Title

Reflecting Process Expertise in Procedural Material Generation

Authors

Gupta, Kunal

Joshi, Gaurav

Chen, Yen-Ru

et al.

Publication Date

2026

DOI

10.1007/978-3-032-37281-9_19

Reflecting Process Expertise in Procedural Material Generation

Kunal Gupta, Gaurav Joshi, Yen-Ru Chen, Seemandhar Jain,
Ishit Mehta and Manmohan Chandraker

University of California San Diego, La Jolla, CA, USA

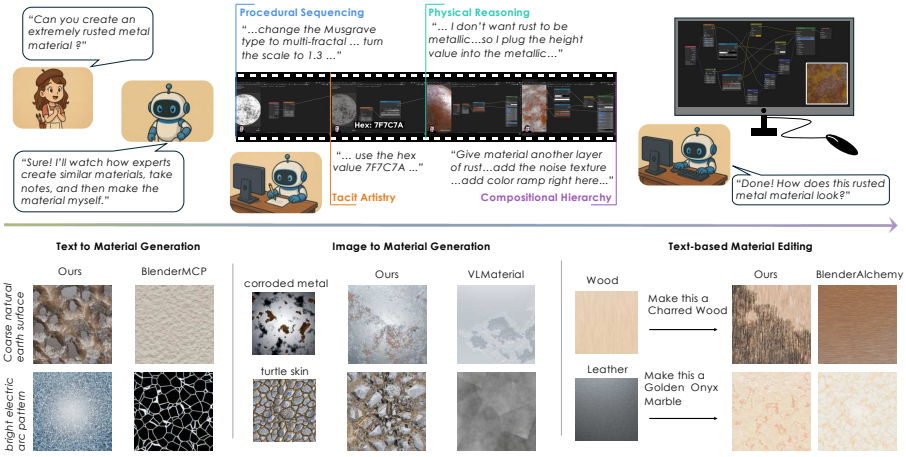


Fig. 1: Retrieval-time process reasoning for procedural materials. (Top) Rather than synthesizing material graphs directly, our approach reasons over *process traces* derived from expert demonstrations (e.g., tutorial videos), capturing procedural sequencing, physical intuition, and compositional design strategies used by artists. (Bottom) This process-centric reasoning enables diverse material creation tasks, including text-to-material generation (left), image-conditioned synthesis (center), and text-based editing (right). Across tasks, materials produced through process reasoning align more closely with user intent and exhibit stronger visual fidelity than prior graph-only methods such as BlenderMCP [2], VLMaterial [16], and BlenderAlchemy [12].

Abstract. Procedural material creation underpins applications in digital content creation, visual effects, and 3D asset design. Achieving high-quality results requires more than reproducing node graphs—it demands understanding the process by which experts construct materials. We formulate procedural material generation as retrieval-time process reasoning over expert demonstrations, elevating process to a first-class representation beyond graph-only synthesis. Concretely, we represent expert workflows as *process traces*: textual records of construction steps, parameters, and design intent. To instantiate this idea, we use a pretrained LLM-based ProcessSynthesizer to synthesize a process trace aligned with a user's intent and a pretrained LLM-based Compiler to ground

the process trace into an executable Blender material graph. Because procedural expertise is most naturally conveyed through demonstrations, we leverage tutorial videos as a source of process knowledge and extract textual, LLM-compatible traces using automated video analysis tools. In an expert study with five Blender artists (avg. 7.5 years of experience), materials generated by reflecting expert demonstrations were found to produce workflows requiring fewer edits, and more closely match professional design strategies than methods operating solely on static artifacts. A user study with 150 participants further shows that our approach achieves superior generation and editing performance compared to prior procedural systems. All code, models, and data will be available at `https://materialapprentice.github.io`.

Keywords: Procedural Generation · Materials · Process Modeling

1 Introduction

Procedural materials power the visual surfaces of films, games, product design, and virtual worlds by encoding appearance as parameterized node graphs rather than fixed bitmaps. Their flexibility makes them essential for scalable, editable, and physically consistent asset creation. Yet crafting these materials remains specialist work: experienced artists rely on visual intuition, procedural reasoning, and domain-specific workflows. Both aspiring and established creators continually refine their craft by observing how experts layer textures, tune parameters, and reason about physical effects. Such demonstrations convey process knowledge that extends far beyond the final graph. Figure 1 illustrates our central premise: material generation should reflect the process expertise artists use when constructing procedural materials. Current methods, even when trained on large procedural material datasets, operate in isolation at inference time: they generate materials but do not incorporate such expertise into the generation.

If process knowledge is central to high-quality material design, how can it be reflected automatically during generation? We address this by framing *procedural material generation as retrieval-time process reasoning*. We introduce MaterialApprentice, a multimodal framework that integrates expert-derived process traces into the generative loop. Rather than operating solely in graph space, MaterialApprentice retrieves relevant traces, synthesizes a target trace aligned with the user's intent, and grounds it into an executable material graph. As illustrated in Fig. 2, MaterialApprentice distinguishes three representations: a *process trace*, which records construction steps, parameters, and design intent; a *material graph*, the executable shader-node layout compiled from the trace; and a *procedural material*, the resulting editable shader asset. Treating process traces as a first-class representation provides a foundation for higher-quality, more editable material generation.

Expert procedural knowledge is typically acquired by observing skilled artists as they construct materials. In practice, this expertise is often externalized through tutorial videos, which provide a temporally ordered and multimodal record of the creation process: the evolving node graph, parameter adjustments, synchronized

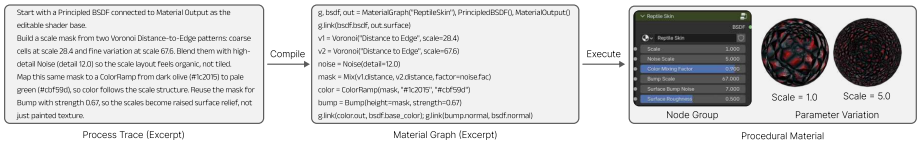


Fig. 2: Three representations used in MaterialApprentice. A *process trace* gives artist-like construction steps with concrete parameters and design intent. The trace is compiled into a *material graph*, an executable shader structure. Executing the graph instantiates a *procedural material* with editable controls and rendered appearances.

narration explaining intent, and visual focus on individual operations. These signals indicate not only the final artifact but the reasoning trajectory behind it.

In our setting, we approximate this observational learning by analyzing Blender tutorial videos and extracting structured, agent-friendly process traces using automated video analysis tools. Rather than forming a training dataset, these traces serve as demonstrations retrieved at inference time. In practice, only a small number of relevant exemplars is sufficient to guide synthesis, forming a lightweight knowledge base for retrieval-time reasoning.

Technically, we instantiate retrieval-time process reasoning through a retrieval-augmented large language model (LLM) framework. Given a user query, MaterialApprentice retrieves relevant process traces and conditions a pretrained LLM-based ProcessSynthesizer to synthesize a process trace aligned with the intended material. This trace is then grounded into an executable representation with a pretrained LLM-based Compiler (differing only in prompting), which translates it into a structured material graph implemented using Blender shader nodes. This design separates reasoning from execution while integrating process-level conditioning directly into the generative loop. Importantly, beyond one-shot synthesis, MaterialApprentice can also *demonstrate the process*, producing step-by-step constructions that mirror expert workflows and support structured editing. In contrast, prior systems such as VLMaterial [16], BlenderAlchemy [12], and Blender-MCP [2] are limited to operation in graph space.

We evaluate MaterialApprentice on a suite of Blender material creation tasks, including multimodal material generation and text-driven editing. Across quantitative metrics and perceptual studies, MaterialApprentice consistently outperforms VLMaterial [16], BlenderAlchemy [12], and Blender-MCP [2]. In addition, a study with experienced Blender artists shows that process traces generated by MaterialApprentice more closely follow expert constructions and require fewer manual edits than alternatives.

In summary, we frame procedural material generation as retrieval-time process reasoning over expert demonstrations. Our contributions are:

- A formulation of procedural material generation as retrieval-time process reasoning, elevating process to a first-class representation beyond graphs.
- A retrieval-augmented architecture with an explicit process-trace representation and compiler that separate reasoning from execution and ground generated traces into executable Blender shader graphs.

- A curated corpus of Blender tutorial videos and video analysis tools for extracting agent-friendly process traces that enable inference-time conditioning.
- Comprehensive evaluation, including with Blender artists, showing process-based synthesis better aligns with expert workflows than graph-based systems.

2 Related Work

Generative Materials Recent work directly synthesizes material appearances. MaterialGAN [9] extends StyleGAN2 [14] to generate SVBRDF textures, but operates in an opaque latent space. Diffusion-based methods improve visual quality: ControlMat [22] uses ControlNet-guided diffusion for SVBRDF generation with limited structural control, DreamMat [27] maps text prompts to point-wise BRDF parameters in a hash grid, and MaterialPicker [17] fine-tunes a video generator for text- and image-conditioned material textures. Huang et al. [13] generate textures directly in UV space, improving surface consistency. However, these methods produce non-procedural, sample-based or latent representations and do not expose editable procedural graphs.

Generative Procedural Materials Closest to our work are methods that synthesize procedural material graphs. Hu et al. [10] generate node-based materials conditioned on text prompts, example images, or partial node graphs, using node graphs as defined in Adobe Substance 3D [1] and the Matformer representation [8]. Li et al. [15] build on these ideas, using synthetic data to train a transformer model for material graph generation and applying RL post-training on both real and synthetic data to improve realism and stability. While these methods produce procedural representations that are more editable and interpretable than purely texture-based outputs, they rely on large-scale training and custom models.

LLMs for Program Synthesis Large language models (LLMs) have demonstrated strong capabilities in program synthesis [4], motivating their use as engines for 3D content creation. Early systems directly generate Python code for Blender, such as 3DGPT [19]. However, operating in a general-purpose, low-level language makes it difficult to maintain structure, guarantee validity, and scale to complex scenes. To address these limitations, several works introduce domain-specific languages (DSLs) tailored for 3D generation. Holodeck [24] and L3GO [23] use LLMs to author scenes in compact DSLs, simplifying high-level planning and constraint reasoning. While DSLs can make LLM outputs more structured, they typically lack large-scale training corpus, which can lead to unexpected behavior [3].

LLMs as Planners Another line of work treats LLMs as planners that orchestrate external tools for 3D creation. LayoutGPT [7] uses LLMs to generate layouts, while SceneCraft [11] and LayoutVLM [20] adopt an analysis-by-synthesis loop in which a vision-language critic evaluates rendered scenes and guides refinement. When combined with tool-use frameworks like ReAct [26], LLM agents can scale to interactive, photorealistic environments [25].

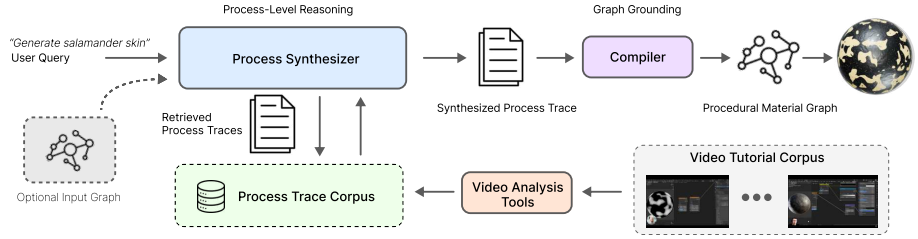


Fig. 3: Overview: Given a user query and optionally an input graph, the ProcessSynthesizer retrieves relevant process traces from a corpus of expert traces and synthesizes a target process trace aligned with the query. These traces are extracted automatically from tutorial videos using video analysis tools that convert demonstrations into textual records of construction steps, parameters, and design intent. The Compiler then grounds the synthesized trace into an executable Blender procedural material graph.

VLMs for Materials LLMs and vision-language models have also been applied directly to material synthesis. VLMaterial [16] fine-tunes a VLM to generate procedural materials represented as node graphs and BlenderAlchemy [12] iteratively samples and refines node graphs using VLM-based evaluation of rendered outputs, demonstrating that large models can reason about procedural material programs and visual feedback. In contrast, our method is training-free, leverages process-trace reasoning, and uses in-context learning from expert material generation workflows to achieve high-quality procedural material synthesis and editing.

