only this stage holds prompts, tools, base-scene execution policy, backbone, and the ten-generator-interaction cap fixed. The enabled stage can still perform additional repair executions.

Disabling refinement degrades performance in 11 of the 12 level–metric cells across the two backbones (Table 4), often by substantial margins. On GPT, removing refinement increases both PL and $N\text{-}CLIP$ errors at every difficulty level, with the largest degradation on Level 3: PL rises from 5.3922 to 11.8475 and $N\text{-}CLIP$ from 6.3201 to 15.2108. The same pattern holds on Qwen, where disabling refinement substantially increases errors on Levels 1 and 3 and raises Level 2 PL from

0.2585 to 2.9924. Paired statistics over the 27 GPT tasks confirm the effect: refinement lowers mean PL by 41.8%, which is statistically significant (bootstrap interval $[6.7, 49.0]$ excludes zero; sign test 24/3 wins, $p = 4.9 \times 10^{-5}$), and lowers mean $N\text{-}CLIP$ by 32.2% with a bootstrap interval of $[14.4, 55.5]$ (Appendix J, Table 12).

These results support adding execution refinement within the same generator-interaction cap. Constructing a repair makes no additional language-model call, but validation can add up to six execution attempts per submitted program. The comparison therefore holds the generator budget fixed, not total computation.

Table 4: **Ablation.** Left: execution refinement on the 27 released BlenderBench tasks, using GPT-5.6-luna and Qwen3.8-27B. Right: context ablations on 280 BlenderGym tasks with GPT-5.6-luna, from a separate run in which all rows are paired on the same tasks. Errors are scaled by 100 (↓); paired statistics are in Appendices J and G.

Model	Setting	Level 1		Level 2		Level 3	
		$PL\downarrow$	$N\text{-}CLIP\downarrow$	$PL\downarrow$	$N\text{-}CLIP\downarrow$	$PL\downarrow$	$N\text{-}CLIP\downarrow$
GPT-5.6-luna	VETR (default)	**6.3523**	**10.1947**	**0.6519**	**2.7021**	**5.3922**	**6.3201**
	w/o refinement	8.3768	10.4058	1.0725	2.7410	11.8475	15.2108
Qwen3.8-27B	VETR (default)	**7.7325**	**11.6530**	**0.2585**	3.9672	**6.1927**	**9.1418**
	w/o refinement	11.3198	13.8822	2.9924	**3.3747**	12.3879	18.5392

Setting	$N\text{-}CLIP\downarrow$	$PL\downarrow$
VETR (default)	**2.6030**	**0.9963**
w/o role asymmetry	2.9347	1.2327
w/o visual verifier	2.7742	1.2776
w/o full history	3.8647	1.7838

Component-wise ablation. We ablate three design choices in one 280-task GPT-5.6-luna cohort, pairing each variant with the default on the same tasks (Table 4, right; statistics in Appendix G): a memory shared by generator and verifier (**w/o role asymmetry**), no visual verifier, and a generator limited to its three most recent messages instead of 24 (**w/o full history**). Limiting history degrades most, raising $N\text{-}CLIP/PL$ by 48.5%/79.0%; both increases are statistically significant, with bootstrap intervals excluding zero and sign tests $p < 10^{-5}$, which supports retaining the longer generator history. Removing role asymmetry raises them by 12.7%/23.7%, also statistically significant on both metrics (bootstrap intervals exclude zero; sign tests $p \leq .003$), supporting separate role contexts. Full statistics are in Appendix G (Table 6). Removing the verifier raises them by 6.6%/28.2%, and the default wins on more tasks (sign tests $p \leq .016$; Appendix G).

4 Discussion and Conclusion

VETR treats executions as evidence for revising an executable scene. Authoritative records identify the result being refined; observed writes and state transitions expose how proposed changes become realized effects. The refiner turns selected observations into executable updates by jointly choosing supported values and compatible bindings, then validates the resulting program against runtime contracts before visual verification. On native BlenderBench, adding this execution stage lowers aggregate visual error on both evaluated model backbones. Recorded interventions show how the same evidence-to-program pipeline operates inside ordinary visual reasoning, while the host-loop studies characterize the surrounding execution and context interfaces.

Scope and transfer. The BlenderGym history and execution-origin contrasts cover broad target-view editing; the native BlenderBench comparison evaluates the complete closed-loop refiner on released tasks. Runtime validation establishes realization and preservation only for the captured properties. Goal relevance is assessed by the ordinary visual verifier, allowing subsequent proposals to build on the delivered result. For another domain, such as SVG or CAD, the same principle requires observable program effects, supported update operators, and an explicit preservation contract. The current refiner operates on a fixed set of write types derived from the Blender execution model, so extending it to new domains requires defining additional write grammars and preservation boundaries. Future work could explore learned preservation models that infer coupling constraints directly from execution traces.

REFERENCES

Mingdeng Cao, Xintao Wang, Zhongang Qi, Ying Shan, Xiaohu Qie, and Yinqiang Zheng. MasaCtrl: Tuning-free mutual self-attention control for consistent image synthesis and editing. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 22560–22570, 2023. URL https://openaccess.thecvf.com/content/ICCV2023/html/Cao_MasaCtrl_Tuning-Free_Mutual_Self-Attention_Control_for_Consistent_Image_Synthesis_and_ICCV_2023_paper.html.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching Large Language Models to Self-Debug. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/2460396f2d0d421885997dd1612ac56b-Abstract-Conference.html.

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*, volume 36, pp. 18225–18250. Curran Associates, Inc., 2023. doi: 10.52202/075280-0802. URL https://arxiv.org/abs/2305.15393.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. CRITIC: Large Language Models Can Self-Correct with Tool-Interactive Critiquing. In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2305.11738.

Yunqi Gu, Ian Huang, Jihyeon Je, Guandao Yang, and Leonidas Guibas. BlenderGym: Benchmarking Foundational Model Systems for Graphics Editing. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 18574–18583, 2025. URL https://arxiv.org/abs/2504.01786.

Guangzhao He, Rundong Luo, Wei-Chiu Ma, and Hadar Averbuch-Elor. Thinking in blender: Staged executable inverse graphics with vision-language models. *arXiv preprint arXiv:2606.02580*, 2026. URL https://arxiv.org/abs/2606.02580.

Amir Hertz, Ron Mokady, Jay Tenenbaum, Kfir Aberman, Yael Pritch, and Daniel Cohen-Or. Prompt-to-prompt image editing with cross attention control. In *International Conference on Learning Representations*, 2023. URL https://arxiv.org/abs/2208.01626.

Yuanzhe Hu, Yu Wang, and Julian McAuley. Evaluating Memory in LLM Agents via Incremental Multi-Turn Interactions. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/fd1eff9dd295df50a41f2521942fa31d-Abstract-Conference.html.

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*, volume 235, pp. 19252–19282. PMLR, 2024. URL https://proceedings.mlr.press/v235/hu24g.html.

Ian Huang, Guandao Yang, and Leonidas Guibas. BlenderAlchemy: Editing 3D Graphics with Vision-Language Models. In *Computer Vision – ECCV 2024*, pp. 297–314. Springer Nature Switzerland, 2024. doi: 10.1007/978-3-031-73024-5_18. URL https://dx.doi.org/10.1007/978-3-031-73024-5_18.

Xuan Ju, Ailing Zeng, Yuxuan Bian, Shaoteng Liu, and Qiang Xu. PnP inversion: Boosting diffusion-based editing with 3 lines of code. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/661caac7729aa7d8c6b8ac0d39ccbc6a-Abstract-Conference.html.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):66:1–66:34, 2018. doi: 10.1145/3158154. URL https://plv.mpi-sws.org/rustbelt/popl18/paper.pdf.

Zhengfei Kuang, Rui Lin, Long Zhao, Gordon Wetzstein, Saining Xie, and Sanghyun Woo. VULCAN: Tool-Augmented Multi Agents for Iterative 3D Object Arrangement. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 23763–23773, 2026. URL https://openaccess.thecvf.com/content/CVPR2026/html/Kuang_VULCAN_Tool-Augmented_Multi_Agents_for_Iterative_3D_Object_Arrangement_CVPR_2026_paper.html.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative Refinement with Self-Feedback. In *Advances in Neural Information Processing Systems*, volume 36, pp. 46534–46594. Curran Associates, Inc., 2023. doi: 10.52202/075280-2019. URL https://arxiv.org/abs/2303.17651.

Mikaël Mayer, Viktor Kunčak, and Ravi Chugh. Bidirectional evaluation with direct manipulation. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):127:1–127:28, 2018. doi: 10.1145/3276497. URL https://lara.epfl.ch/~kuncak/papers/MayerETAL18BidirectionalEvaluation.pdf.

Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: Program repair via semantic analysis. In *Proceedings of the International Conference on Software Engineering*, pp. 772–781, 2013. doi: 10.1109/ICSE.2013.6606623. URL https://research.ibm.com/publications/semfix-program-repair-via-semantic-analysis.

Merlin Nimier-David, Delio Vicini, Tizian Zeltner, and Wenzel Jakob. Mitsuba 2: A retargetable forward and inverse renderer. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 38(6), 2019. doi: 10.1145/3355089.3356498. URL https://rgl.epfl.ch/publications/NimierDavidVicini2019Mitsuba2.

Yulin Pan, Xiangteng He, Chaojie Mao, Zhen Han, Zeyinzi Jiang, Jingfeng Zhang, and Yu Liu. ICE-Bench: A unified and comprehensive benchmark for image creating and editing. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 16586–16596, 2025. URL https://openaccess.thecvf.com/content/ICCV2025/html/Pan_ICE-Bench_A_Unified_and_Comprehensive_Benchmark_for_Image_Creating_and_ICCV_2025_paper.html.

Artemis Panagopoulou, Honglu Zhou, Silvio Savarese, Caiming Xiong, Chris Callison-Burch, Mark Yatskar, and Juan Carlos Niebles. ViUniT: Visual Unit Tests for More Robust Visual Programming. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 24646–24656, 2025. URL https://openaccess.thecvf.com/content/CVPR2025/html/Panagopoulou_ViUniT_Visual_Unit_Tests_for_More_Robust_Visual_Programming_CVPR_2025_paper.html.

Pixar Animation Studios. Universal Scene Description: Scenegraph Instancing, 2023. URL https://openusd.org/23.05/api/_usd__page__scenegraph_instancing.html. OpenUSD 23.05 documentation. Accessed September 20, 2026.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning Transferable Visual Models From Natural Language Supervision. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139, pp. 8748–8763. PMLR, 2021. URL https://proceedings.mlr.press/v139/radford21a.html.

John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pp. 55–74, 2002. doi: 10.1109/LICS.2002.1029817. URL https://www.cs.cmu.edu/~jcr/seplogic.pdf.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652. Curran Associates, Inc., 2023. doi: 10.52202/075280-0377. URL https://arxiv.org/abs/2303.11366.