3 Method

Procedural material creation is not merely graph assembly, but the execution of a structured reasoning trajectory. Skilled artists iteratively refine structure, adjust parameters, and articulate intent; the final node graph is only the endpoint of this process. We therefore distinguish the three representations used by MaterialApprentice: a *process trace*, which records construction steps, parameters, and design intent; a *material graph*, the executable shader-node layout compiled from the trace; and a *procedural material*, the resulting editable shader asset. While prior systems operate directly in graph space, we perform generation in process space.

To operationalize this formulation, MaterialApprentice instantiates retrieval-time process reasoning. As summarized in Figure 3, given a user query, it retrieves relevant process traces and synthesizes a process trace via ProcessSynthesizer aligned with the intended material. This trace is then grounded into an executable Blender shader graph through the Compiler, explicitly separating reasoning from execution. Process traces are derived from expert video demonstrations using video analysis tools which convert raw tutorials into structured, agent-friendly textual representations. These traces form the knowledge base for retrieval-time reasoning. Figure 4 details the ProcessSynthesizer, DeCompiler, and video analysis tools, which are described in the following subsections.

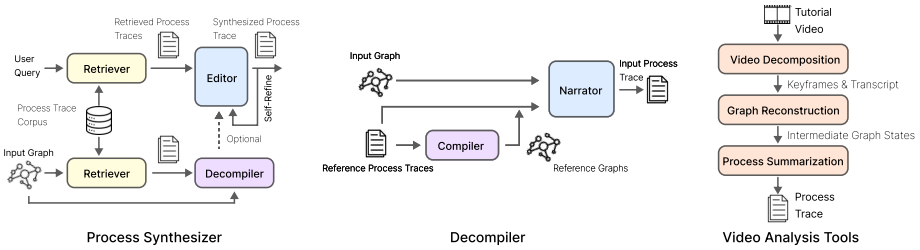


Fig. 4: Components of MaterialApprentice. (Left) ProcessSynthesizer: Given a user query, the retriever selects relevant process traces, which condition the Editor to synthesize a target trace. For editing, an input graph is first converted into a process trace by the DeCompiler. (Middle) DeCompiler: An input material graph is converted into a process trace using exemplar graph–trace pairs; the Narrator matches graph substructures to similar patterns in the exemplars and transfers their associated construction steps. (Right) Video analysis tools: Tutorials are decomposed into keyframes and transcripts, intermediate graph states are reconstructed, and multimodal signals are summarized into process traces for the ProcessSynthesizer.

3.1 Process Synthesizer

The ProcessSynthesizer is the core reasoning module of MaterialApprentice, synthesizing process traces aligned with user intent. Given a query q, a Retriever selects relevant reference traces P from the demonstration corpus C_P. Conditioned on q and P, the Editor synthesizes a target trace p describing how to construct the desired material.

For editing, ProcessSynthesizer additionally accepts an input graph g_i. In this setting, the Retriever selects process traces relevant to g_i, and a DeCompiler module reconstructs a corresponding process trace p_i. This inferred process trace, together with the user's edit instruction, conditions the Editor to synthesize an updated process trace consistent with the requested modification.

Retriever. The Retriever selects relevant demonstrations for both generation and editing. Rather than ranking examples by surface appearance alone, it uses the pretrained LLM to compare how materials are constructed. Given material names and descriptions in C_P, it predicts which expert demonstrations best match the user's intent, desired material properties, and construction strategy. The top-K corresponding process traces are selected to condition synthesis.

Editor. The Editor is a pretrained LLM prompted to perform in-context synthesis in the *process* domain. Conditioned on P and q, it generates a target process trace p specifying the construction sequence, parameter choices, and compositional structure required to realize the requested material. As illustrated in Fig. 5, the Editor combines construction patterns from retrieved traces into a target trace before compilation. Here, lowercase p denotes a single trace, while P refers to sets of retrieved process traces from C_P. To improve reliability, the Editor employs a lightweight self-refinement step, critiquing and revising its draft before finalizing p, enhancing consistency and adherence to the query. Because the

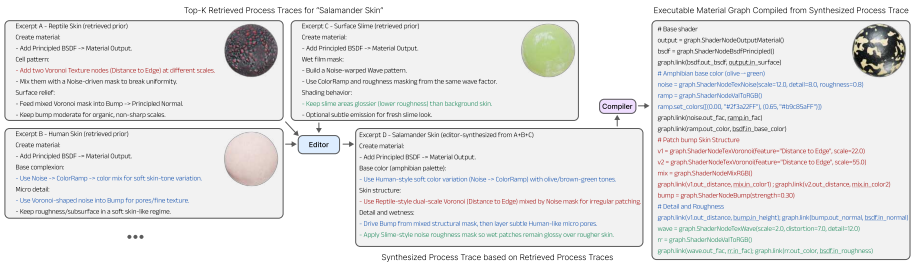


Fig. 5: Process-trace synthesis by reflecting on retrieved demonstrations. (Left) Top-K retrieved process traces for the query "salamander skin". Although visually distinct, these demonstrations contain useful construction patterns—scale structure from reptile skin, fine bump detail from human skin, and smooth organic noise from surface slime. (Center) The Editor uses these retrieved traces to synthesize a target process trace that combines their construction strategies for the query. (Right) The Compiler translates the synthesized trace into an executable material graph program, which produces the final material.

Editor operates purely in process space, editing requires representing existing graphs in the same modality, motivating the DeCompiler module.

DeCompiler. The DeCompiler converts an input material graph g_i into a process trace, enabling editing within a unified representation. Given g_i, it reconstructs a plausible trace p_i using exemplar graph–trace pairs from the reference corpus. Each reference trace is first compiled into graph form via the Compiler, yielding exemplar pairs $E = \{(g_j, p_j)\}_{j=1}^{k}$. The exemplars are selected to match the input graph in construction strategy rather than surface appearance. A specialized LLM module, Narrator, then reconstructs $p_i = \textsc{Narrator}(g_i; E)$ by matching substructures of g_i to similar graph patterns in the exemplars and transferring their associated construction steps. The resulting p_i captures the operations, parameters, and intent implied by the input graph.

3.2 Grounding Process Traces in Material Graphs

The Compiler translates textual process traces into executable Blender shader graphs. It is implemented as a pretrained LLM prompted to generate Blender Python graph-construction code conditioned on a trace p. A lightweight self-refinement stage enforces consistency between node topology, parameter definitions, and layer ordering. Given p, the compiler produces an executable graph g. The Compiler generates multiple candidate programs N per synthesized process trace and selects the first syntactically valid and executable instance. The compiler targets Blender versions 3.0–4.5, with custom node groups for compatibility across API changes.

3.3 Video Tutorials for Extracting Process Traces

To support retrieval-time process reasoning, we construct a corpus of expert Blender material tutorials. The corpus consists of 158 publicly available YouTube tutorials from established creators, selected for clear stepwise instruction, synchronized narration, and focused visual presentation (e.g., zoomed-in node editing,

explicit parameter explanation, and structured workflow progression). It spans 29 material categories (e.g., metals, fabrics, organic surfaces), with an average of 5.5 tutorials for each. Tutorials average 12.6 minutes in duration, with materials containing on average 14.3 shader nodes, providing rich signals about compositional structure, parameter tuning, and procedural reasoning.

In addition, we create 10 original tutorials for which we hold full rights and will release publicly to ensure long-term reproducibility. Publicly sourced tutorials are used exclusively as a retrieval-time knowledge source; no model training is performed on raw video content. We release YouTube IDs and category annotations to enable reconstruction of the corpus while preserving attribution to original creators. Detailed statistics are provided in the supplementary.

3.4 Process Extraction via Video Analysis Tools

To derive process traces from expert demonstrations, MaterialApprentice uses automatic video analysis tools to convert tutorials into structured text that LLM agents can directly retrieve and reason over. This component is primarily an engineering step rather than a core research contribution: the goal is simply to automatically extract procedural signals from demonstrations so that they can be used for retrieval-time reasoning. In practice, Blender tutorials exhibit a simple box-and-wire interface, allowing standard detection and segmentation models to reliably recover graph structure. Given a tutorial video v, the pipeline extracts keyframes, reconstructs intermediate graph states, and aligns visual edits with narrated explanations.

Video Decomposition. Videos are first decomposed into frames and audio narration. Informative keyframes are selected to capture meaningful transitions in the material construction process while removing redundant views. The audio stream is transcribed to preserve the artist's verbal explanations and design rationale.

Graph Reconstruction. The visual stream is analyzed to recover the evolving node graph. From selected keyframes, the system identifies node layouts, connectivity patterns, and parameter states using standard detection and segmentation models fine-tuned for the Blender shader editor. This produces a sequence of intermediate graph representations reflecting the progressive assembly of the material.

Process Summarization. A Summarizer module integrates the reconstructed graph states with the transcribed narration to produce a compact textual process trace p_v, capturing both the sequence of construction steps and the accompanying procedural reasoning expressed by the artist. These traces populate the retrieval corpus used by the ProcessSynthesizer at inference time.

Robustness of Process Extraction. Three design choices make extraction reliable. (1) *Grounding*: frame understanding relies on explicit vision tasks – node, socket, and wire detection – rather than direct LLM inference. Given the structured box-and-wire layout of the Blender shader editor, these detectors achieve mAP50 = 0.98 and mIoU = 0.87. (2) *Selection*: a fine-tuned frame-selection model retains only visually parseable frames robust to zoom, pan, and occlusion. (3) *Redundancy*: frames are parsed at 0.2 Hz and combined with narration through

an LLM summarizer, mitigating localized perception errors and producing stable process traces. Further, while the pipeline contains multiple stages, errors in early stages are mitigated by the process abstraction. The ProcessSynthesizer reasons over construction patterns rather than exact node configurations, allowing it to tolerate minor inaccuracies in reconstructed graph states or narration alignment. In practice, we observe that small parsing errors rarely propagate to catastrophic failures in the final material graph. Further details are provided in the supplementary.

3.5 Implementation Details

All LLM-based components—Retriever, Editor, DeCompiler, Compiler, and Summarizer—use the same pretrained GPT-5 model, differing only in prompt structure and contextual conditioning. The Retriever selects the top $K = 3$ process traces per query, balancing contextual coverage and computational efficiency. The compiler generates $N = 10$ programs. While processes synthesized by the ProcessSynthesizer are typically coherent and faithful to the query, final material quality may be further refined through established procedural optimization techniques [12]. A single process trace is $\approx 8k$ tokens, a single run costs $\approx$ 2USD and runs for 30 mins.

4 Experiments

Our experiments probe the implications of viewing procedural material generation as *retrieval-time process reasoning*. Section 4.1 evaluates the key design choices implied by this formulation—reasoning in process space rather than graph space and grounding execution through the Compiler. Section 4.2 then situates MaterialApprentice within existing procedural material systems through comparisons on established generation and editing benchmarks. Finally, Section 4.3 presents ablations analyzing generalization, retrieval behavior, the impact of video analysis tools, and the reproducibility of our framework through experiments on in-house tutorial videos.

4.1 Reasoning in the Process Domain

To evaluate process-space reasoning, we isolate the representation used for retrieval while keeping the output modality fixed. Our method retrieves *process traces*, while the baseline retrieves *material graphs*; both variants synthesize process traces that artists follow to construct and refine material graphs. This isolates the effect of structured procedural knowledge relative to raw graph structure.

Evaluation setup. We recruit five Blender artists (avg. 7.5 years of procedural-material experience) and evaluate 60 synthesized process traces from 30 prompts under two retrieval conditions. For each prompt, artists see anonymized, randomly ordered process traces from the two variants. The study has two stages: artists first construct a material graph by following each trace, then refine the graph for up to ten minutes to better match the input prompt. We record atomic graph

Edit Effort			Expert Preference		Editing Sentiment	
Metric	**Proc.**	**Graph**	**Criterion**	**Pref.**	**Signal**	**Δ(P-G)**
Node edits ↓	**2.42**	3.42	Impl. clarity ↑	**100%**	Confusion ↓	**-0.15**
Conn. edits ↓	**5.42**	7.42	Proc. strategy ↑	**83%**	Hesitation ↓	**-0.10**
Param. tweaks ↓	5.75	**4.83**	Outcome quality ↑	**92%**	Satisfaction ↑	**+0.40**
Struct. edits ↓	**7.84**	10.84	Param. control ↑	**83%**	Completion ↑	**+0.25**
Total edits ↓	**13.58**	15.67	Pedagogy ↑	**84%**		
			Final render ↑	**83%**		

Table 1: Expert Study. (Left) Average refinement edits. (Middle) LLM-assigned preferences from artists' think-aloud commentary; final render preference is provided by artists after refinement. (Right) Process−Graph sentiment deltas after mapping low/medium/high to 1/2/3. Process-derived graphs require fewer structural/total edits, are preferred on key criteria, and reduce editing friction.

edits—node edits, connection edits, and parameter refinements—as a direct measure of corrective editing effort.

Artists also follow a think-aloud protocol [6] during construction and refinement, verbalizing their understanding of each trace, the rationale behind their edits, and their assessment of the resulting graph. We ask them to comment on implementation clarity, procedural strategy, expected outcome quality, parameter control, and pedagogical value. An LLM then analyzes the recorded narrations with a fixed rubric to assign structured pairwise preferences and editing-friction labels. Final render preference is collected separately after refinement. Table 1 summarizes the study along three axes: edit effort, expert preference, and think-aloud sentiment. We provide the GPT-5 analysis prompts in the supplement.

Editing effort. The left block of Table 1 reports edits required to refine the generated graphs. Node and connection edits modify graph structure and therefore reflect changes to the underlying procedural strategy, while parameter refinements tune existing controls such as sliders and color ramps. Process-conditioned traces require fewer structural edits than graph-conditioned traces (7.84 vs. 10.84) and fewer total edits overall (13.58 vs. 15.67), indicating that they produce graphs closer to the intended procedural structure. They require slightly more parameter tweaks (5.75 vs. 4.83), which we interpret as a favorable shift: artists spend less effort repairing structure and more effort tuning exposed controls.

Expert preference. The middle block reports LLM-assigned preferences from artists' think-aloud commentary; final render preference is provided directly by artists after refinement. Process-conditioned workflows are favored across all criteria, especially implementation clarity (100%) and outcome quality (92%), with strong preferences for procedural strategy (83%), parameter control (83%), pedagogical value (84%), and final render quality (83%). This indicates that process traces provide clearer construction guidance beyond final appearance.

Editing sentiment. The right block analyzes editing friction from think-aloud narrations. An LLM assigns low/medium/high labels to four signals: *confusion* and *hesitation*, which indicate difficulty understanding the graph or choosing the

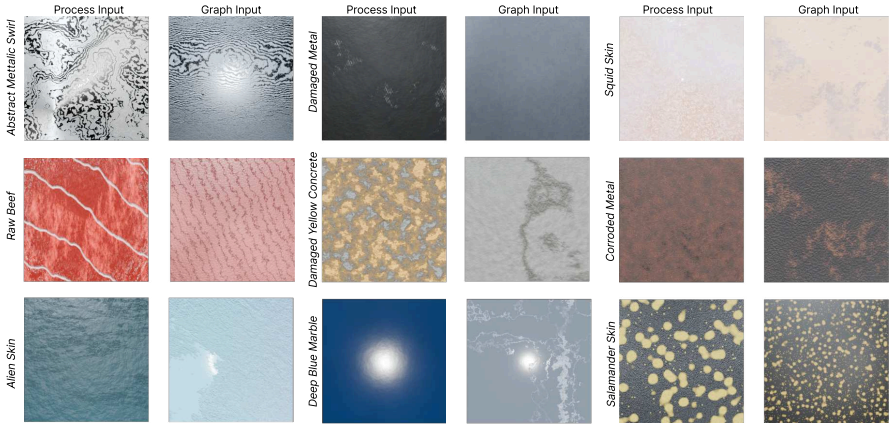


Fig. 6: Qualitative comparison of fully automatic generation using process-conditioned and graph-conditioned retrieval. Each pair shows outputs from the complete pipeline without manual editing. Process-conditioned generation better captures the intended procedural structure and visual characteristics, while graph-conditioned retrieval often produces oversimplified or structurally inconsistent patterns.

next edit, and *satisfaction* and *completion*, which indicate successful refinement and confidence in the result. After mapping labels to $1/2/3$, process-conditioned workflows reduce confusion (-0.15) and hesitation (-0.10), while increasing satisfaction ($+0.40$) and completion ($+0.25$). This suggests that process-derived graphs are easier for artists to understand, modify, and finalize.

Fully automatic generation. Figure 6 evaluates the complete pipeline without manual editing. For 50 diverse material prompts, we compare paired outputs from process-conditioned and graph-conditioned retrieval. Three experts judge each pair against the input prompt, with majority vote determining preference. Process-conditioned outputs are preferred in 72% of comparisons, showing that process retrieval improves not only artist-facing editability but also automatic prompt alignment. Qualitatively, process-conditioned retrieval better captures hierarchical procedural structure, including directional fibers, layered corrosion, and structured surface noise, whereas graph-conditioned retrieval often yields simpler or structurally incorrect patterns.

Need for Compiler. We evaluate a variant where the EDITOR directly generates material graphs instead of processes. We measure reliability by *execution rate*: the fraction of prompts for which at least one candidate program runs successfully in Blender. Across 100 prompts, direct graph generation fails in 30% of cases, whereas our full process-to-compiler pipeline achieves a 100% execution rate.

4.2 Comparison with Prior Procedural Material Systems

We compare MATERIALAPPRENTICE against three state-of-the-art procedural material systems: BlenderAlchemy [12], BlenderMCP [2], and VLMaterial [16]. We also report a Nearest-Neighbor baseline (the closest retrieved process) to verify

Image-to-Material Generation							Text-based Material Editing				
Dataset	**Method**	**CLIP ↓**	**Style Loss ↓**	**SWD ↓**	**GPT-5 Pref. ↑**	**User Pref. (%) ↑**	**Dataset**	**Method**	**CLIP ↓**	**GPT-5 Pref. ↑**	**User Pref. (%) ↑**
BlenderKit	VLMaterial	**0.856**	**0.040**	**2.44**	0.660	80	**Coarse**	BlenderAlchemy	0.244	0.611	85
	BlenderMCP	0.781	0.061	4.428	0.835	86		BlenderMCP	**0.237**	0.611	80
	Nearest Neighbour	0.791	0.059	4.326	0.874	84		Nearest Neighbour	0.238	0.722	100
	Ours	0.852	0.043	2.567	–	–		Ours	0.260	–	–
In the Wild	VLMaterial	0.764	0.066	8.973	0.771	89	**Fine-grained**	BlenderAlchemy	0.240	0.643	81
	BlenderMCP	0.737	0.080	8.227	0.800	85		BlenderMCP	**0.215**	0.929	100
	Nearest Neighbour	0.748	0.071	7.373	0.714	87		Nearest Neighbour	0.228	0.786	100
	Ours	**0.768**	**0.065**	**5.949**	–	–		Ours	0.253	–	–

Table 2: Quantitative comparison of image-to-material generation (Left) and text-based material editing (Right) across datasets. User preference (%) reports the fraction of comparisons where users preferred **Ours** over each baseline. GPT-5 and Users prefer our editing over baselines.

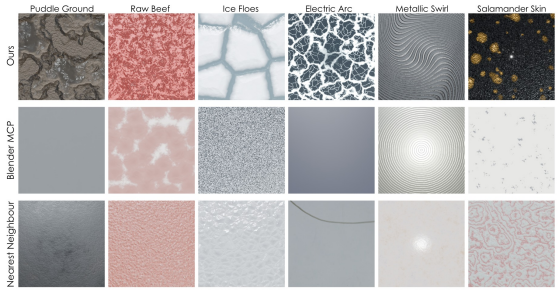


Fig. 7: Text-conditioned Material Generation. Our method produces visually diverse materials across a wide range of text prompts. Unlike BlenderMCP, our outputs align more closely with the user prompt while remaining distinct from the nearest retrieved example.

that MaterialApprentice isn't simply copying demonstrations. All baselines and MaterialApprentice modules use a GPT-5 backbone for fairness.

For generation, perceptual similarity is measured following [16] using CLIP cosine similarity, style loss, and sliced Wasserstein distance (SWD). For editing, where no reference render exists, we follow [12] and compute CLIP similarity between the edit instruction and the generated output. We additionally conduct GPT-5 and user forced-choice studies: for generation, evaluators receive the target and two outputs; for editing, they receive the initial material, the edit instruction, and two candidate results. We recruit 150 users on Amazon MT, with comparisons rated by three annotators and aggregated by majority vote.

Procedural Material Generation. We evaluate all methods, excluding BlenderAlchemy (which does not support prompt-based generation), on two benchmarks: (i) 110 rendered materials from BlenderKit and (ii) 35 in-the-wild close-up photographs. Each prompt includes a short text description (e.g., "banana skin") and a reference image. Figure 7 shows text-to-material generation results. MaterialApprentice produces materials more closely aligned with textual prompts, achieving higher CLIP similarity (0.806 vs. 0.791 for nearest neighbor and 0.781 for BlenderMCP). Users prefer our results in 86% and 92% of comparisons against nearest neighbor and BlenderMCP, respectively. Table 2 reports quantitative results. MaterialApprentice achieves the strongest performance on the challenging in-the-wild benchmark and is competitive with VLMaterial on BlenderKit, where VLMaterial is fine-tuned. CLIP, Style Loss, and SWD capture image-level similarity and are sensitive to scale, alignment, and rendering, while GPT-5 and user preferences reflect prompt faithfulness and procedural plausibility; these metrics therefore provide complementary views of quality. Across both preference studies, MaterialApprentice is consistently favored over competing methods.

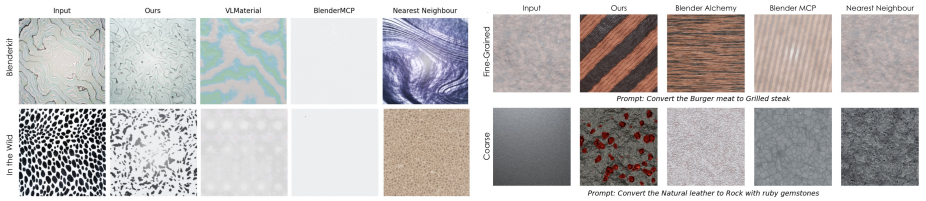


Fig. 8: Image conditioned Material Generation. (Left) Images include nacre and dalmatian material. Our method better captures micro and macro details specified via the image reference. **Text conditioned Material Editing.** (Right) Our method enables more precise and reliable manipulation of material graphs. For coarse edits, it makes the substantial structural changes required to achieve the intended transformation, while for fine edits, it can introduce subtle, localized modifications—producing nuanced effects that competing baselines fail to capture.

Figure 8 (left) shows qualitative comparisons. MaterialApprentice reproduces both macro- and micro-scale structures—from nacre patterns to dalmatian textures—while VLMaterial often produces coarse approximations and BlenderMCP drifts from the target appearance. Our outputs remain clearly distinct from nearest-neighbor examples, indicating genuine synthesis rather than retrieval.

Procedural Material Editing. We evaluate BlenderAlchemy [12], BlenderMCP [2], and MaterialApprentice on two editing tasks: 25 coarse edits (e.g., wood $\rightarrow$ marble) and 25 fine-grained edits (e.g., adding moss to brick), using base materials from BlenderKit. Coarse edits require substantial restructuring of the shader graph, while fine edits involve localized modifications.

As shown in Table 2, GPT-5 consistently prefers edits produced by MaterialApprentice over BlenderMCP and nearest-neighbor baselines. Qualitative examples (Fig. 8, right) show that baselines often fail to incorporate key prompt attributes—for example, missing structural features—while MaterialApprentice introduces both large-scale structural changes and subtle surface details.

4.3 Ablations

We now analyze several aspects of MaterialApprentice including its ability to generalize beyond the materials present in the process corpus and the effect of the number of retrieved demonstrations used during synthesis.

Generalization beyond the process corpus. To test whether MaterialApprentice relies on category-level cues from the corpus, we remove entire material categories corresponding to the evaluation prompts—Organic (blue jay egg), Concrete (cracked concrete), and Marble (midnight gold marble). Figure 9 shows that even under this setting, MaterialApprentice continues to synthesize plausible procedural structures that capture the essential characteristics of the input images, indicating that the model generalizes procedural strategies rather than memorizing category-specific examples.