Chunyi Sun, Junlin Han, Weijian Deng, Xinlong Wang, Zishan Qin, and Stephen Gould. 3D-GPT: Procedural 3D Modeling with Large Language Models. In *2025 International Conference on 3D Vision (3DV)*, pp. 1253–1263. IEEE, 2025. doi: 10.1109/3dv66043.2025.00119. URL https://dx.doi.org/10.1109/3DV66043.2025.00119.

Narek Tumanyan, Michal Geyer, Shai Bagon, and Tali Dekel. Plug-and-play diffusion features for text-driven image-to-image translation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1921–1930, 2023. URL https://pnp-diffusion.github.io/.

Hanyang Wang, Yimo Cai, Weiliang Chen, Jiawei Chi, Haowen Sun, Qiyu Dai, Yi-Hsin Hung, Xingzhuo Guo, Jinshan Ren, Runmao Yao, Ziwei Liu, Mingsheng Long, Yueqi Duan, Jun Gao, Jiangran Lyu, Fangfu Liu, and Jialong Wu. Code as worlds: Agentic discovery of executable world representations for physical reasoning. *arXiv preprint arXiv:2608.27549*, 2026a. URL https://arxiv.org/abs/2608.27549.

Hao Wang, Wenhui Zhu, Shao Tang, Zhipeng Wang, Xuanzhao Dong, Xin Li, Xiwen Chen, Ashish Bastola, Xinhao Huang, Yalin Wang, and Abolfazl Razi. EZBlender: Efficient 3D Editing with Plan-and-ReAct Agent. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV) Workshops*, pp. 1343–1352, 2026b. URL https://openaccess.thecvf.com/content/WACV2026W/VALED/html/Wang_EZBlender_Efficient_3D_Editing_with_Plan-and-ReAct_Agent_WACVW_2026_paper.html.

Su Wang, Chitwan Saharia, Ceslee Montgomery, Jordi Pont-Tuset, Shai Noy, Stefano Pellegrini, Yasumasa Onoe, Sarah Laszlo, David J. Fleet, Radu Soricut, Jason Baldridge, Mohammad Norouzi, Peter Anderson, and William Chan. Imagen editor and EditBench: Advancing and evaluating text-guided image inpainting. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 18359–18369, 2023. URL https://imagen.research.google/editor/.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable Code Actions Elicit Better LLM Agents. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 50208–50232. PMLR, 2024. URL https://proceedings.mlr.press/v235/wang24h.html.

Zhou Wang, Alan C. Bovik, Hamid R. Sheikh, and Eero P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

Xueqing Wu, Zongyu Lin, Songyan Zhao, Te-Lin Wu, Pan Lu, Nanyun Peng, and Kai-Wei Chang. VDebugger: Harnessing execution feedback for debugging visual programs. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 9845–9860, 2024. doi: 10.18653/v1/2024.findings-emnlp.575. URL https://aclanthology.org/2024.findings-emnlp.575/.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-Mem: Agentic Memory for LLM Agents. In *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 17577–17604. Curran Associates, Inc., 2025. doi: 10.52202/085713-0593. URL https://arxiv.org/abs/2502.12110.

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 (CVPR)*, pp. 16227–16237, 2024. URL https://arxiv.org/abs/2312.09067.

Shaofeng Yin, Jiaxin Ge, Zora Zhiruo Wang, Chenyang Wang, Xiuyu Li, Michael J. Black, Trevor Darrell, Angjoo Kanazawa, and Haiwen Feng. Vision-as-Inverse-Graphics Agent via Interleaved Multimodal Reasoning. *arXiv preprint arXiv:2601.11109*, 2026. URL https://arxiv.org/abs/2601.11109.

Kai Zhang, Lingbo Mo, Wenhu Chen, Huan Sun, and Yu Su. MagicBrush: A manually annotated dataset for instruction-guided image editing. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://osu-nlp-group.github.io/MagicBrush/.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Y Zou, and Kunle Olukotun. Agentic Context Engineering: Evolving Contexts for Self-Improving Language Models. In *International Conference on Learning Representations*, 2026. URL https://arxiv.org/abs/2510.04618.

Richard Zhang, Phillip Isola, Alexei A. Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *CVPR*, 2018.

Yunzhi Zhang, Zizhang Li, Matt Zhou, Shangzhe Wu, and Jiajun Wu. The Scene Language: Representing Scenes with Programs, Words, and Embeddings. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 24625–24634, 2025. URL https://openaccess.thecvf.com/content/CVPR2025/html/Zhang_The_Scene_Language_Representing_Scenes_with_Programs_Words_and_Embeddings_CVPR_2025_paper.html.

AI USE STATEMENT

Generative AI assisted code inspection, literature organization, manuscript and figure preparation, and statistical scripting. Experimental measurements come from the recorded runs and stored renders. The authors are responsible for the final claims, references, analyses, and artifacts.

REPRODUCIBILITY STATEMENT

The supplement provides execution and context settings, full paired tables, metric definitions, and per-task provenance. The supplied scripts and recorded programs, renders, and outcomes support regeneration of the reported analyses. The native BlenderBench results retain per-level measurements for both model backbones; the GPT records additionally retain per-task scores, task identities, source hashes, and runtime counters for all 27 tasks of every iterative arm and the 21 rendered One-Shot tasks.

A Related Work

Executable representations and program revision. Programs provide compositional control over scene structure: LayoutGPT, 3D-GPT, Holodeck, and SceneCraft generate scenes via language models (Feng et al., 2023; Sun et al., 2025; Yang et al., 2024; Hu et al., 2024); Scene Language unifies programs with semantic and visual representations (Zhang et al., 2025); Code-as-World treats executable worlds as revisable hypotheses (Wang et al., 2026a). For iterative editing, BlenderAlchemy, VIGA, and VULCAN combine rendered feedback, visual memory, and geometric constraints (Huang et al., 2024; Yin et al., 2026; Kuang et al., 2026); EZBlender adds local validation (Wang et al., 2026b); SEIG imposes stage budgets and rollback (He et al., 2026). These systems establish feedback loops but lack mechanisms to extract alternatives from execution traces. VETR fills this gap by compiling observed writes and bindings into bounded updates validated before the next proposal.

Agent memory, execution evidence, and preservation. Feedback-driven refinement is studied by Self-Refine, Reflexion, CRITIC (Madaan et al., 2023; Shinn et al., 2023; Gou et al., 2024), Self-Debugging, and CodeAct (Chen et al., 2024; Wang et al., 2024); VDebugger leverages stepwise traces (Wu et al., 2024); visual unit tests guide revision (Panagopoulou et al., 2025); and agent-memory research examines experience reuse (Hu et al., 2026; Xu et al., 2025; Zhang et al., 2026). Preservation is a complementary requirement: Prompt-to-Prompt, Plug-and-Play, and MasaCtrl maintain structure or appearance (Hertz et al., 2023; Tumanyan et al., 2023; Cao et al., 2023); PIE-Bench, EditBench, MagicBrush, and ICE-Bench evaluate preservation alongside editing (Ju et al., 2024; Wang et al., 2023; Zhang et al., 2023; Pan et al., 2025).

Execution-guided updates and local reasoning. SemFix uses symbolic execution for repair synthesis (Nguyen et al., 2013); bidirectional evaluation propagates outputs backward through functional programs (Mayer et al., 2018). Unlike these methods targeting internal values, VETR derives candidates from witnessed imperative executions against an image objective, requiring no additional model inference. Separation logic and RustBelt motivate local state reasoning (Reynolds, 2002; Jung et al., 2018); we instantiate this as finite runtime checks over captured properties. Graphics systems like Mitsuba 2 and OpenUSD provide the foundation of reusable, interacting scene resources (Nimier-David et al., 2019; Pixar Animation Studios, 2023), within which VETR performs recursive within-task program revision using execution-grounded evidence.

B Execution-refiner implementation details

B.1 Compiling trace evidence into realized edits

The execution refiner converts bounded observations into constrained executable updates between raw execution and visual verification (Figure 2). It constructs candidate effects from earlier writes displaced during execution or effective changes between runtime states, then compiles them into parameter updates and permitted resource-ownership changes. The generator still proposes complete programs with its original prompt and tool schema. The refiner examines every submitted execution, including improving raw renders; planning actions do not invoke it. No additional language model identifies an error or constructs a repair.

Observe the write, its owner, and its fate. Alongside $\mathrm{Execute}(B, q_t)$, the observer records

$$\tau_t = \big[(o_i, k_i, v_i, \ell_i)\big]_{i=1}^{m_t}, \qquad S_t = (\theta_t, \Pi_t, G_t). \tag{6}$$

Here o_i is the owner bound at the executed statement, k_i a supported numerical property path, v_i its observed value, and ℓ_i its source location. Final numerical values, object-to-resource references, and captured structure reside in θ_t, Π_t, and G_t, respectively. Resolving owners during execution distinguishes calls to the same submitted helper that write different objects. The observer covers supported assignments in submitted source, including local helpers; imported-library writes and unresolved aliases do not provide owner-qualified candidate evidence. A supported standalone assignment is read back at the next eligible Python trace event, observing application-level coercion without evaluating its right-hand side twice. Unsupported bindings and incomplete observations remain unknown.

Initial reference and required evidence. The public initial program p_0 is executed once for (S_0, τ_0) without another render. We write $z_0 = (p_0, r_0, S_0, \tau_0)$ for this initial reference, explicitly marking r_0 as the supplied visual anchor rather than the output of this observation pass. Initial recovery needs both τ_0 and the initial endpoint map θ_0 in S_0; Algorithm 1 therefore takes z_0, matching its caller. Subsequent delivered program/render pairs retain the provenance of their generated render. Reusing the initial anchor after failure is not a successful generated answer.

Recovery requires a source write, its source endpoint, and a resolvable target in the reference program. Projection requires the in-scope light observations in both S_t and S^-. A missing required observation cannot establish a candidate's endpoint or preservation condition; unavailable evidence is not assigned a default value. The exact supported-path, owner-matching, and missing-data policies remain implementation details identified in Section B.1.

Lost values become competing hypotheses. For source execution $s \in \{t, 0\}$ and a supported non-light cell (o, k), the recovery rule selects

$$\widehat{v}_s(o,k) = v_{i^\star}, \qquad i^\star = \max\{i \text{ in } \tau_s : o_i = o,\ k_i = k,\ v_i \not\approx \theta_s(o,k)\}, \tag{7}$$

when this set is nonempty. The current and initial traces supply separate source hypotheses. Both propose restoring eligible cells in the current complete program; initial-source recovery does not rewind the whole program to p_0. The per-cell rule supplies candidate values, not by itself a complete definition of how eligible cells are grouped into a write support.