Effect of the number of retrieved demonstrations. We next vary the number of demonstrations provided as context during synthesis (Fig. 9). Even a single retrieved process trace substantially improves generation quality compared to using no demonstrations. However, the benefit saturates quickly, with limited

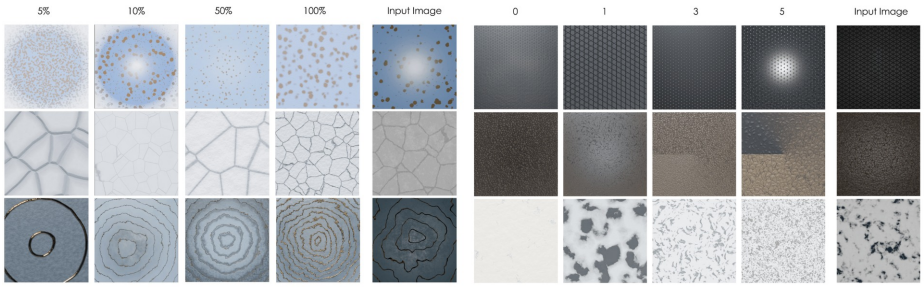


Fig. 9: Effect of Video Corpus Size on Generation Quality. (Left) Our retrieval-based approach maintains strong generation quality even when given only a small fraction of the tutorial corpus—evaluated at 5% to 100% availability. This demonstrates that MaterialApprentice can synthesize novel, high-quality materials by reflecting on only a handful of expert demonstrations. **Effect of Number of Retrieved Demonstrations.** (Right) We evaluate MaterialApprentice using 0 (no retrieval), 1, 3, and 5 retrieved demonstrations. Quality improves from 0 to 3 retrieved process traces, with the largest gain from the first example; beyond three, improvements are marginal relative to the token cost.

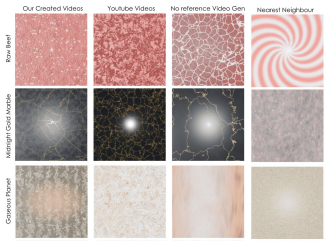
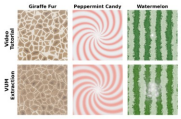


Fig. 10: Generation using our curated video set. Materials generated by MaterialApprentice using only our released tutorial videos as references (left column). Results remain strong across diverse prompts and differ clearly from YouTube-based generation, generation without references, and the nearest neighbor baseline.

improvement beyond three retrieved examples, suggesting that a small number of relevant demonstrations is sufficient to guide effective process synthesis.

Reproducibility with a curated video set. To further demonstrate reproducibility, we evaluate MaterialApprentice using a curated set of tutorial videos that we created ourselves and release publicly with full rights. Figure 10 shows text-conditioned generation results when the reference pool is restricted to this small set. Despite the limited corpus, MaterialApprentice continues to synthesize diverse materials that differ clearly from both nearest-neighbor retrieval and generation without demonstrations, indicating that even a handful of shareable tutorials provides sufficient process-level cues for effective material synthesis.

Impact of Video Analysis Tools. We analyze the contribution of our video analysis pipeline by constructing three variants: (1) *Audio-only*, which ignores the visual stream and relies solely on narration; (2) *GPT-5 Frame Parsing*, which replaces our specialized detectors with GPT-5-based frame interpretation; and (3) *Random Keyframe Selection*, which removes our informativeness-based keyframe filtering. We evaluate all variants on 100 tutorial videos by reconstructing material graphs from extracted process traces and comparing them with ground-truth graphs. Node detection and topology reconstruction are evaluated using precision, recall, and F1 scores. Parameter fidelity is measured as the normalized absolute difference



Method	N-Prec ↑	N-Rec ↑	N-F1 ↑	T-Prec ↑	T-Rec ↑	T-F1 ↑	Params ↑	SWD ↓	Edits ↓
Ours (Full)	**98.60**	**98.72**	**98.65**	**83.16**	**83.01**	**83.07**	**85.96**	**0.0304**	**2.66**
Audio-Only	88.24	93.13	90.37	60.58	64.64	62.25	78.32	0.0528	4.02
GPT-5 Frame Parsing	77.98	83.42	79.99	53.98	59.74	56.23	69.84	0.1572	4.66
Random Keyframe Selection	89.21	91.77	90.21	62.58	65.51	63.71	76.92	0.0585	3.78

Fig. 11: Video analysis tools ablation. (Left) Materials reconstructed from VAT-extracted process traces closely match those recreated by humans from the original tutorials. (Right) Quantitative comparison with three ablations: audio-only extraction, GPT-5 frame parsing, and random keyframe selection. Our full pipeline achieves the best node accuracy, topology reconstruction, parameter fidelity, perceptual similarity, and lowest edit distance.

between predicted and ground-truth parameters, while perceptual similarity is computed using sliced Wasserstein distance (SWD). We additionally report graph edit distance, defined as the number of atomic operations required to transform the reconstructed graph into the ground-truth graph (node additions/removals, connection changes, and parameter adjustments), estimated by prompting an LLM to produce the minimal edit sequence. As shown in Fig. 11, our full pipeline consistently achieves the best performance across all metrics and produces graphs that most closely match the original materials. These results highlight the importance of jointly modeling visual structure, narration, and keyframe selection for reliable process extraction from tutorials.

5 Conclusion

Procedural material design is inherently process-driven: artists construct materials through layered operations, parameter tuning, and physical reasoning. We frame procedural material generation as retrieval-time process reasoning over expert demonstrations, elevating process to a first-class representation beyond graph-only synthesis. By retrieving process traces and grounding them into executable shader graphs, MaterialApprentice produces materials that better reflect expert workflows while remaining interpretable and editable. More broadly, this perspective suggests generative systems that learn directly from creative practice rather than static artifacts alone. Extending process reasoning to multi-step design workflows, additional authoring tools beyond Blender, and larger corpora of demonstrations are promising directions for future work. Failure cases are discussed in the supplementary. Code, models, and data will be publicly released.

6 Acknowledgments

This work was supported in part by NSF grant 2402583 and gifts from Qualcomm, Google, and Adobe. We are grateful to the Blender experts who participated in our expert study: Barnabas Thomas, John Arquero, Benjamin Olawuni, Ricardo Raimundo, and Tanushree Roychoudhury. We also thank Krishna Chaitanya, Harsh Sinha, Kanav, and Diptimayee Gupta for their inspiration, support, and helpful discussions throughout this work.

References

1. Adobe: Substance 3d designer (2023)
2. Ahuja, S.: Blender-mcp. https://github.com/ahujasid/blender-mcp (2025), accessed: 2025-11-06
3. Barbero Jiménez, Á.: An evaluation of llm code generation capabilities through graded exercises. arXiv preprint arXiv:2410.16292 (2024)
4. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.D.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al.: Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 (2021)
5. Cheng, B., Misra, I., Schwing, A.G., Kirillov, A., Girdhar, R.: Masked-attention mask transformer for universal image segmentation. In: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. pp. 1290–1299 (2022)
6. Ericsson, K.A., Simon, H.A.: Verbal reports as data. Psychological review **87**(3), 215 (1980)
7. Feng, W., Zhu, W., Fu, T.j., Jampani, V., Akula, A., He, X., Basu, S., Wang, X.E., Wang, W.Y.: Layoutgpt: Compositional visual planning and generation with large language models. NeurIPS **36**, 18225–18250 (2023)
8. Guerrero, P., Hasan, M., Sunkavalli, K., Mech, R., Boubekeur, T., Mitra, N.: Matformer: A generative model for procedural materials. ACM Transactions on Graphics **41**(4), 46:1–46:12 (2022)
9. Guo, Y., Smith, C., Hašan, M., Sunkavalli, K., Zhao, S.: Materialgan: Reflectance capture using a generative svbrdf model. ACM TOG **39**(6), 254:1–254:13 (2020). https://doi.org/10.1145/3414685.3417779
10. Hu, Y., Guerrero, P., Hasan, M., Rushmeier, H., Deschaintre, V.: Generating procedural materials from text or image prompts. In: ACM SIGGRAPH 2023 conference proceedings. pp. 1–11 (2023)
11. Hu, Z., Iscen, A., Jain, A., Kipf, T., Yue, Y., Ross, D.A., Schmid, C., Fathi, A.: Scenecraft: An llm agent for synthesizing 3d scenes as blender code. In: Forty-first International Conference on Machine Learning (2024)
12. Huang, I., Yang, G., Guibas, L.: Blenderalchemy: Editing 3d graphics with vision-language models. In: ECCV. pp. 297–314. Springer (2024)
13. Huang, X., Wang, T., Liu, Z., Wang, Q.: Material anything: Generating materials for any 3d object via diffusion. In: Proceedings of the Computer Vision and Pattern Recognition Conference. pp. 26556–26565 (2025)
14. Karras, T., Laine, S., Aittala, M., Hellsten, J., Lehtinen, J., Aila, T.: Analyzing and improving the image quality of StyleGAN. In: CVPR (2020)
15. Li, B., Hu, Y., Guerrero, P., Hasan, M., Shi, L., Deschaintre, V., Matusik, W.: Procedural material generation with reinforcement learning. ACM TOG **43**(6), 1–14 (2024)
16. Li, B., Wu, R., Solar-Lezama, A., Zheng, C., Shi, L., Bickel, B., Matusik, W.: Vlmaterial: Procedural material generation with large vision-language models. In: ICLR (2025)
17. Ma, X., Deschaintre, V., Hašan, M., Luan, F., Zhou, K., Wu, H., Hu, Y.: Materialpicker: Multi-modal dit-based material generation. ACM Transactions on Graphics (TOG) **44**(4), 1–12 (2025)
18. Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., et al.: Self-refine: Iterative refinement with self-feedback. NeurIPS **36**, 46534–46594 (2023)

19. Sun, C., Han, J., Deng, W., Wang, X., Qin, Z., Gould, S.: 3d-gpt: Procedural 3d modeling with large language models. In: 2025 International Conference on 3D Vision (3DV). pp. 1253–1263. IEEE (2025)
20. Sun, F.Y., Liu, W., Gu, S., Lim, D., Bhat, G., Tombari, F., Li, M., Haber, N., Wu, J.: Layoutvlm: Differentiable optimization of 3d layout via vision-language models. In: Proceedings of the Computer Vision and Pattern Recognition Conference. pp. 29469–29478 (2025)
21. Varghese, R., M., S.: Yolov8: A novel object detection algorithm with enhanced performance and robustness. In: 2024 International Conference on Advances in Data Engineering and Intelligent Computing Systems (ADICS). pp. 1–6 (2024). https://doi.org/10.1109/ADICS58448.2024.10533619
22. Vecchio, G., Martin, R., Roullier, A., Kaiser, A., Rouffet, R., Deschaintre, V., Boubekeur, T.: Controlmat: a controlled generative approach to material capture. ACM Transactions on Graphics **43**(5), 1–17 (2024)
23. Yamada, Y., Chandu, K., Lin, B.Y., Hessel, J., Yildirim, I., Choi, Y.: L3go: Language agents with chain-of-3d-thoughts for generating unconventional objects. In: Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (System Demonstrations). pp. 456–469 (2025)
24. Yang, Y., Sun, F.Y., Weihs, L., VanderBilt, E., Herrasti, A., Han, W., Wu, J., Haber, N., Krishna, R., Liu, L., Callison-Burch, C., Yatskar, M., Kembhavi, A., Clark, C.: Holodeck: Language guided generation of 3d embodied ai environments. In: CVPR. pp. 16227–16237 (2024)
25. Yao, K., Zhang, L., Yan, X., Zeng, Y., Zhang, Q., Xu, L., Yang, W., Gu, J., Yu, J.: Cast: Component-aligned 3d scene reconstruction from an rgb image. ACM TOG **44**(4), 1–19 (2025)
26. Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K.R., Cao, Y.: React: Synergizing reasoning and acting in language models. In: ICLR (2023), https://openreview.net/forum?id=WE_vluYUL-X
27. Zhang, Y., Liu, Y., Xie, Z., Yang, L., Liu, Z., Yang, M., Zhang, R., Kou, Q., Lin, C., Wang, W., et al.: Dreammat: High-quality pbr material generation with geometry-and light-aware diffusion models. ACM Transactions on Graphics (TOG) **43**(4), 1–18 (2024)
28. Zhou, X., Hasan, M., Deschaintre, V., Guerrero, P., Hold-Geoffroy, Y., Sunkavalli, K., Kalantari, N.K.: Photomat: A material generator learned from single flash photos. In: ACM SIGGRAPH 2023 conference proceedings. pp. 1–11 (2023)

7 Supplementary Video

Please watch the supplementary video for an overview of the method and highlights about key experiments.

8 Expert Study

8.1 Matched Graph-Space Baseline

To further isolate the role of the intermediate representation, we evaluate a matched *Graph*→*Graph* baseline in which both retrieval and generation operate

Metric	G→G	G→P	P→P
Preference (%) ↑			
Initial Output	6	18	**76**
After Target Refinement	12	23	**65**
After Instructional Edit	18	0	**82**
Target-Refinement Effort ↓			
Node edits (N)	8.5	5.8	**4.7**
Connection edits (C)	23.5	19.9	**15.3**
Parameter refinements (P)	**5.9**	8.5	8.1
Structural edits (N+C)	32.0	25.7	**20.0**
Total edits (N+C+P)	37.9	34.2	**28.1**
Instructional-Edit Effort ↓			
Total edits (parameters only)	**8.0**	10.9	10.7

Table 3: Matched representation ablation comparing Graph→Graph, Graph→Process, and Process→Process. Process→Process is preferred at all stages and requires the fewest structural and total edits during target refinement. Graph→Graph requires fewer parameter-only instructional edits, but its final edited outputs are rarely preferred, suggesting that fewer exposed controls do not imply better editability.

directly in graph space. This baseline retrieves material graphs and directly generates a material graph, removing the intermediate process representation. We compare it with *Graph→Process*, which retrieves graphs but synthesizes process traces, and our full *Process→Process* method, which retrieves and synthesizes in process space.

Evaluation protocol. We evaluate 17 materials across the three methods. For each material, artists are given the final generated graph from each method and perform two independent editing tasks. In *Target Refinement*, artists edit the graph to better match the reference material. In *Instructional Edit*, artists apply a semantic edit using only parameter changes. Both tasks start from the same initial output graph and are limited to ten minutes. Separately, two artists choose the best result within each method triplet at three stages: the initial output, after target refinement, and after instructional edit.

Results. Table 3 shows that *Process→Process* is preferred at all stages: 76% for the initial output, 65% after target refinement, and 82% after instructional edit. It also requires the fewest node edits, connection edits, structural edits, and total edits during target refinement, indicating that process-space reasoning produces graphs that are closer to the intended procedural structure. *Graph→Graph* requires fewer parameter-only edits during instructional editing, but this reflects the fact that its simpler graphs expose fewer tunable controls; despite requiring fewer parameter tweaks, its final edited outputs are rarely preferred. These results show that the advantage of process-space reasoning persists under a matched

graph-space baseline and improves both initial graph quality and artist-facing editability.

8.2 Parameter-only Semantic Editing

To further evaluate whether generated materials expose useful controls, we conduct a parameter-only semantic editing study. Unlike the main expert study, where artists may modify graph structure, this study fixes the generated material graph and permits only parameter changes. This isolates whether the graph produced by MaterialApprentice contains editable controls that support meaningful downstream refinement.

We evaluate 6 generated materials with 4 semantic edits per material, for a total of 24 editing tasks. Two Blender artists perform each edit using only exposed parameters, with at most 3 refinement attempts and a 5-minute time limit per edit. After each task, artists rate satisfaction with the final edit and ease of use on a 5-point Likert scale.

Artists report an average satisfaction score of 4.10/5 and an ease-of-use score of 3.75/5, with an average editing time of 1:48 minutes per edit. These results suggest that MaterialApprentice does not merely produce executable graphs, but also exposes parameters that support efficient semantic refinement without requiring structural graph edits.

8.3 Qualitative Examples

Figure 12 provides representative examples from the expert study comparing graph-conditioned and process-conditioned retrieval. For each prompt, artists first constructed a material graph by following the generated process trace and then refined the graph to better match the target material. The figure shows the input prompt, the initial and edited materials, the number of edits and editing time, and summarized think-aloud feedback from the artists.

Across these examples, process-conditioned retrieval typically produces graphs that start closer to the target appearance and require fewer corrective edits. For instance, in the *Alien Rock* and *Rough Leather* examples, graph-conditioned retrieval leads to substantial restructuring, including replacing procedural textures, reconnecting nodes, and debugging unexpected node behavior. In contrast, process-conditioned retrieval more often yields a graph with the right high-level structure, allowing artists to focus on parameter tuning such as scale, color, roughness, and bump strength.

The artist comments further illustrate this difference. Graph-conditioned outputs often cause confusion or trial-and-error editing when the node organization is difficult to interpret or when parameters have unclear effects. Process-conditioned outputs are described as easier to follow and quicker to refine, suggesting that process traces transfer not only node-level operations but also higher-level construction logic. These qualitative examples complement the quantitative results in Table 1, showing how process-conditioned retrieval reduces structural repair and improves artist-facing editability.

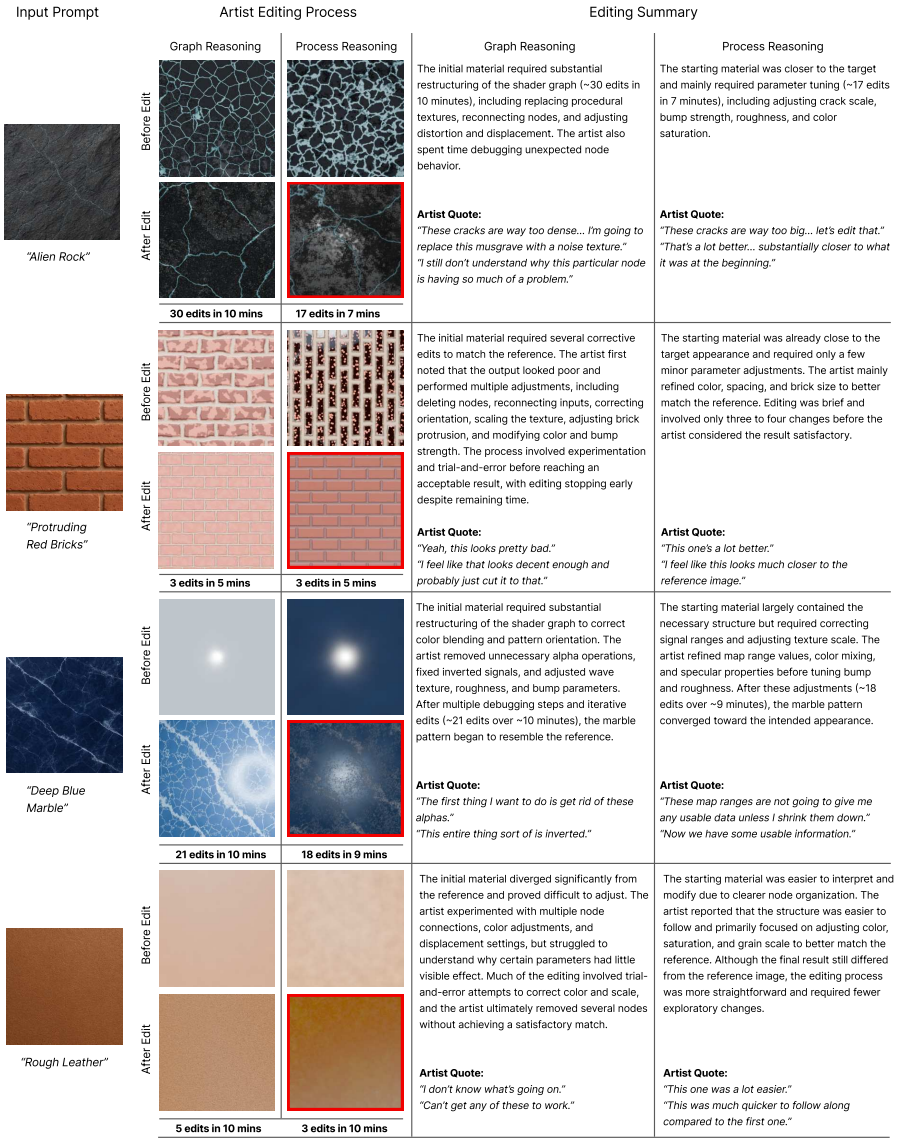


Fig. 12: Expert study comparing graph retrieval and process retrieval. Artists first constructed material graphs by following the generated process traces and then edited them to better match the target prompt. Edit count and editing time serve as proxies for process quality. Across examples, process-based retrieval produces materials that start closer to the target, require fewer edits, and are consistently preferred by artists (red borders), while graph retrieval often requires substantial restructuring of the shader graph. Artists also reported that process-derived graphs were easier to interpret and modify.

Category	#	Avg Nodes	Video (min)
Animal	5	15.8	10.7
Asphalt	2	22.0	19.0
Bricks	5	13.4	12.4
Ceramic	2	10.5	8.0
Concrete	4	19.5	13.5
Fabric	10	10.8	9.7
Floor	3	20.0	20.7
Food	11	11.5	11.8
Fruit	9	15.6	20.9
Fx	8	14.1	10.5
Glass	6	9.8	9.2
Grass	1	7.0	8.0
Ground	5	17.6	15.0
Human	4	14.3	11.8
Ice	4	13.0	14.5
Leather	3	15.7	11.3
Liquid	5	11.8	8.4
Marble	5	16.6	15.0
Metal	14	15.4	10.1
Organic	5	14.0	9.6
Ornaments	9	13.3	8.7
Paper	2	11.0	12.0
Paving	4	17.3	16.3
Plaster	4	13.0	12.5
Plastic	9	15.8	11.1
Rock	8	15.6	13.0
Sand	4	16.3	12.5
Tech	4	11.3	15.5
Wood	3	13.3	12.7

Table 4: Statistics of the tutorial video dataset used by MaterialApprentice to capture tacit expertise during procedural material generation.

Category	BlenderKit	In-the-Wild
Animal	6	8
Asphalt	2	1
Bricks	5	0
Ceramic	3	1
Concrete	4	1
Fabric	5	3
Floor	5	0
Food	8	0
Fruit	6	3
Fx	9	1
Glass	4	0
Grass	1	0
Ground	4	0
Human	1	1
Ice	3	0
Leather	5	1
Liquid	4	0
Marble	7	0
Metal	8	4
Organic	2	1
Ornaments	4	2
Plastic	6	1
Plaster	2	0
Rock	5	1
Sand	1	0
Tech	3	0
Wood	0	5
Paving	0	1

Table 5: Category distribution of test prompts across the BlenderKit and In-the-Wild datasets used in our evaluation. The table reports the number of materials per category in each dataset.

9 Video Tutorial Corpus

To enable MaterialApprentice to reflect on expert workflows, we curate a tutorial video corpus containing **158** publicly available Blender material–creation tutorials from established YouTube creators such as Ryan King Art, Ducky 3D, Sam Bowman, PIXXO 3D, BlenderBiteSize, and others. These videos provide diverse demonstrations of real expert practice that MaterialApprentice retrieves at inference time.

The corpus spans 29 material categories, with each category averaging 5.5 tutorials. Tutorials have an average duration of 12.57 minutes, and the demonstrated materials contain on average 14.32 nodes, offering rich signals about procedural structure, parameter tuning, and tacit artistic reasoning (see Table 4). We will release the YouTube IDs and category labels for all videos, allowing the corpus to be reconstructed while preserving attribution to the original creators.

This corpus is strictly used for retrieving process traces at inference time. Experiments in section 4.3/ Figure 10 (main paper) show that MaterialAp-

PRENTICE is able to generate novel materials with even a small fraction of the corpus which is further demonstrated by running MATERIALAPPRENTICE on the corpus of videos made by us (Figure 9).

10 Video Analysis Tools for Process Extraction

This section provides implementation details for the video analysis pipeline used to extract process traces from tutorial videos. As discussed in Sec. 3.4 of the main paper, this component is primarily an engineering step that converts demonstrations into structured signals usable for retrieval-time process reasoning.

Blender tutorial videos exhibit a consistent box-and-wire interface, allowing procedural structure to be reliably recovered using standard computer vision tools. Our pipeline automatically decomposes each tutorial video v into frames and narration, reconstructs intermediate node graphs, and summarizes these multimodal signals into a textual process trace p_v describing the material construction workflow.

The pipeline relies on lightweight pretrained detectors and segmentation models fine-tuned on Blender shader editor screenshots. Because the interface exhibits highly consistent layout, color schemes, and typography, standard architectures achieve high accuracy without extensive tuning. All stages operate automatically and require no manual annotation. We note that minor perception errors rarely propagate to the final generation stage, as the PROCESSSYNTHESIZER reasons over construction patterns rather than exact node configurations.

10.1 Video Decomposition

Each tutorial video v is decomposed into visual frames and audio narration. Let $X_v = \{(t_k, I_k)\}_{k=1}^{T_v}$ denote the frame sequence and a_v the audio stream.

Keyframe Selection. Tutorial videos often contain frames that are difficult to parse due to occlusions, extreme zoom, or limited visibility of the node graph. We therefore first identify frames that are suitable for reliable graph reconstruction. Frames are categorized according to their suitability for parsing, ranging from *useless* to *excellent*, as illustrated in Fig. 13.