The light channel reconstructs observed owner-local energy/color requests displaced by later shared writes, resetting earlier requests after explicit data rebinding. This reset requires ordered binding evidence: the final reference map Π_s alone does not describe when a binding changed. Last-write-wins defines the executed result but does not identify the program that best matches a new visual goal. Displaced values remain hypotheses, not inferred intent or proof that a later assignment was erroneous.

Compile values and compatible ownership. For a recovery hypothesis, let W_h be the supported cells to change and $d_h(o, k)$ their required endpoints; other captured cells retain their raw endpoints. Shared owners with incompatible requirements require permitted local copying before writes. Compatible requirements do not force a split. A shared geometry-node group, for example, cannot realize two different required radii through one shared value; lamps with compatible requests may retain shared data.

The light compiler uses the effective-value map $\phi(E, c) = Ec$ to enumerate bounded scalar-write supports and group owners by compatible required tuples. Non-light compilation restores supported values directly and detaches conflicting material, mesh, camera, or node-group references where supported. The map ϕ does not uniquely determine the light reconstruction: (E, c) and $(2E, c/2)$ have the same effect whenever both are admissible. Consequently, the bounds, selected tuple components, ordering of feasible reconstructions, and allowed copying/rebinding must come from the compiler rather than being inferred from ϕ.

Project observed effects into executable updates. In the ordered controller of Algorithm 1, projection is considered if no recovery candidate has been accepted, subject to the projection channel's evidence and scope requirements. For each in-scope owner o, Eq. (3) forms $u_{\alpha,o}$ using S^- and S_t, with $\alpha = 1$ before $\alpha = \frac{1}{2}$. These endpoints come from executions, not hidden target parameters. Projection compiles into the previous delivered program while retaining its other captured state; recovery instead uses the raw program. Thus, candidate construction, compilation, and the decision to execute a compiled program are distinct steps. The endpoint-based shortcuts below can skip a constructed candidate.

Preserve a bounded frame around the repair. Each compiled program runs afresh from B, not a scene mutated by a previous trial. For numerical-property *recovery*, the reference is the raw record and the check is

$$\theta'_h(o,k) \simeq \begin{cases} d_h(o,k), & (o,k) \in W_h, \\ \theta_t(o,k), & (o,k) \in \mathcal{C}_h \setminus W_h. \end{cases} \tag{8}$$

This is Eq. (4) specialized to recovery, not a different reference rule for projection. The captured complement $\mathcal{C}_h \setminus W_h$ is known and checked; properties outside $\mathcal{C}_h$ are not covered by this numerical

contract. Further checks retain captured node/mesh structure, protected object frames, and render context. Light candidates are checked against their owner tuples, effective endpoints, and allowed reference changes. Every check uses the candidate's designated reference: raw for recovery, previous delivered for projection. This finite operational frame follows the local-reasoning distinction between changed and retained state (Reynolds, 2002; Jung et al., 2018); it is not a guarantee about uncaptured properties or visual quality.

A single acceptance contract, including raw reuse. Write $\mathsf{Valid}_h(z')$ for the conjunction of successful execution, a readable render, realization of all required endpoints, preservation relative to z_h^{ref}, and applicable scope/binding checks. Required unknown values cannot make this predicate true. It is the same acceptance obligation for a fresh trial and for reusing the raw record as a projection candidate. In particular, raw reuse under full-effect projection must satisfy preservation relative to z^-, not merely agree with its own numerical endpoints. This notation specifies the contract; it does not supply undocumented numerical tolerances or an implementation of the checks.

Endpoint shortcuts and the skipped full-effect candidate. The supplied controller skips a light-recovery trial when its required endpoint is already realized by the raw record. For full-effect projection, it returns the raw record only if the endpoint-match test and $\mathsf{Valid}_h(z)$ both pass. If the endpoint matches but the contract fails, it *continues to the next candidate without executing the compiled full-effect program.* Algorithm 1 retains this branch; it is not silently changed to fall through to execution.

The distinction is substantive. Under the effective-endpoint interpretation,

$$u_{1,o} = \phi(S^-(o)) + [\phi(S_t(o)) - \phi(S^-(o))] = \phi(S_t(o)).$$

Hence, whenever these values are available, the raw light effect matches by construction. If the shortcut tests only this effect, the full-effect stage acts as a raw-reuse check or skip, rather than a fresh full-effect repair. A check of a more restrictive required owner tuple is a different predicate and must be specified explicitly. We do not infer such a tuple check from the effective-value equation.

As a schematic example, let a previous record have light energy 1 and a protected object position $x = 0$, with fixed light color. A raw proposal with energy 2 and $x = 5$ realizes the full-effect light target, but violates a projection that must retain $x = 0$. A compiled program with energy 2 and $x = 0$ is not equivalent to the raw program. Skipping it is therefore a restriction of the search policy, not proof that rerendering would be redundant. The endpoint-match predicates need implementation confirmation (Section B.1).

Execution cap, delivery, and visual feedback. The order is current/initial non-light recovery, current/initial light recovery, then full-/half-effect projection. Supported alternatives are constructed on demand. The counter b in Algorithm 1 makes the reported limit explicit: at most six *additional execution attempts* per submitted program, including failed attempts. Enumerating scalar supports during compilation is not itself an execution. The counter does not assume that each channel necessarily supplies exactly one alternative.

A successful fresh trial passing Valid_h is returned immediately. When no candidate is accepted, a successful raw execution is retained; execution/render failure returns the exact preceding delivered record. Raw fallback is not a claim that the raw program passed a repair contract. Diagnostics for a failed submission are kept distinct from a cached render's provenance. The ordinary visual verifier subsequently compares the delivered program/image with the goal. No image-error decrease is required by runtime acceptance, and no additional language-model call constructs the repair.

Specification boundary. The equations and listing above clarify the recorded procedure but do not fully specify its implementation-dependent selectors. Exact reproduction still requires the supported-property registry, source-to-reference owner matching, selection/grouping of W_h, and the rule selecting projection scope. The light inverse additionally requires admissible bounds, required tuple components, feasible-solution ordering, allowed copies/rebindings, and infeasibility handling. Ordered rebinding observations cannot be recovered from the final Π_s alone; their recording and alignment with numerical writes must be specified. Finally, the numerical agreement predicate, missing-readback behavior, shortcut endpoint tests, and recovery-to-projection eligibility must be tied to the evaluated code. No undocumented values, selectors, or tie-breaking rules are assumed

Algorithm 1 Execution-evidence refinement: reported skip policy and explicit execution cap

Require: base B, raw record z, initial reference z_0, previous delivered record z^-

$b \leftarrow 0$ ▷ additional executions for this submission
Inspect available writes, endpoints, binding evidence, and supported edit scope
for each supported alternative h in the fixed recovery/projection order **do**
    if a required observation or supported compilation is unavailable **then**
        continue
    end if
    Set $z_h^{\mathrm{ref}} \leftarrow z$ for recovery, or z^- for projection
    Compile q'_h and its endpoint, preservation, scope, and binding requirements
    if h is light recovery and its endpoint-match predicate holds on z **then**
        continue ▷ reported endpoint-based skip
    end if
    if h is full-effect projection and its endpoint-match predicate holds on z **then**
        if $\mathsf{Valid}_h(z)$ **then** ▷ full projection contract relative to z^-
            return z
        end if
        continue ▷ skip q'_h; do not fall through to execution
    end if
    if $b = 6$ **then**
        break
    end if
    $b \leftarrow b + 1$; $z' \leftarrow \textsc{ExecuteAndTrace}(B, q'_h)$
    if $\mathsf{Valid}_h(z')$ **then**
        return z'
    end if
end for
if the raw program executes successfully and produces a readable render **then**
    return z
end if
return z^- ▷ reuse the exact preceding record

Table 5: **Recorded cost in the GPT Blender 3.6 cohort.** Means use tasks with saved cost records (n); completion is reported separately. Quality for both widths is reported in Table 9.

Variant	n	Interactions	Model calls	Blender renders	Agent time (s)
VETR $N = 1$	277	8.74	18.58	7.16	204
VETR $N = 4$	280	8.76	66.17	28.56	733

here. In particular, the ordered presentation of projection after unaccepted recovery must not be substituted for a different eligibility gate in the actual controller.

Making an edit observable. Let $R_B(q)$ denote the rendered result of a complete program. Distinct programs can satisfy $q_1 \neq q_2$ yet $R_B(q_1) \simeq R_B(q_2)$ when execution erases their differing intermediate values. Their traces may still supply different alternatives. Conversely, multiple parameterizations of a light effect can differ in surrounding state. Execution evidence supplies bounded hypotheses whose realization can be tested; the visual verifier assesses their goal relevance. An improving raw image does not establish that every observed request survived, motivating examination of every submitted execution without asserting that each repair improves the image.

C Execution-interface diagnostics and cost

C.1 Diagnostic study: consistent realization of proposals

To diagnose execution consistency under the complete-program interface, this study retains full generator history in both arms and changes the scene loaded for execution. Rebuild loads the task's initial scene each time; persistent scene loads and saves a job-local scene across rounds. Both use GPT, $N = 1$, the same verifier, a complete-program interface, and ten generator interactions. This separate run set completes 278/280 tasks in each arm, with 276 jointly valid answers.

Rebuilding reduces mean N-$CLIP$ by 33.9%, with a 95% interval of $[19.8, 45.8]$ and 154 wins, 120 losses, and two ties. The effect is concentrated in geometry, with a 53.4% reduction; Appendix H reports every family.

A deterministic replay isolates one mechanism. Executing the same saved `geometry20` program nine times on a persistent scene accumulates nine modifiers and reduces mesh volume to $1/256$ of its first-execution value. Rebuild keeps one modifier and constant volume. Explicitly replacing previous modifiers on the persistent scene also prevents accumulation (Appendix P). Thus, the requirement is consistency between a program and its execution origin; a compatible incremental or replacement interface can also retain scene state.

C.2 Refinement cost and candidate breadth

With an interaction cap T and candidate width N, at most TN raw candidate programs enter the execution stage. Each submission permits at most $R = 6$ additional repair executions, so the raw-program/repair stage has a worst-case bound of $(1 + R)TN$ execution attempts, plus the separate initial-program observation. This is an upper bound rather than a reconstruction of recorded cost counters. Planning actions, unsupported candidates, failures, and no-op shortcuts can reduce execution or render counts. A tournament over k available delivered renders uses $k - 1$ pairwise comparisons. The refiner itself makes no additional language-model calls; the generator, ordinary verifier, and tournament have their own inference costs. The generator's visible history is bounded by its message window, while archived trajectories grow with recorded interactions.

Table 5 makes the cost of wider search explicit: $N = 4$ uses about $3.6\times$ the model calls and $4.0\times$ the renders of $N = 1$. The core contrasts use $N = 1$.