To automatically estimate this quality, each frame I_k is assigned an informativeness score

$$s_k = \Phi_{\text{score}}(I_k), \tag{1}$$

which evaluates node visibility, text readability, graph coverage, and overall visual clarity. Technically, this is achieved via the scoring network Φ_{score} uses a YOLOv8n [21] backbone fine-tuned as a five-class classifier to predict frame informativeness (*excellent*, *good*, *fair*, *poor*, *useless*). The scoring process is illustrated in Fig. 14. We train the score model on our internally created and annotated dataset consisting of 1000 synthesized Blender screenshots labeled across the five categories for 50 epochs at 1080p resolution with a batch size of 128, achieving 70% top-1 accuracy.

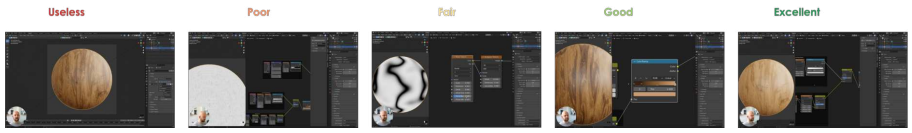


Fig. 13: Illustration of frame informativeness used for keyframe selection. Frames range from *useless* to *excellent* depending on their suitability for node-graph reconstruction. Low-quality frames exhibit occlusion, extreme zoom, or limited graph visibility, whereas high-quality frames clearly expose node layouts, connections, and parameter controls required for parsing.

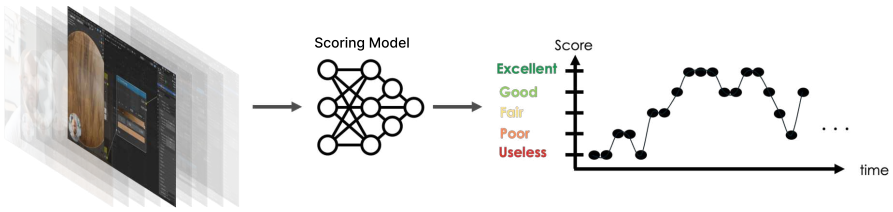


Fig. 14: Keyframe scoring pipeline. Video frames are evaluated by a scoring model that assigns an informativeness score based on node visibility, parameter readability, and graph coverage. The resulting score sequence over time identifies frames suitable for reliable graph reconstruction.

Frames with scores above a threshold τ_s form a candidate set

$$\mathcal{C}_v = \{(t_k, I_k) \mid s_k \geq \tau_s\}. \tag{2}$$

Even after filtering, tutorials may still contain many visually similar frames where the node graph remains unchanged while the instructor explains a concept. To reduce redundancy and improve efficiency, candidate frames are embedded using Φ_{emb} (CLIP) and clustered by cosine similarity. The final frame from each cluster is retained, capturing the most evolved graph state at that stage of the tutorial.

Narration Transcription. The audio stream a_v is transcribed using the Whisper-Large automatic speech recognition model, producing a time-aligned transcript

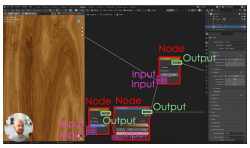
$$\tau_v = \Phi_{\text{asr}}(a_v). \tag{3}$$

The outputs of this stage are the keyframe set $X_v^{\mathcal{K}}$ and the aligned narration transcript τ_v.

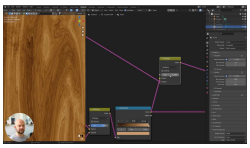
10.2 Graph Reconstruction

For each selected keyframe I_k, the pipeline reconstructs the corresponding node-graph state by detecting nodes, recovering UI parameters, and inferring graph connectivity.

Image-Level Parsing. We first identify the structural components of the shader graph. An object detector localizes nodes together with their input and output sockets, while a segmentation model extracts connection wires (see figure on the right):

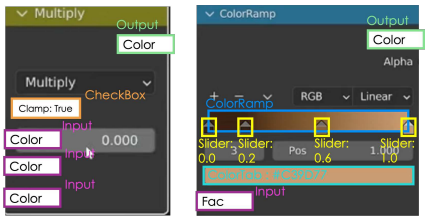


Node, Input and Output Socket Detection


Wire Connection Detection

$$\mathcal{N}_k, \mathcal{S}_k^{in}, \mathcal{S}_k^{out} = \Phi_{\text{det}}^{img}(I_k), \quad \mathcal{W}_k = \Phi_{\text{seg}}^{wire}(I_k). \tag{4}$$

OCR is then applied within detected node regions to recover node names and socket labels.



ColorRamp,Slider, CheckBox and ColorTab detection, inference and OCR

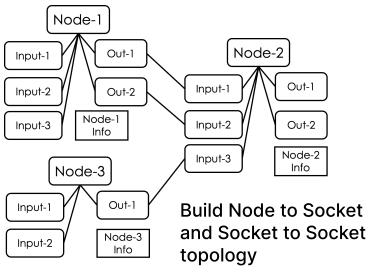
Node-Level Parameter Extraction. Each detected node region I_n is further analyzed to recover UI elements encoding parameter values. A lightweight node-level detector identifies interface components such as sliders, color ramps, checkboxes, and color tabs (see figure on the left):

$$\mathcal{U}_n = \Phi_{\text{det}}^{node}(I_n). \tag{5}$$

Slider positions are normalized within their UI bounds, while color tabs are converted to RGB values, producing node parameter states Θ_n.

Both the image-level detector Φ_{det}^{img} and the node-level detector Φ_{det}^{node} use YOLOv8n trained at 1080p resolution for 50 epochs. Both achieve mAP_{50} scores near 0.98 with precision and recall in the 0.95–0.98 range. Connection wires are segmented using Mask2Former [5] (Swin-B backbone) trained on 950/50 train/validation frames using AdamW with cosine decay and standard augmentations, achieving a mean IoU of 0.87.

Topology Reconstruction. Graph connectivity is inferred from geometric relationships between nodes and sockets. Socket–node associations are determined via spatial overlap, while socket–socket connections are inferred by fitting candidate Bézier curves between output and input sockets and measuring their overlap with the wire segmentation mask.



Build Node to Socket and Socket to Socket topology

Combining detected nodes, sockets, inferred edges, and parameter states yields the graph representation (see figure on the right):

$$G_k = (\mathcal{N}_k, \mathcal{S}_k, E_k, \Theta_k), \tag{6}$$

where E_k represents graph connectivity and Θ_k stores node parameters. Each pair (t_k, G_k) therefore represents a temporally grounded snapshot of the evolving material graph.

10.3 Process Summarization

Finally, the sequence of reconstructed graph states is fused with the narration transcript to produce a textual process trace. Given $\{(t_k, G_k)\}$ and the narration transcript τ_v, a language-model summarizer integrates these multimodal signals:

$$p_v = \Phi_{\text{sum}}(\{(t_k, G_k)\}, \tau_v). \tag{7}$$

The summarizer aligns narrated explanations with observed graph edits and produces a compact description of the construction process. Self-refinement prompts ensure that the generated process remains consistent with both the transcript and the reconstructed graph sequence. The resulting trace p_v captures both explicit construction steps and the procedural reasoning expressed by the artist.

Figures 30 and 31 show example outputs before and after summarization. Raw narration transcripts are typically long (often exceeding 60k tokens for a 10 minute tutorial), while the summarized process traces are significantly shorter (typically $\sim$ 8k tokens), making them easier to condition the ProcessSynthesizer during retrieval-time reasoning.

11 Test Prompts used for Evaluation

All experimental evaluations—including text-conditioned generation, image-conditioned generation, and ablation studies—use materials sourced from the BlenderKit dataset. Table 5 summarizes the distribution of test prompts across the 29 BlenderKit material categories used in our benchmark, averaging 3.9 prompts per category. The source materials used in both coarse and fine-grained editing experiments are also drawn from BlenderKit.

For image-conditioned generation, we additionally evaluate on a set of in-the-wild close-up material photographs collected from the internet. The category distribution of these images is also reported in Table 5.

12 Ablations

In addition to the ablations presented in the main paper, we analyze the contribution of the video analysis tools used for process extraction, the effect of summarizer self-refinement, and the role of the DeCompiler in enabling reliable editing.

12.1 Ablation of Video Analysis Tools

The video analysis pipeline converts a tutorial video into a structured process trace through three stages. First, **Video Decomposition** extracts frames and audio, scores frames for informativeness, removes near-duplicates using perceptual embeddings, and produces a time-aligned transcript via ASR. Second, **Graph Reconstruction** parses each selected keyframe using lightweight detectors, segmentors, and OCR to recover the evolving node graph—its nodes, socket connections, and parameter states such as sliders and color ramps. Finally, **Process Summarization** fuses these graph states with the narration using an LLM with self-refinement to produce a concise textual process that captures both explicit editing steps and the expert's underlying rationale.

To assess the role of each stage, we evaluate four ablations of this pipeline and analyze their impact on text-conditioned material generation:

- **Audio-only**: removes all visual analysis and reduces the pipeline to ASR on the narration followed by process summarization.
- **Random Keyframe Sampling**: replaces the scored and deduplicated keyframe selection in video decomposition with uniform random sampling.
- **GPT-based Frame Parsing**: replaces the domain-adapted detectors and segmentors used for graph reconstruction with GPT-5-based vision parsing of each frame.
- **w/o Summarizer Self-Refine**: removes the iterative self-refinement stage in process summarization, forcing the summarizer to compress the raw multimodal trace in a single pass.

Each ablation isolates a single stage while keeping the rest of the pipeline unchanged. The resulting process traces are then used by MaterialApprentice during material generation.

Figures 15–19 illustrates how these components affect generation quality. **Audio-only** performs the worst: without spatial information about the node graph, the model cannot resolve references such as "connect the output of the color ramp to the mix factor," especially when multiple such nodes exist, resulting in severely degenerate graphs. **Random Keyframe Sampling** improves over **Audio-only** by exposing the correct set of nodes, but randomly selected frames rarely capture meaningful structural cues—many contain isolated nodes, occlusions, or uninformative UI states—leading to incorrect or incomplete graph connectivity. **GPT-based Frame Parsing** further improves quality since it analyzes visual frames rather than relying purely on narration, but generic vision parsing often misreads Blender UI elements such as ColorRamp stops, color tabs, and slider values, producing inaccurate graph states that degrade the extracted procedural cues. **w/o Summarizer Self-Refine** starts from the strongest multimodal trace, but a single-pass summary often overlooks subtle design rationale and parameter intent, yielding weaker guidance for generation. In contrast, the full pipeline—combining importance-weighted keyframe selection, domain-adapted graph reconstruction, and iterative process summarization—produces

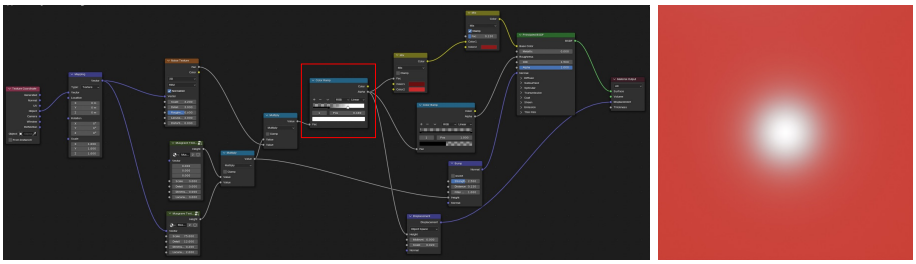


Fig. 15: Audio-only ablation. Without visual parsing, the system relies solely on spoken narration to reconstruct the process, producing an oversimplified and structurally incorrect node graph (left). Lacking spatial and topological cues, multiple semantic roles—color, roughness, bump, and displacement—collapse into a single reused mask, resulting in a flat, low-fidelity material (right). This illustrates that narration alone is insufficient for recovering procedural workflows, as critical graph structure is conveyed visually rather than verbally.

the most faithful process traces and consistently delivers the highest-quality generated materials.

To illustrate the role of each component more concretely, we analyze a representative example: generating a *raw–beef* material. This example is diagnostically useful because high-quality results require (1) multi-scale feature construction, (2) correct node-graph topology, and (3) coherent cross-channel reasoning across color, roughness, bump, and displacement. Even small parsing failures therefore produce large and visually noticeable artifacts. Figures 15–19 show qualitative results for each ablation.