Algorithm 2 Complete refinement procedure with optional candidate selection

Require: task inputs $x = (B, p_0, r_0, y^\star)$, interaction cap T, width N, history window L_G
Initialize generator history H with its prompt and task inputs
Obtain z_0 by observing p_0 without rerendering; set $z^- \leftarrow z_0$
for $j = 1, \ldots, T$ **do**
    Sample N actions from the generator's window $W_G(H; L_G)$
    Parse actions and choose the most frequent valid action type
    if no valid action **then**
        Append parsing feedback; log candidates; **continue**
    end if
    if the chosen action produces renders **then**
        for each candidate a of the selected type, with complete program q_a **do**
            $z_a \leftarrow \textsc{ExecuteAndTrace}(B, q_a)$
            $\tilde{z}_a \leftarrow \textsc{Refine}(B, z_a, z_0, z^-)$ ▷ Algorithm 1
        end for
        Select one delivered outcome by its render, otherwise use diagnostics
        Set z^- to the selected delivered record, when available
    else
        Execute one instance of the chosen non-render action
    end if
    if no outcome is returned **then**
        Append failure feedback; log candidates; **continue**
    end if
    Obtain visual-verifier feedback when requested by the selected tool
    Append selected action and delivered outcome to H; log all candidates
    if the chosen action is end **then**
        break
    end if
end for
Evaluation: score the last successful selected generated render
    under Appendix N; cached images retain their original provenance

D Complete refinement procedure

Algorithm 2 expands the single-submission procedure to both candidate widths. At $N = 1$, action voting and the visual tournament reduce to the single available response. At $N > 1$, only candidates of the selected action type are executed. Each render-producing candidate passes through Algorithm 1 before visual candidate selection. All candidates in an interaction use the same previous delivered record; only the selected outcome advances that record. The tournament compares delivered candidates, whereas runtime acceptance inside the refiner uses no visual-improvement gate. Planning and inspection actions consume generator interactions but do not invoke the refiner.

The returned record and failure diagnostics have distinct provenance. If an attempted script fails and the refiner returns a cached record, the cached render remains paired with its own program, not with the failed submission. The supplied initial image is a reference, not a successful generated render. Runs with no successful generated render, and timed-out or incomplete runs, remain invalid under the existing answer rule.

E Complete system results

Table 1 reports family-level means and completion counts in the main text. The complete paired comparisons in Appendix J provide the head-to-head analyses. The descriptive improvement column in Table 1 is

$$\mathrm{Impr}(m) = \frac{100}{10} \sum_{k=1}^{10} \frac{x_k^{\text{One-Shot}} - x_k^m}{x_k^{\text{One-Shot}}}, \tag{9}$$

where k indexes the five families and two benchmark metrics. GPT uses the single Blender 3.6 system cohort, including BlenderAlchemy with four candidates per interaction. Qwen retains its separately documented configuration (Appendix K).

GPT completion and paired quality. One-Shot completes 214 tasks; failures concentrate in geometry (23/55 valid) and material (13/45). The $N = 1$ comparison therefore uses the 212 tasks completed by both systems, and $N = 4$ uses all 214 One-Shot tasks. Table 10 reports the matched One-Shot reference for each intersection. Relative reductions are 40.3%/56.3% (N-$CLIP$/PL) for $N = 1$ and 52.7%/58.8% for $N = 4$. Both variants improve each extended metric on their corresponding pairs.

Comparison with iterative systems. Against VIGA, $N = 1$ reduces N-$CLIP$ by 36.3% (CI $[24.3, 46.4]$, 161 wins and 114 losses, $p = .0054$). At $N = 4$, the mean reductions are 40.6% in N-$CLIP$ and 24.6% in PL. Both widths improve both primary metrics relative to BlenderAlchemy.

Qwen: gains over VIGA extend to a second backbone. On Qwen3.8-27B, N=1 reduces N-$CLIP$/PL by 31.9%/50.8% on 261 paired tasks, and N=4 by 40.0%/56.3% on 264; all four comparisons meet both criteria. Against One-Shot, N=4 establishes 28.1%/49.6% reductions (250 tasks), and N=1 yields 22.1%/43.6% (248 tasks). Both widths outperform BlenderAlchemy on both metrics. Completion is 251/280 for One-Shot, 265 for VIGA, 279 for BlenderAlchemy, 276 for N=1, and 279 for N=4.

Additional image metrics. Table 2 summarizes image similarity, while Table 14 reports matched effects, confidence intervals, and win counts. On GPT-5.6-luna, both VETR variants improve all five metrics significantly over One-Shot; against VIGA, the $N = 1$ CLIP comparison and the $N = 4$ edge F1 and LPIPS comparisons meet the paired criterion. On Qwen3.8-27B, $N = 1$ establishes gains in PSNR, LPIPS, and CLIP over VIGA, and in LPIPS over One-Shot. At $N = 4$, all five comparisons with VIGA and the SSIM, LPIPS, and CLIP comparisons with One-Shot meet the criterion. Neither backbone establishes an extended-metric advantage of $N = 4$ over $N = 1$

F EDITING STRATEGIES

The examples below characterize the archived program trajectories under the One-Shot prompt. On the 100 geometry and material tasks, the One-Shot runner and Blender 3.6 runtime are fixed. We characterize the observed editing behavior by the fraction of initial-program lines retained in the generated program (Appendix M, Table 15). GPT-5.6-luna retains a median 10% of the start lines: 92 of 100 programs keep less than half, 46 begin by deleting the scene's objects or clearing the node tree, 65 contain a binding that exists only in Blender 4.x (`"Specular IOR Level"` 44, `"Lacunarity"` 13, `BLENDER_EEVEE_NEXT` 12); 66 fail, 40 of them with such a binding as the first exception. Figure 4(b) shows this strategy on one task. The observed strategy changes with the full configuration: inside VETR's loop, whose prompt asks for one to three edits over bindings present in the program or supplied by the verifier, GPT-5.6-luna's *first-round* program retains a median 96% of the start lines on the same 100 tasks (89 above 90%), 4 first-round programs contain a 4.x binding, 2 fail to render, and the line completes 100/100 (Figure 4(c)).

G COMPLETE CONTEXT ABLATIONS

The context ablations form one cohort: Blender 3.6.0, GPT-5.6-luna, N=1, ten generator interactions, and 280 attempted tasks per arm. The default run and three variants differ only in the named switch, and quality metrics were computed only after sampling had finished. Every arm has a valid answer on all 280 tasks, so each comparison is paired on the full benchmark. Table 6 reports the paired statistics, computed with the procedure of Appendix J.

Execution history. Limiting the generator to its three most recent messages produces the largest effect: the default has 32.7%/44.2% lower N-$CLIP$/PL than the ablated arm, equivalently the ablated errors are 48.5%/79.0% higher than the default. Both metrics meet the paired criterion.

(a) Start program (`geometry28`, 86 lines; excerpt)

```
from infinigen.core.nodes.node_wrangler import Nodes
from infinigen.core import surface

def shader_material_002_002(nw):
    bsdf = nw.new_node(Nodes.PrincipledBSDF,
      input_kwargs={'Base Color': (0, .0086, .8, 1)})
    ...
def geometry_nodes(nw):
    subdiv = nw.new_node(Nodes.SubdivideMesh,
      input_kwargs={'Mesh': ..., 'Level': 7})
    noise = nw.new_node(Nodes.NoiseTexture)
    ramp = nw.new_node(Nodes.ColorRamp,
      input_kwargs={'Fac': noise.outputs["Fac"]})
    ramp.color_ramp.elements[0].position = 0.0
    ramp.color_ramp.elements[1].position = 1.0
    delete = nw.new_node(Nodes.DeleteGeometry,
      input_kwargs={'Geometry': subdiv,
                    'Selection': ramp.outputs["Color"]})
apply(bpy.context.active_object)
```

(b) GPT-5.6-luna One-Shot: 293-line rewrite, no render

```
import bpy, math, random
from mathutils import Vector
# Reset the scene geometry ...
for obj in list(bpy.data.objects):
    if obj.type == "MESH":
        bpy.data.objects.remove(obj, do_unlink=True)
mat = bpy.data.materials.new("FragmentBlue")
nodes = mat.node_tree.nodes
nodes.clear()
bsdf = nodes.new("ShaderNodeBsdfPrincipled")
bsdf.inputs["Base Color"].default_value = ...
bsdf.inputs["Specular IOR Level"].default_value = .28
# ... 240 lines: fragments built by hand with bpy
scene = bpy.context.scene
scene.render.engine = "BLENDER_EEVEE_NEXT"
```

Retains 4% of the start lines. Blender 3.6 raises `KeyError: "Specular IOR Level" not found` at the incompatible socket binding; there is no second attempt, so the task is lost.

(c) GPT-5.6-luna inside VETR $N{=}1$: rounds 1–9

```
# round 1 -- verifier: "volume not perforated";
#   smallest safe test: lower the subdivision level
    subdiv = nw.new_node(Nodes.SubdivideMesh,
      input_kwargs={'Mesh': ..., 'Level': 4})
# round 2 -- render unchanged: the noise has no
#   spatial input, so wire object position into it
    pos = nw.new_node(Nodes.InputPosition)
    noise = nw.new_node(Nodes.NoiseTexture,
    input_kwargs={'Vector': pos.outputs["Position"]})
# rounds 3-9 -- move the two ramp stops, one per round
    ramp.color_ramp.elements[0].position = 0.40
    ramp.color_ramp.elements[1].position = 0.55
```

Round-1 program retains 99% of the start lines; 9 of 9 rounds render; answer N-$CLIP$ 14.1 ($N{=}4$: 8.7; VIGA: 16.8).

Figure 4: **Editing strategy and execution outcome on the same task.** (a) The supplied procedural program. (b) Under the One-Shot prompt, GPT-5.6-luna rewrites the scene and accesses a socket absent in Blender 3.6, so this attempt yields no render. Orange highlights the rewrite and incompatible identifiers. (c) With VETR's prompt and iterative feedback, GPT preserves most of the program and makes the edits highlighted in blue. Comments summarize observed changes and feedback; ellipses omit unchanged or lengthy code. Scores are N-$CLIP \times 100$ (↓).

Table 6: **Context ablations in one cohort** (GPT-5.6-luna, $N{=}1$, Blender 3.6.0, 280 tasks). Means are scaled by 100. Red.: relative reduction of the default mean error against the ablated mean (positive favors the default); the main text instead expresses each ablated error relative to the default. CI: 95% task bootstrap (5,000 resamples, seed 0). W/L/T: default wins/losses/ties. p: exact two-sided sign test excluding ties. †: CI excludes zero and $p < .05$.