Audio-Only. With only narration available, the system has no access to graph structure or spatial cues present in the tutorial frames. As shown in Figure 15, the reconstructed graph collapses into a minimal structure driven almost entirely by a single reused ColorRamp mask, which is incorrectly repurposed across color, roughness, bump, and displacement. The rendered output becomes a flat two-tone pattern with no fat content, strand flow, or multi-scale variation. Audio alone does not describe the topology, grouping, or numeric structure required for procedural reconstruction.

Random Keyframe Sampling (Figure 16). Using visual frames but selecting them uniformly at random exposes many relevant nodes but rarely captures the structural relationships between them. Important edges and parameter interactions simply never appear in the sampled frames. As shown in Figure 16, the system assembles the correct primitives (Noise, Musgrave, Voronoi) but connects them incorrectly: displacement becomes disconnected from color reasoning, and the intended directional warp collapses into an isotropic bump field. The result resembles a lumpy sausage interior rather than raw beef. This demonstrates that which frames are selected is as important as having frames at all.

GPT-5 Frame Parsing. Replacing the domain-adapted detectors with generic GPT-5 vision parsing yields noisier node states. GPT-5 frequently mis-

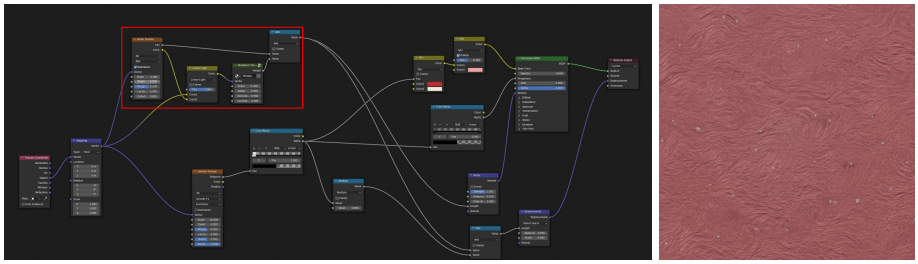


Fig. 16: Random keyframe sampling ablation. Without importance-weighted keyframe selection, the pipeline receives randomly sampled frames that often miss critical graph-level relationships. While many correct primitives (e.g., Noise, Musgrave, Voronoi) are detected, they are assembled with incorrect topology, producing a diffuse, isotropic texture rather than the elongated fiber structure of raw beef. This highlights the importance of selecting informative keyframes for recovering coherent procedural structure.

reads sliders, color ramps, and numeric parameters—particularly in complex nodes such as ColorRamp or Musgrave. As a result (Figure 17), frequency values become exaggerated, color-ramp thresholds collapse to black, and vector outputs are misinterpreted as scalar fields. The resulting material resembles an over-sharpened sponge rather than meat. This highlights the need for domain-adapted parsing capable of reliably extracting Blender-specific parameters.

No Self-Refinement in Process Summarization. When keyframes and graph reconstruction are correct but the summarizer compresses the verbose process trace in a single pass, subtle structural dependencies are lost. As shown in Figure 18, the graph contains the correct families of nodes (fat masks, noise layers, fiber masks) but their relationships become mis-sequenced or collapsed. The rendering resembles ground meat—plausible in isolation but lacking the anisotropic strand-aligned structure of raw beef. Iterative refinement is required to preserve long-range dependencies across channels.

Full Pipeline. Only the complete pipeline—importance-weighted keyframes, domain-adapted graph reconstruction, accurate color and parameter extraction, and iterative process summarization—successfully reconstructs the procedural logic of the tutorial. As shown in Figure 19, the recovered graph captures the intended multi-scale structure: a warped vector field for strand direction, a mid-frequency Musgrave mask for elongated fibers, Voronoi fat specks modulating color and roughness, micro-noise for fine grain, and consistent cross-channel connections tying these elements together. The resulting render exhibits anatomically plausible marbling, strand flow, and shading, demonstrating that all stages of the pipeline are required to recover the full procedural structure of expert workflows.

12.2 Refinement

Refinement plays a key role in improving generation quality in MaterialApprentice. We study two forms of refinement: (1) Self-Refine [18], where an LLM

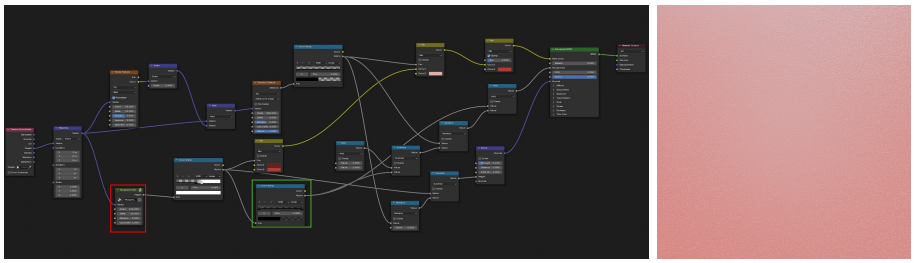


Fig. 17: GPT-5 Frame Parsing Ablation : GPT-5–based frame parsing introduces systematic parameter and color-value misinterpretations. Although the model identifies high-level node types, it frequently misreads slider values, color-ramp stops, and vector–color outputs, leading to incorrect frequency scales and collapsed shading cues. The resulting graph (left) contains plausible structure but incorrect numeric detail, producing an over-sharpened, sponge-like material (right) rather than the intended raw-beef appearance.

critiques and revises its own output, and (2) iterative material-graph refinement using BlenderAlchemy [12]. Motivated by the gains observed from self-refine in the SUMMARIZER, we apply the same strategy to both the EDITOR and COMPILER and evaluate their individual contributions. Figure 20 shows qualitative results across five text-to-material examples. Removing self-refine from either the EDITOR or COMPILER causes a clear drop in fidelity and coherence. Although the raw outputs from the COMPILER are already strong, subsequent iterative refinement further improves realism and prompt alignment, demonstrating complementary benefits between LLM-based self-correction and post-hoc graph optimization.

12.3 Decompiler

The DECOMPILER converts an input material graph g_i into a process trace p_i, allowing the EDITOR to operate in the same modality as the retrieved process traces. Using a small bank of exemplar (graph, process) pairs, it aligns substructures in g_i with analogous patterns in these exemplars to reconstruct a plausible authoring process. Without this step, the EDITOR must edit graphs directly, which leads to unstable edits and semantic drift. Figure 21 shows this effect on the *same editing setup and prompts used in the main paper*: bypassing the DECOMPILER and supplying the initial graph directly to the EDITOR produces noticeably weaker and less controlled edits compared to our full pipeline.

13 Comparison to PhotoMat

We provide qualitative comparisons with PhotoMat [28], a state-of-the-art *non-procedural* texture generation method based on diffusion models. Unlike MATERIALAPPRENTICE, which generates procedural shader graphs, PhotoMat directly

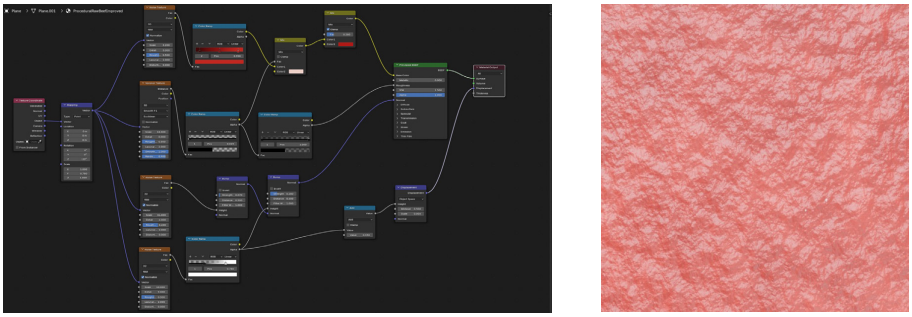


Fig. 18: No summarizer self-refinement ablation. Without the self-refinement loop, the summarizer compresses the multimodal trace too aggressively, dropping key procedural cues and weakening dependencies between graph components. Although most nodes and masks are correctly identified, their functional relationships are reconstructed imprecisely. The resulting material resembles ground or processed meat rather than structured raw beef, illustrating that iterative refinement is important for preserving long-range dependencies across color, roughness, bump, and displacement.

synthesizes texture maps from image inputs. Figure 22 shows representative results on several materials. While PhotoMat often captures coarse color statistics, it frequently struggles to reproduce the structured, hierarchical patterns present in many materials. In contrast, MaterialApprentice produces outputs that better preserve directional structures and characteristic construction patterns (e.g., the layered fibers in *Salmon* or the clustered patterns in *Salamander Skin*), highlighting the advantages of process-based procedural generation.

14 Additional Results

Additional qualitative results for text conditioned generation, image conditioned generation and text based editing are shown in Figures 23, 24, 25 respectively.

15 Limitations

MaterialApprentice is the first system to explicitly leverage tacit expertise in procedural material creation to synthesize new materials, but several limitations remain.

First, expert-made materials often fall into two extremes: highly elaborate graphs (see Fig26) with many interacting layers, or extremely concise graphs that use a few nodes to express rich structure. MaterialApprentice struggles in both regimes. Large graphs introduce too many degrees of freedom for reliable synthesis, while compact, elegant graphs reflect artistic intuition developed over years—intuition that cannot be captured from a small number of demonstrations 26. Second, as shown in Figure 27, while MaterialApprentice created

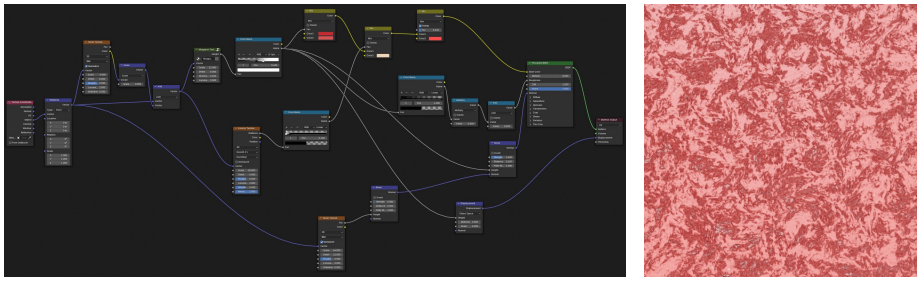


Fig. 19: Full pipeline. Combining importance-weighted keyframe selection, domain-adapted frame parsing, accurate parameter extraction, and iterative summarization enables faithful reconstruction of the procedural logic in the tutorial. The recovered graph captures the intended multi-scale structure—global strand flow, mid-frequency muscle fibers, Voronoi fat marbling, and micro-scale surface grain—with consistent modulation across color, roughness, bump, and displacement. The resulting render exhibits anatomically plausible raw-beef structure, showing that all components of the extraction pipeline are necessary to recover expert procedural workflows.

materials are often structurally close, they sometimes lack adequate parameter tuning which may require some human inputs to achieve desired quality. Lastly, MaterialApprentice depends on a powerful but slow and expensive backbone LLM (GPT-5) for both video understanding and process synthesis in a single forward pass. Future work could explore more efficient multi-agent or modular architectures that decompose the task—e.g., separating video parsing, structural reasoning, and graph generation—allowing these components to be handled by smaller, faster, and more accessible models.