Variant	Metric	n	Default	Ablated	Red. (%) [CI]	W/L/T	p
w/o full history (3 msgs)	N-$CLIP$	280	2.603	3.865	$+32.7$ $[22.5, 41.8]^{\dagger}$	177/99/4	3.10×10^{-6}
	PL	280	0.996	1.784	$+44.2$ $[25.7, 56.9]^{\dagger}$	183/97/0	3.08×10^{-7}
w/o role asymmetry	N-$CLIP$	280	2.603	2.935	$+11.3$ $[3.7, 23.5]^{\dagger}$	164/113/3	.0026
	PL	280	0.996	1.233	$+19.2$ $[2.9, 31.7]^{\dagger}$	169/108/3	.00030
w/o visual verifier	N-$CLIP$	280	2.603	2.774	$+6.2$ $[-5.6, 17.0]$	159/118/3	.016
	PL	280	0.996	1.278	$+22.0$ $[-4.2, 40.8]$	161/117/2	.0098

Table 7: **Scene origin matters under complete-program execution.** Both arms retain generator history; only the scene-persistence setting is changed. Means use jointly valid tasks and are scaled by 100. Red.: reduction of rebuild relative to persistent; CI: 95% task bootstrap (5,000 resamples, seed 0). W/L/T: rebuild wins/losses/ties. p: exact two-sided sign test excluding ties; †: CI excludes zero and $p < .05$. Both arms complete 278/280 tasks. Family results are in Appendix H.

Tasks	n	Metric ↓	Rebuild	Persistent	Red. (%) [CI]	W/L/T	p
Overall	276	$N\text{-}CLIP$	2.371	3.585	$+33.9\ [19.8, 45.8]^{\dagger}$	154/120/2	.0460
Overall	276	PL	1.047	1.417	$+26.1\ [-2.4, 43.6]$	147/129/0	.3062
Geometry	52	$N\text{-}CLIP$	4.821	10.341	$+53.4\ [34.7, 67.2]^{\dagger}$	35/16/1	.0110
Geometry	52	PL	3.111	5.039	$+38.3\ [-3.4, 61.1]$	38/14/0	.0012

Table 8: **Scene-origin ablation by task family.** Same definitions and joint criterion as Table 7. Both means use the paired task set; means are $\times 100$. Intervals bootstrap the relative reduction of mean error.

Family	n	Metric ↓	Rebuild	Persistent	Red. (%) [CI]	W/L/T	p
Blend-shape	84	$N\text{-}CLIP$	0.934	0.883	$(5.8)\ [-49.2, 24.5]$	43/40/1	.8264
Blend-shape	84	PL	0.301	0.313	$+4.0\ [-26.3, 26.9]$	42/42/0	1.0000
Placement	50	$N\text{-}CLIP$	1.487	1.490	$+0.2\ [-25.7, 21.3]$	26/24/0	.8877
Placement	50	PL	0.603	0.580	$(3.9)\ [-19.6, 12.0]$	22/28/0	.4799
Geometry	52	$N\text{-}CLIP$	4.821	10.341	$+53.4\ [34.7, 67.2]^{\dagger}$	35/16/1	.0110
Geometry	52	PL	3.111	5.039	$+38.3\ [-3.4, 61.1]$	38/14/0	.0012
Lighting	45	$N\text{-}CLIP$	0.773	0.765	$(1.1)\ [-56.1, 36.1]$	25/20/0	.5515
Lighting	45	PL	0.414	0.399	$(3.8)\ [-64.9, 32.9]$	22/23/0	1.0000
Material	45	$N\text{-}CLIP$	4.800	5.966	$+19.5\ [0.8, 36.0]$	25/20/0	.5515
Material	45	PL	1.180	1.241	$+4.9\ [-28.5, 24.7]$	23/22/0	1.0000

Role asymmetry. Sharing one memory between the generator and verifier raises both errors. The default wins on 164 and 169 of the 280 tasks, and both metrics meet the paired criterion.

Visual verification. Removing the verifier raises both mean errors, and the default wins on more tasks (159 and 161; $p = .016$ and .0098).

H SCENE-ORIGIN DETAILS AND ADDITIONAL TRAJECTORIES

Table 7 and Figure 5 give the aggregate origin contrast and its recorded trajectories. Both arms retain history; only the scene supplied to complete-program execution changes. This independent GPT-5.6-luna run set uses all 280 tasks, the full generator context, one candidate per interaction, and a ten-interaction cap. Both arms request complete scripts and use the same final-answer rule. The persistent-scene switch copies the initial `.blend` to a job-local path and uses that path for both loading and saving; the rebuild arm loads the initial task file without saving over it. Successful scene changes persist only in the first arm. The execution script saves the scene after execution and rendering. The two arms are separate sampled trajectories with the same inference settings.

Rebuild has no valid answer on `blendshape40` and `geometry53`; persistent has none on `geometry32` and `geometry37`. The intersection therefore contains 84 blend-shape, 50 placement, 52 geometry, 45 lighting and 45 material tasks. Both arms' stored final renders were scored together with the same evaluator, without agent calls or scene rerendering. The bootstrap follows the main paired implementation: sorted task IDs, 5,000 samples with replacement using seed 0, and percentile endpoints of the relative mean reduction; exact ties are excluded from the two-sided sign test. On the persistent-relative scale, the overall and geometry $N\text{-}CLIP$ reductions are 33.9% and 53.4%; on the rebuild-relative scale, persistent increases error by 51.2% and 114.5%, respectively.

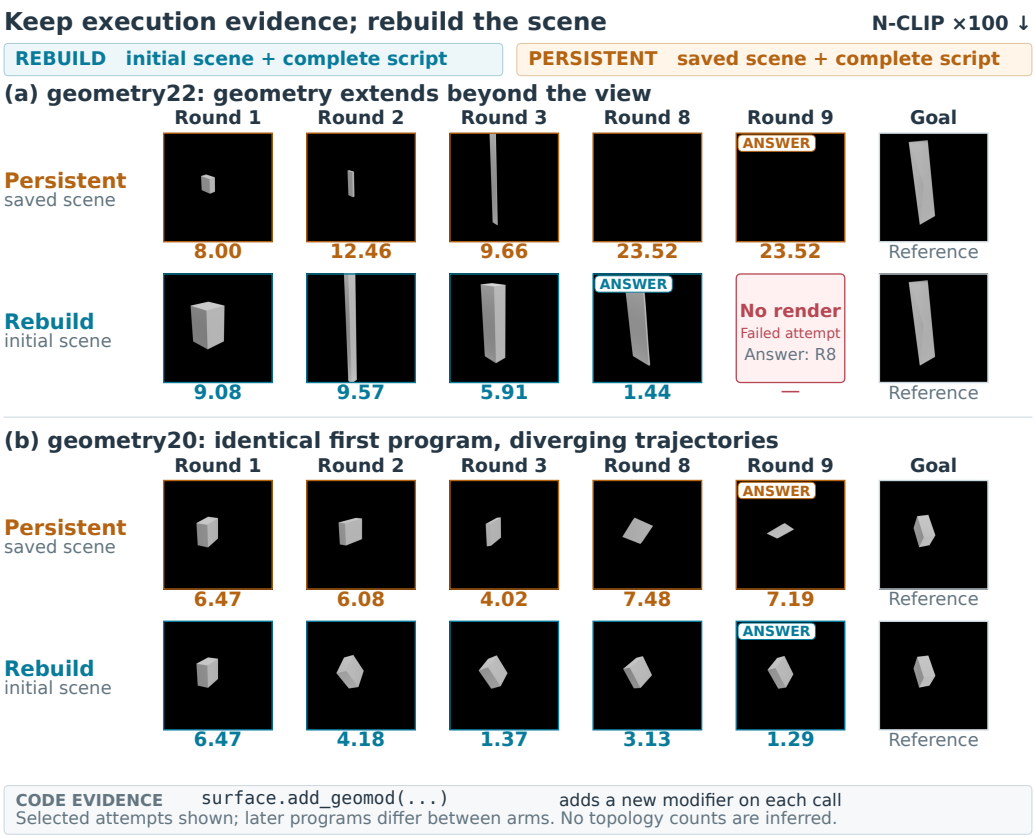

Figure 5: **A complete program can compound edits left in a persistent scene.** Selected execution attempts from the scene-origin ablation; both arms retain the full generator history. (a) On `geometry22`, the persistent trajectory elongates the object until it leaves the visible frame, while rebuild retains a target-like object. Rebuild's round 9 has no render; its scored answer is round 8 ($N\text{-}CLIP \times 100 = 1.44$), versus persistent's round 9 (23.52). (b) On `geometry20`, the first programs are byte-identical and produce equal scores; later persistent renders become progressively flatter, while rebuild finishes closer to the goal (1.29 vs. 7.19). The shared `surface.add_geomod` helper creates a new modifier on each call. Appendix P separately executes the identical first program repeatedly and measures modifier counts, mesh volume, and object bounds.

Reading the trajectory figure. Figure 5 displays selected execution attempts 1, 2, 3, 8 and 9; the ten-interaction budget also includes the initial planning action. For `geometry22`, rebuild's ninth script fails to produce a render, so its answer is the eighth render. For `geometry20`, the first submitted programs have identical SHA-256 hashes. The helper's modifier allocation is verified in the implementation.

I Candidate breadth

Table 9 compares the two candidate widths in the current GPT cohort. Completion is 278/280 at $N = 1$ and 280/280 at $N = 4$. On the 278 shared tasks, mean $N\text{-}CLIP$ decreases by 7.7% (CI $[-4.6, 18.4]$, $p = .278$) and PL by 9.4% (CI $[-14.7, 26.6]$, $p = .436$). The $N = 4$ tournament selects a candidate other than candidate zero in 497 of 2,453 interactions (20.3%). The observed costs are $3.6\times$ the model calls, $4.0\times$ the renders, and $3.6\times$ the agent time of $N = 1$. The controlled history result uses the default $N = 1$ setting.

Table 9: **Candidate breadth in the GPT Blender 3.6 cohort.** Means use each variant's valid tasks; the paired row uses their intersection. Costs are in Table 5. The saved candidate logs contain 2,453 interactions for $N = 4$, with 497 nonzero winners; selected indices 0/1/2/3 occur 1,955/222/194/81 times.

Variant	Valid	$N\text{-}CLIP \downarrow$	$PL \downarrow$
VETR $N = 1$	278	2.616	1.149
VETR $N = 4$	280	2.397	1.034
Paired red. ($n = 278$)		$+7.7\ [-4.6, 18.4]$	$+9.4\ [-14.7, 26.6]$

Table 10: **Matched-task GPT image quality.** Each system is paired with One-Shot on their valid-task intersection; the One-Shot reference is recomputed for that exact set. The benchmark errors are scaled by 100. Paired effects and tests are in Tables 11 and 14.

System (paired subset)	n	$N\text{-}CLIP \downarrow$	$PL \downarrow$	PSNR↑	SSIM↑	Edge F1↑	LPIPS↓	CLIP↑
One-Shot (N1 pairs)	212	2.607	1.402	24.67	0.883	0.771	0.129	0.974
VETR $N = 1$	212	1.555	0.613	29.15	0.923	0.833	0.076	0.984
One-Shot (N4 pairs)	214	2.583	1.389	25.17	0.884	0.773	0.127	0.974
VETR $N = 4$	214	1.220	0.573	30.85	0.924	0.840	0.069	0.988
One-Shot (VIGA pairs)	214	2.583	1.389	25.17	0.884	0.773	0.127	0.974
VIGA	214	2.198	0.877	28.18	0.911	0.812	0.087	0.978
One-Shot (Alchemy pairs)	212	2.570	1.296	25.23	0.886	0.780	0.125	0.974
BlenderAlchemy	212	7.167	4.072	19.00	0.787	0.614	0.266	0.928

J Detailed paired comparisons

Table 11 reports current GPT system comparisons and all ten Qwen method pairs. Each comparison pairs valid task identities within its backbone. Table 10 shows the GPT quality means for each One-Shot intersection.

Paired statistics for the execution-refinement ablation. Table 12 pairs the default VETR run with the refinement-disabled run on the 27 released BlenderBench tasks for GPT-5.6-luna, using the archived per-task ledger; both arms complete all 27 tasks. Reductions compare mean errors across tasks, and intervals, wins, and sign tests follow the procedure above. Refinement lowers mean PL by 41.8% (24/3 wins, $p = 4.9 \times 10^{-5}$), meeting the paired criterion, and lowers mean $N\text{-}CLIP$ by 32.2%, with a bootstrap interval of $[14.4, 55.5]$ that excludes zero.

Paired statistics for the BlenderBench system comparison. Table 13 pairs VETR with VIGA on the 27 released BlenderBench tasks for GPT-5.6-luna, using the same archived per-task ledger. Both systems complete all 27 tasks, so the means equal the averages of the level means in Table 3. VETR lowers mean $N\text{-}CLIP$ by 55.0% and mean PL by 67.7%, and both bootstrap intervals exclude zero.

K Qwen3.8-27B evaluation: settings and scope

Backbone and runtime. The second-backbone runs use a locally served Qwen endpoint, denoted Qwen3.8-27B, on all 280 tasks. The run environment uses four H100 80 GB GPUs, Blender 3.6.0, and BlenderGym's Infinigen fork, Cycles CUDA rendering with 512 samples, and the same CLIP ViT-B/32 and RGB-MSE evaluator. One-Shot, VIGA, and VETR use enabled thinking; VIGA and VETR have a ten-round budget.

Budgets and failures. VIGA and VETR $N{=}1$ have 7,200 s timeouts; $N{=}4$ and BlenderAlchemy have 21,600 s. A timeout remains a failure even if an intermediate render exists. One-Shot has 29 no-render failures; VIGA has three timeouts and 12 no-render failures; $N{=}1$ has four timeouts; $N{=}4$

Table 11: **Detailed paired comparisons by backbone.** N1/N4: VETR $N{=}1/4$; Alchemy: BlenderAlchemy (2×2 for Qwen). n: tasks valid for both systems; Red.: relative reduction of the first system's mean error against the second (positive favors the first); CI: 95% task-bootstrap interval (5,000 resamples, seed 0); Wins: strictly lower error; p: exact two-sided sign test, excluding ties. †: CI excludes zero and $p < .05$ (unadjusted). Qwen includes all ten method pairs; BlenderAlchemy uses depth 2, breadth 2, one try, and disabled thinking (Appendix K).

		N-CLIP			*PL*		
Comparison	n	Red. (%) [95% CI]	Wins	p	Red. (%) [95% CI]	Wins	p
GPT-5.6-luna							
N1 vs. VIGA	277	$+36.3\ [24.3, 46.4]^{\dagger}$	161	.0054	$+16.7\ [-9.1, 36.6]$	149	.2294
N4 vs. VIGA	279	$+40.6\ [30.2, 49.3]$	149	.1845	$+24.6\ [5.6, 40.0]$	152	.1506
N1 vs. One-Shot	212	$+40.3\ [29.4, 50.5]^{\dagger}$	159	8.41×10^{-14}	$+56.3\ [44.6, 66.7]^{\dagger}$	144	7.70×10^{-8}
N4 vs. One-Shot	214	$+52.7\ [44.9, 60.3]^{\dagger}$	168	1.19×10^{-18}	$+58.8\ [45.9, 68.8]^{\dagger}$	156	2.19×10^{-12}
N4 vs. N1	278	$+7.7\ [-4.6, 18.4]$	147	.2777	$+9.4\ [-14.7, 26.6]$	146	.4356
VIGA vs. One-Shot	214	$+14.9\ [-6.9, 34.1]$	153	4.58×10^{-11}	$+36.8\ [18.3, 51.5]^{\dagger}$	147	4.69×10^{-8}
N1 vs. Alchemy	260	$+72.5\ [67.1, 77.3]^{\dagger}$	234	5.08×10^{-43}	$+83.3\ [77.8, 87.5]^{\dagger}$	222	8.49×10^{-33}
N4 vs. Alchemy	262	$+75.6\ [71.0, 79.7]^{\dagger}$	241	1.54×10^{-48}	$+83.8\ [79.5, 87.4]^{\dagger}$	227	1.25×10^{-35}
VIGA vs. Alchemy	261	$+59.7\ [51.3, 67.6]^{\dagger}$	228	4.92×10^{-37}	$+76.9\ [69.8, 82.4]^{\dagger}$	217	1.17×10^{-28}
Alchemy vs. One-Shot	212	$(178.9)\ [-238.3, -133.0]^{\dagger}$	39	2.45×10^{-21}	$(214.1)\ [-314.4, -141.4]^{\dagger}$	56	4.18×10^{-12}
Qwen3.8-27B							
N4 vs. N1	275	$+11.7\ [-1.8, 23.8]$	148	.2277	$+11.5\ [-6.7, 28.5]$	141	.7176
N4 vs. VIGA	264	$+40.0\ [30.2, 49.0]^{\dagger}$	171	1.82×10^{-6}	$+56.3\ [42.4, 66.9]^{\dagger}$	169	6.16×10^{-6}
N4 vs. One-Shot	250	$+28.1\ [16.0, 39.0]^{\dagger}$	148	.0043	$+49.6\ [31.8, 62.6]^{\dagger}$	149	.0023
N4 vs. Alchemy	278	$+29.5\ [18.2, 40.0]^{\dagger}$	186	1.80×10^{-8}	$+44.5\ [30.6, 56.1]^{\dagger}$	174	2.36×10^{-5}
N1 vs. VIGA	261	$+31.9\ [20.1, 42.4]^{\dagger}$	162	1.16×10^{-4}	$+50.8\ [34.3, 63.0]^{\dagger}$	152	.0092
N1 vs. One-Shot	248	$+22.1\ [9.4, 32.7]$	139	.0653	$+43.6\ [21.2, 59.4]$	137	.0979
N1 vs. Alchemy	275	$+17.5\ [4.6, 28.6]^{\dagger}$	170	1.06×10^{-4}	$+37.2\ [18.5, 50.7]^{\dagger}$	163	.0020
Alchemy vs. VIGA	264	$+19.2\ [7.5, 29.8]$	133	.9509	$+19.9\ [-8.2, 39.0]$	129	.7584
Alchemy vs. One-Shot	250	$(0.7)\ [-15.7, 12.3]$	105	.0186	$+8.0\ [-28.7, 35.6]$	119	.6108
VIGA vs. One-Shot	238	$(15.0)\ [-35.6, 2.1]$	109	.2180	$(14.2)\ [-63.2, 22.5]$	111	.3309

Table 12: **Paired effect of execution refinement on BlenderBench (GPT-5.6-luna, 27 tasks).** Means are scaled by 100. Red.: relative reduction of the default mean error against the refinement-disabled run (positive favors refinement); CI: 95% task bootstrap (5,000 resamples, seed 0); W/L: per-task wins/losses of the default (no ties); p: exact two-sided sign test; †: CI excludes zero and $p < .05$.

Metric	w/o refinement	VETR	Red. (%) [CI]	W/L	p
N-CLIP	9.453	6.406	$+32.2\ [14.4, 55.5]$	18/9	.122
PL	7.099	4.132	$+41.8\ [6.7, 49.0]^{\dagger}$	24/3	4.9×10^{-5}

Table 13: **Paired comparison with VIGA on BlenderBench (GPT-5.6-luna, 27 tasks).** Means are scaled by 100. Red.: relative reduction of the VETR mean error against VIGA; CI: 95% task bootstrap (5,000 resamples, seed 0); W/L: per-task wins/losses of VETR (no ties); p: exact two-sided sign test; †: CI excludes zero and $p < .05$.

Metric	VIGA	VETR	Red. (%) [CI]	W/L	p
N-CLIP	14.233	6.406	$+55.0\ [29.3, 69.8]$	19/8	.052
PL	12.781	4.132	$+67.7\ [37.7, 74.1]$	15/12	.701

has one (`geometry37`). BlenderAlchemy's only failure (`material5`) produced no candidate render after an upstream planning error.

BlenderAlchemy configuration. The Qwen line uses the released implementation with family-specific edit styles and refinement strategies (depth 2, breadth 2, one try). Visual imagination is disabled because the target render is supplied, and thinking is disabled to obtain parseable planning responses.

Statistical scope and traceability. All means use successful renders from normally completed tasks. Paired comparisons intersect valid task IDs, bootstrap tasks 5,000 times (seed 0), and apply an exact two-sided sign test after removing ties. Tests are reported without a multiple-comparison correction, as in the primary evaluation. The export contains 1,400 task–method records in `per_task_metrics.csv` and full-precision merged records in `raw/qwen38_all__metrics_committed.json`, under `supplement/blendergym_tables_export/`. Table 1's Qwen block and Table 11's Qwen block have their tabulated values independently recomputed and numerically cross-checked by `supplement/update_results.py`.

L EXTENDED IMAGE METRICS

Definitions. The extended scorer reads the same stored final-answer images used for the primary metrics, matches render/goal filenames, and averages per-image values over available views and then over valid tasks. Images are loaded as RGB in $[0, 1]$ at goal resolution; a size mismatch is resized to that resolution. *PSNR* is $-10 \log_{10} \mathrm{MSE}$ in dB, with 100 dB used when $\mathrm{MSE} \leq 10^{-10}$. It is computed per image before averaging. *SSIM* uses `skimage.metrics.structural_similarity`, RGB channels, `data_range=1`, and the default uniform 7×7 window. *LPIPS* uses the AlexNet backbone with the package's default learned calibration and inputs scaled to $[-1, 1]$. *Edge F1* applies OpenCV Canny to both 8-bit grayscale images with thresholds $(100, 200)$. Each edge map is dilated by a 5×5 all-ones kernel: a match is within two pixels in Chebyshev distance. Precision is the fraction of predicted edges matched to a goal edge, and recall the converse; their harmonic mean is reported. Both maps empty gives 1; exactly one empty gives 0. *CLIP* is cosine similarity from ViT-B/32 image embeddings, equal to $1 - N\text{-}CLIP$ before the latter's $\times 100$ table scaling. All CLIP tables derive it from the same canonical per-task $N\text{-}CLIP$ score, so their favorable orderings and ties agree exactly; separately recomputed cosine scores are retained in the scoring exports. For a higher-is-better metric, paired relative improvement is $(\bar{m}_{\text{first}} - \bar{m}_{\text{second}})/\bar{m}_{\text{second}}$; for LPIPS its sign is reversed. Reported percentages multiply these ratios by 100. Task-bootstrap CIs, wins, and sign tests follow Appendix J.

One-Shot means. One-Shot extended metrics cover all 214 of its valid GPT tasks. Its family composition differs from that of iterative systems because failures concentrate in geometry and material. GPT comparisons use 212 jointly valid tasks for $N = 1$ and 214 for $N = 4$, with each reference recomputed on the same set (Table 10).

M ONE-SHOT EDITING STRATEGY: STATISTICS BEHIND SECTION F

Measure. For this program-form analysis, each source line is truncated at its first `#`, whitespace is normalized, and empty lines are removed; thus inline comments are stripped. *Retention* counts initial-line occurrences whose normalized text appears anywhere in the generated program, divided by the initial line count. *Changed lines* counts initial-line occurrences absent from the generated program plus generated-line occurrences absent from the initial program. Identical means equality of the normalized line sequences. A program is *scene-clearing* if the source contains `read_factory_settings`, `bpy.ops.object.delete`, `objects.remove(`, or `nodes.clear()`. A program *contains a 4.x binding* if it names a Blender identifier introduced in 4.0–4.2 and absent in 3.6 (`BLENDER_EEVEE_NEXT`, `surface_render_method`, the Principled-BSDF sockets `Specular IOR Level`, `Subsurface Weight`, `Subsurface Scale`, `Transmission Weight`, `Coat Weight`, `Coat Roughness`, `Sheen Weight`, `Emission Color`, and the Noise-Texture socket `Lacunarity`). The 100 geometry and material tasks are used because they are the tasks whose archived GPT One-Shot programs were produced and executed under Blender 3.6.

Reading. The GPT One-Shot failures are concentrated in programs that discarded the start program: of the 92 programs retaining less than half of it, 65 fail, against 1 of the 8 that kept at least half. With VETR's prompt and loop, GPT produces first-round programs on all 100 tasks, with median retention 0.96 and 98 rendered outcomes; the full run has 41 logged no-render outcomes in 814 interactions on these tasks.

Table 14: **Paired comparisons on the extended metrics.** N1/N4 denote VETR at $N = 1/4$. n: paired sample size; Δ: PSNR difference in dB and relative improvement (%) for the other metrics, in the first system's favorable direction; CI: 95% task bootstrap (5,000 resamples, seed 0); W/L: per-task wins/losses (ties excluded); p: exact two-sided sign test (unadjusted); †: recorded CI excludes zero and $p < .05$.

GPT-5.6-luna					**Qwen3.8-27B**				
Metric	n	Δ [CI]	W/L	p	Metric	n	Δ [CI]	W/L	p
N1 vs. VIGA					*N1 vs. VIGA*				
PSNR	277	$+1.38\ [0.13, 2.64]$	149/128	.2294	PSNR	261	$+1.51\ [0.35, 2.75]^{\dagger}$	152/109	.0092
SSIM	277	$+2.7\ [1.1, 4.4]$	149/125	.1646	SSIM	261	$+4.1\ [2.0, 6.3]$	146/114	.0543
Edge F1	277	$+5.1\ [1.9, 8.6]$	135/110	.1250	Edge F1	261	$+4.5\ [2.0, 6.9]$	141/120	.2157
LPIPS	277	$+14.1\ [5.7, 22.0]$	150/127	.1861	LPIPS	261	$+24.0\ [16.3, 31.3]^{\dagger}$	155/106	.0029
CLIP	277	$+1.5\ [0.9, 2.3]^{\dagger}$	161/114	.0054	CLIP	261	$+1.6\ [1.2, 2.0]^{\dagger}$	162/99	.0001
N4 vs. VIGA					*N4 vs. VIGA*				
PSNR	279	$+2.24\ [1.01, 3.54]$	152/127	.1506	PSNR	264	$+2.48\ [1.34, 3.72]^{\dagger}$	169/95	6.16×10^{-6}
SSIM	279	$+2.6\ [0.9, 4.3]$	149/127	.2061	SSIM	264	$+6.3\ [4.1, 8.8]^{\dagger}$	161/102	.0003
Edge F1	279	$+5.6\ [1.9, 9.6]^{\dagger}$	144/97	.0030	Edge F1	264	$+6.9\ [4.7, 9.2]^{\dagger}$	164/100	9.82×10^{-5}
LPIPS	279	$+18.5\ [10.6, 26.3]^{\dagger}$	159/120	.0227	LPIPS	264	$+28.6\ [21.9, 35.1]^{\dagger}$	173/91	5.06×10^{-7}
CLIP	279	$+1.7\ [1.1, 2.4]$	149/126	.1845	CLIP	264	$+2.0\ [1.5, 2.4]^{\dagger}$	171/93	1.82×10^{-6}
N1 vs. One-Shot					*N1 vs. One-Shot*				
PSNR	212	$+4.48\ [2.87, 6.20]^{\dagger}$	144/66	7.70×10^{-8}	PSNR	248	$+0.17\ [-1.28, 1.56]$	137/110	.0979
SSIM	212	$+4.6\ [3.2, 6.3]^{\dagger}$	150/59	2.51×10^{-10}	SSIM	248	$+0.3\ [-1.6, 2.2]$	129/117	.4832
Edge F1	212	$+8.1\ [5.4, 11.2]^{\dagger}$	147/47	3.45×10^{-13}	Edge F1	248	$+0.9\ [-1.3, 3.1]$	132/116	.3409
LPIPS	212	$+40.9\ [32.5, 49.1]^{\dagger}$	162/50	5.52×10^{-15}	LPIPS	248	$+15.8\ [6.8, 24.0]^{\dagger}$	144/104	.0131
CLIP	212	$+1.1\ [0.7, 1.4]^{\dagger}$	159/52	8.41×10^{-14}	CLIP	248	$+0.8\ [0.3, 1.3]$	139/109	.0653
N4 vs. One-Shot					*N4 vs. One-Shot*				
PSNR	214	$+5.68\ [3.94, 7.59]^{\dagger}$	156/55	2.19×10^{-12}	PSNR	250	$+1.37\ [-0.06, 2.79]$	149/100	.0023
SSIM	214	$+4.5\ [3.0, 6.2]^{\dagger}$	157/52	1.92×10^{-13}	SSIM	250	$+2.0\ [0.2, 3.9]^{\dagger}$	146/102	.0062
Edge F1	214	$+8.7\ [5.5, 12.0]^{\dagger}$	149/44	1.52×10^{-14}	Edge F1	250	$+2.6\ [0.4, 4.8]$	140/110	.0664
LPIPS	214	$+45.7\ [38.5, 52.7]^{\dagger}$	168/46	1.61×10^{-17}	LPIPS	250	$+19.8\ [11.8, 27.5]^{\dagger}$	150/100	.0019
CLIP	214	$+1.4\ [1.1, 1.8]^{\dagger}$	168/43	1.19×10^{-18}	CLIP	250	$+1.0\ [0.6, 1.4]^{\dagger}$	148/102	.0043
N4 vs. N1					*N4 vs. N1*				
PSNR	278	$+0.88\ [-0.53, 2.28]$	146/132	.4356	PSNR	275	$+1.17\ [-0.19, 2.60]$	141/134	.7176
SSIM	278	$-0.1\ [-1.4, 1.2]$	140/134	.7627	SSIM	275	$+1.9\ [0.2, 3.9]$	146/129	.3346
Edge F1	278	$+0.6\ [-2.6, 3.6]$	125/116	.6064	Edge F1	275	$+3.1\ [0.8, 5.3]$	148/127	.2277
LPIPS	278	$+5.1\ [-3.1, 12.6]$	138/140	.9522	LPIPS	275	$+6.9\ [-0.8, 14.7]$	144/131	.4694
CLIP	278	$+0.2\ [-0.1, 0.5]$	147/128	.2777	CLIP	275	$+0.4\ [0.0, 0.8]$	148/127	.2277
VIGA vs. One-Shot					*VIGA vs. One-Shot*				
PSNR	214	$+3.02\ [1.91, 4.21]^{\dagger}$	147/67	4.69×10^{-8}	PSNR	238	$-0.90\ [-2.28, 0.33]$	111/127	.3309
SSIM	214	$+3.1\ [1.8, 4.5]^{\dagger}$	145/65	3.45×10^{-8}	SSIM	238	$-3.2\ [-5.4, -1.1]$	105/133	.0799
Edge F1	214	$+5.1\ [2.8, 7.5]^{\dagger}$	135/57	1.76×10^{-8}	Edge F1	238	$-3.2\ [-5.1, -1.4]$	110/128	.2704
LPIPS	214	$+32.0\ [23.6, 40.4]^{\dagger}$	152/62	6.61×10^{-10}	LPIPS	238	$-9.9\ [-22.0, 0.6]$	114/124	.5597
CLIP	214	$+0.4\ [-0.2, 0.9]$	153/58	4.58×10^{-11}	CLIP	238	$-0.5\ [-0.8, -0.2]$	109/129	.2180

Table 15: **One-Shot program form.** GPT-5.6-luna on the 100 geometry and material tasks under Blender 3.6, and the first-round program of the same backbone inside VETR $N{=}1$ on the same tasks. Valid: tasks with a render for One-Shot; rendered programs / tasks run for round 1. First-error counts name the exception that ended the failed program.

	One-Shot	VETR $N{=}1$, round 1
Valid	34	98 / 100
Median retention of start lines	0.10	0.96
Programs retaining $\geq 50\%$ (valid)	8 (7)	99
Programs retaining $< 50\%$ (valid)	92 (27)	1
Programs identical to the start program	0	–
Median changed lines	188	–
Scene-clearing programs	46	–
Programs containing a 4.x binding	65	4
`Specular IOR Level` / `Lacunarity` / `BLENDER_EEVEE_NEXT`	44 / 13 / 12	–
First error = 4.x binding (same three)	21 / 10 / 9	–
First error = other socket, type or syntax error	18	–
Failed with no Python exception in the log (crash / kill)	8	–
No-render generator rounds, all rounds of the 100 tasks	–	41 / 814

Table 16: **Experiment map.** Each row identifies a separate source cohort. Iterative VIGA/VETR runs allow ten generator interactions; N is the number of candidate actions per interaction. Within each ablation, the backbone, Blender version, tools, prompts, and budget are fixed except for the named switch. Completion and paired task intersections are reported with the corresponding results.

Evidence / source	Tasks	Blender	Configuration and comparison
GPT systems (Table 1)	280 each	3.6	GPT-5.6-luna; One-Shot, BlenderAlchemy, VIGA, VETR $N = 1/4$. Each system uses its own instructions and runner; BlenderAlchemy uses four candidates per interaction.
Context ablations (Table 6)	280 each	3.6.0	GPT, $N = 1$; one default run and three variants (three-message generator history, shared generator/verifier memory, no visual verifier); quality scored after sampling finished.
Scene origin (Table 7)	280 each	3.6	GPT, $N = 1$; fixed base vs. saved scene; both retain 24 messages and use identical complete-program instructions.
Guidance (Table 18)	280 / 85	3.6	GPT, $N = 1$, fixed origin, 24 messages. VIGA role-prompt substitution on 280 tasks; recipe removal on 85 blend-shape tasks. Full is the scene-origin arm above.
Qwen systems (Appendix K)	280 each	3.6	Qwen3.8-27B; all five systems. VIGA/VETR use thinking; BlenderAlchemy uses depth 2, breadth 2, one try, with thinking disabled. Timeout caps differ by system.

N Execution settings

Runtime configuration. All GPT and Qwen runs use Blender 3.6.0 with BlenderGym's Infinigen fork. BlenderGym's unmodified initial programs execute on all 280 tasks in the benchmark-compatible environment.

Result provenance. Each task/system has one designated source result. The GPT system table uses one complete Blender 3.6 run set, and the context ablations use one separately identified run set; failures in every source are retained. The GPT BlenderAlchemy row uses the same Blender 3.6 environment as the other current GPT systems, with four candidates per interaction. Per-task sources and scripts are enumerated in Appendix Q.

Answer extraction. All iterative systems are scored using their last successful selected render from a completed run. A later execution failure contributes feedback and does not replace that answer. Tasks without a successful render, and timed-out or incomplete runs, are invalid. Stored renders are rescored by the same evaluator. Primary comparisons use the intersection of valid task identities; separate completion counts retain the information about missing answers.

O Within-run progress and guidance controls

First executed program versus the final answer. We evaluate the first executed script in each Full trajectory from the scene-origin cohort. This proposal may follow an initial planning action but precedes feedback from any executed program in that run. It uses the same task, role instructions, backbone, and fixed-origin backend as the subsequent refinements. We score stored first-attempt renders irrespective of eventual completion: 258/280 first proposals render, 278/280 final runs are valid, and 256 tasks have both. No metric-based historical selection is applied.

Role-instruction sensitivity. Two existing controls retain the Full runner, full generator history, $N = 1$, and fixed execution origin. The first replaces the generator and verifier role-system prompts with the released VIGA role prompts (277/280 valid; 276 paired with Full). The second removes the body-skin recipe and worked example on the 85 blend-shape tasks (84 valid; 83 paired). Full completes 278/280 tasks overall. Table 18 reports both primary metrics for both controls; neither

Table 17: **Refinement of the same initial proposals.** All 256 jointly available tasks; means are scaled by 100. Reduction compares final versus first executed program. Bootstrap, sign tests, and † use the same definitions as the main experiments. W/L/T favors the final answer.

Metric	First program	Final answer	Reduction (%) [CI]	W/L/T	p
$N\text{-}CLIP$	3.922	2.255	$+42.5$ $[33.7, 50.6]^{\dagger}$	190/50/16	2.11×10^{-20}
PL	1.710	0.829	$+51.5$ $[38.4, 61.7]^{\dagger}$	188/52/16	2.91×10^{-19}

Table 18: **Guidance controls under the same refinement configuration.** Positive reductions favor Full. Complete paired intervals and exact sign tests are shown, with ties excluded from the latter. The no-recipe control is restricted to blend-shape tasks.

Control	n	Metric	Full reduction (%) [CI]	W/L/T	p
VIGA role prompts	276	$N\text{-}CLIP$	$+9.4$ $[-5.4, 21.9]$	148/128/0	.2527
	276	PL	(6.7) $[-36.2, 14.8]$	145/130/1	.3986
Without body-skin recipe	83	$N\text{-}CLIP$	(0.7) $[-24.2, 19.4]$	42/39/2	.8243
	83	PL	$+3.6$ $[-20.4, 24.7]$	43/40/0	.8264

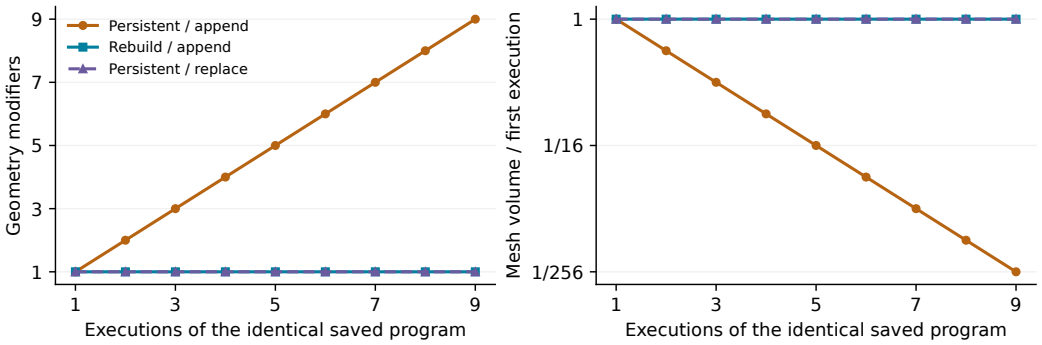

Figure 6: **The same program has different repeated-execution semantics.** Measured modifier count and evaluated mesh volume on `geometry20`. Persistent appending creates one additional modifier per execution and halves the current volume each time. Rebuild and explicit modifier replacement coincide at one modifier and constant volume. All three use the same archived program and Blender 3.6.0; the volume is normalized by each policy's identical first execution.

control produces a significant paired difference. Source hashes and task-level configuration records accompany the measurements.

P IDENTICAL-PROGRAM REPLAY OF SCENE OPERATIONS

We replay the archived first program of `geometry20`, which is byte-identical in the two closed-loop trajectories of Figure 5(b). Its node transform sets scale $(0.5, 1, 1)$ and a z-rotation of 0.35 radians, then calls `surface.add_geomod`. The original base scene and script are held fixed. We execute it nine times under three policies: reload the base before each execution; retain the edited scene; or retain the scene but remove modifiers absent from the initial base before executing the unchanged script. This last policy supplies a compatible replacement contract. No language-model requests or rendering calls are made.

Persistent execution yields one, three, and nine modifiers at steps 1, 3, and 9; the corresponding measured volumes are approximately 4, 1, and 0.015625 Blender units cubed. Rebuild and replacement both retain one modifier and volume 4 throughout. The evaluated meshes retain eight vertices and six faces under all policies: the effect here is repeated transformation, not growing mesh topology. Object bounds, modifier identities, script/base/helper hashes, and all 27 observations are supplied. This single-program intervention explains a concrete accumulation mechanism and identifies a com-

patible persistent alternative; the 280-task closed-loop experiment separately measures the overall backend policy.

Q ARTIFACTS AND TRACEABILITY

Table 16 identifies the experiment associated with each result. The following records and scripts connect those results to their task-level sources.

- **GPT system metrics.** `supplement/gpt_single_runtime/metrics.json` contains all 1,400 task–system records in the Blender 3.6 comparison. The same directory contains per-task extended metrics, complete paired analyses, and saved cost counters. The 212-task intersection used for the $N{=}1$ comparisons is listed in `analysis.json`. The context-ablation and scene-origin experiments use their separately identified run sets.
- **Paired tests.** `paired_stats.py` produces task bootstraps and exact sign tests. Cost counters are in `run_stats.json`.
- **Scene-origin ablation.** `supplement/scene_execution/metrics.json` contains all 560 task–arm records, including invalid tasks and hashes of scored renders. `paired_stats.py` in the same directory produces `paired_stats.json`, `paired_tasks.csv`, and both scene-ablation tables. The source run is `commit_gate_grid`, with arms `comp_full` and `comp_persistent_scene`; `source_audit/` preserves the runner, evaluator, task manifest, status records and representative logs. Figure 5 is built by `fig/scene_execution/build_figure.py` from stored images and candidate CSVs; its manifest records source hashes.
- **Runtime audit.** The runtime validation records use `env_check_start_scripts.py` outputs (`results.json` and renders) and `env_fix_report.py`.
- **Extended metrics.** `scripts/extended_metrics.py` scores stored final-answer images; per-task and paired records are under `extended_metrics/`. Qwen sources are listed in Appendix K.
- **Program-strategy examples.** Figure 4 and Table 15 use One-Shot programs under `<run>/sampled_data/<task>/baseline/<model>.py`, iterative scripts under `<line>/<task>/scripts/<k>.py`, `generator_candidates.json`, and the corresponding One-Shot logs.

Base scene and proposal contract. The task supplies both `blender_file.blend` (B) and `start.py` (p_0). The latter is the editable reference program shown to the generator; the runner does not execute it as an extra preparation step before a proposal. It loads the supplied native scene and executes the submitted script once. Thus, a complete proposal is complete relative to the assets and objects already supplied in B. The answer rule in Appendix N associates the evaluated render with the program in its selected delivered record, which may be a repaired program rather than the submitted script. A cached fallback retains the provenance of its earlier render and does not assign that render to a failed submission.

Pinned planning channel. The BlenderGym planning tool accepts a free-form `detailed_plan` string covering visual analysis, code analysis, and intended edits. Its response is inserted into the second pinned message under `Initial plan:`; a subsequent planning call replaces that text in place. Both history settings use this mechanism.

Implementation-to-claim mapping. `agents/generator.py` samples and executes candidate actions, selects a returned outcome, invokes the verifier when requested, appends the selected interaction, and logs all candidates. `agents/prompt_builder.py` pins the initial two messages and applies the role-specific recent-message window, with tool-boundary handling. The runner sets the generator window to three for the short-history variant, enables shared history for the role-asymmetry control, and bypasses verification for the no-verifier control. The complete-script response field and latest-render guidance are prompt instructions; the final-answer evaluator separately selects the last successful render. The $N{>}1$ path votes over action types before executing candidates of the selected type; Algorithm 2 explicitly places refinement before the selection of delivered outcomes.

Prompt scope and system controls. The released generator prompts combine shared evidence-use instructions, Blender binding/API conventions, and family-specific guidance. All context-window conditions share these instructions. The VIGA runner saves successful scene updates, whereas the VETR runner loads the task base scene for each complete-script execution. History and scene-origin contrasts change individual settings; the same-run prefix analysis holds initial guidance fixed, and role-prompt/recipe controls are reported in Appendix O.