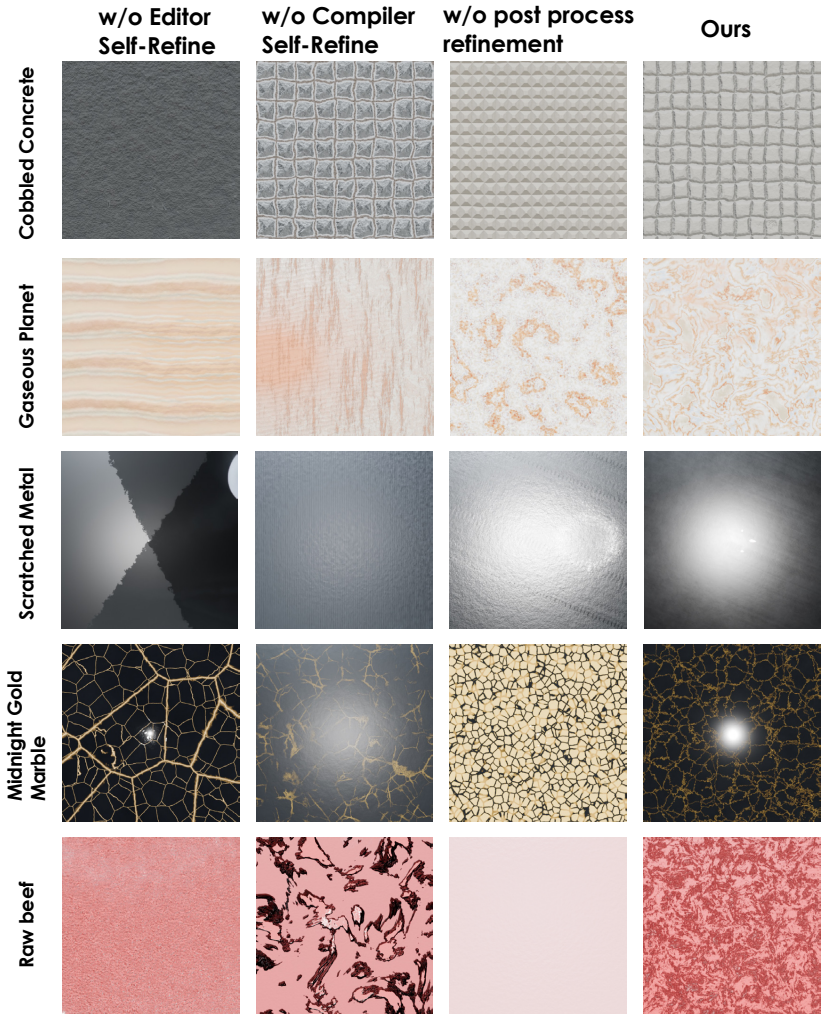


Fig. 20: Ablation of refinement strategies : We evaluate the impact of three refinement components on text-to-material generation: removing self-refinement from the Editor, removing self-refinement from the Compiler, and omitting post-process refinement (BlenderAlchemy-style iterative improvement). Across five representative materials, removing either self-refinement stage leads to noticeable structural and perceptual degradation, while skipping post-process refinement reduces realism and prompt alignment. Our full pipeline (rightmost column), which includes all refinement stages, consistently produces the most faithful and visually coherent materials.

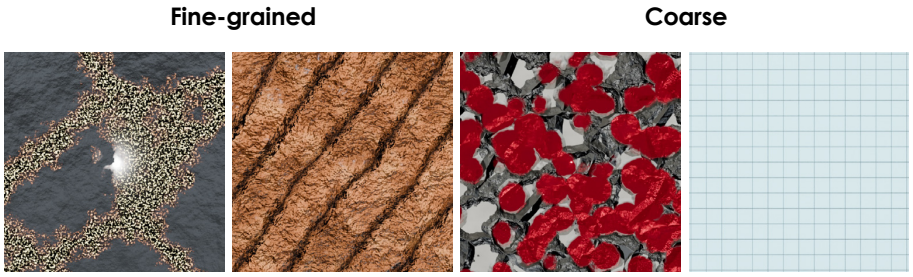


Fig. 21: Ablation of the DeCompiler: Editing results on the same prompts as in the main paper. Without the DeCompiler (results shown here), the editor receives only the raw input graph and produces weak or semantically drifting edits. With the DeCompiler (results shown in main paper Figure 7), converting the graph into a process trace enables stable, aligned, and higher-quality edits.

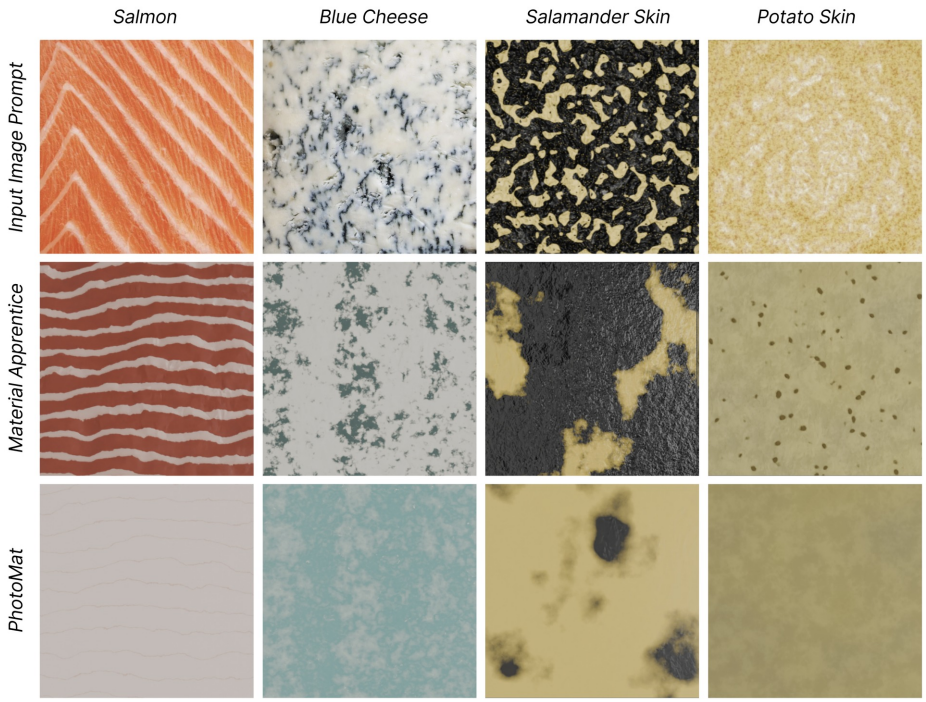


Fig. 22: Qualitative comparison with PhotoMat [28]. Given an input reference image (top row), MaterialApprentice generates procedural materials that better reproduce the characteristic structural patterns of the target materials. In contrast, PhotoMat—despite capturing approximate color statistics—often produces overly smooth or structurally simplified textures due to its purely image-based generation paradigm.

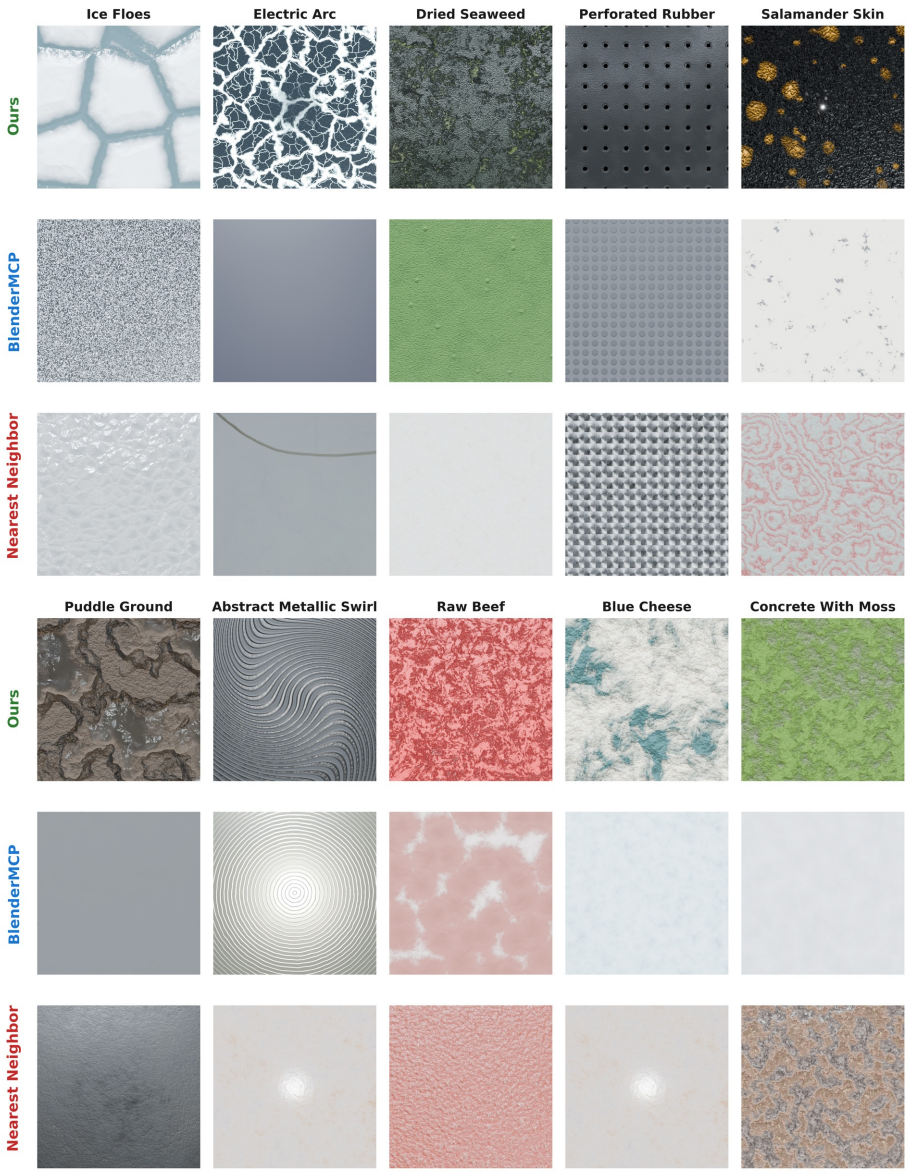


Fig. 23: Example showing the Text to material Generation for 10 prompts.

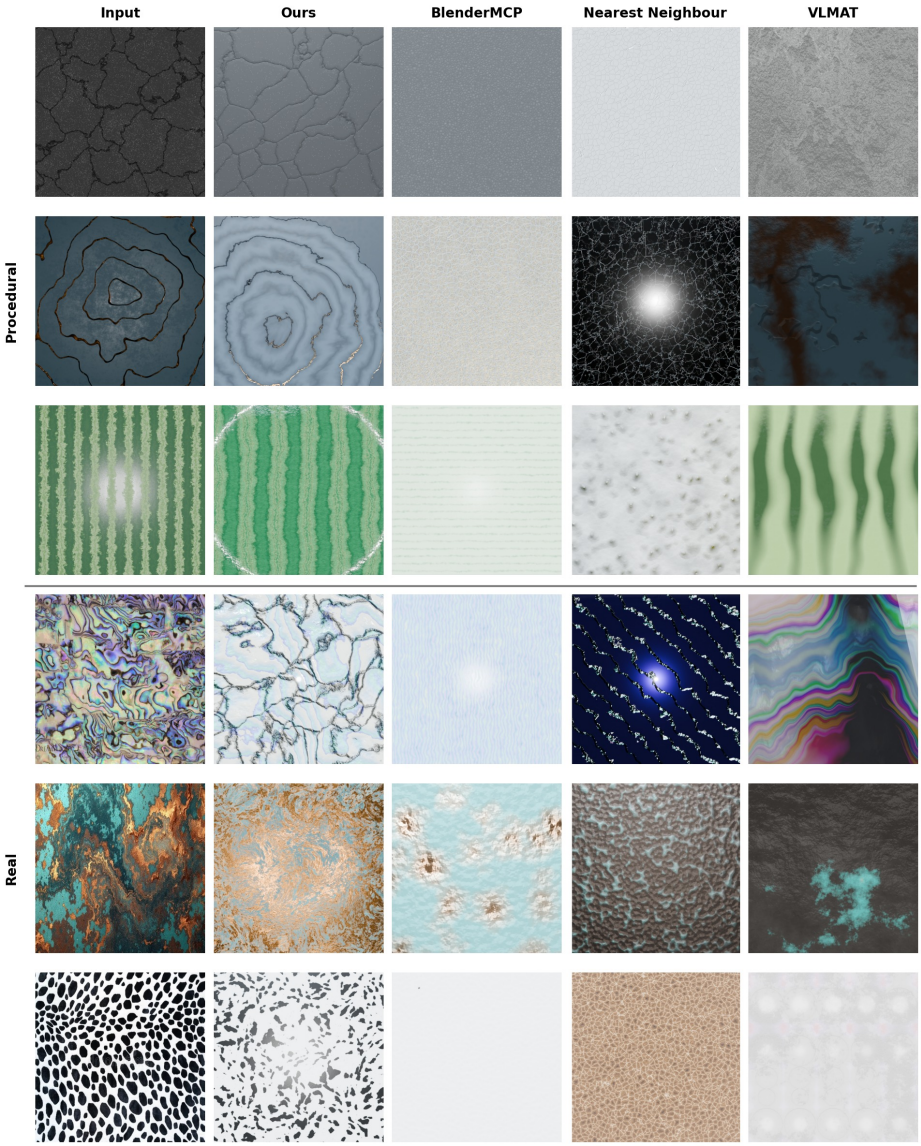


Fig. 24: Image conditioned Material Generation. Images include whethered asphalt, midnight gold marble, watermelon, abalone shell sheet, copper platina, dalmatian skin. Our method better captures micro and macro details specified via the image reference.

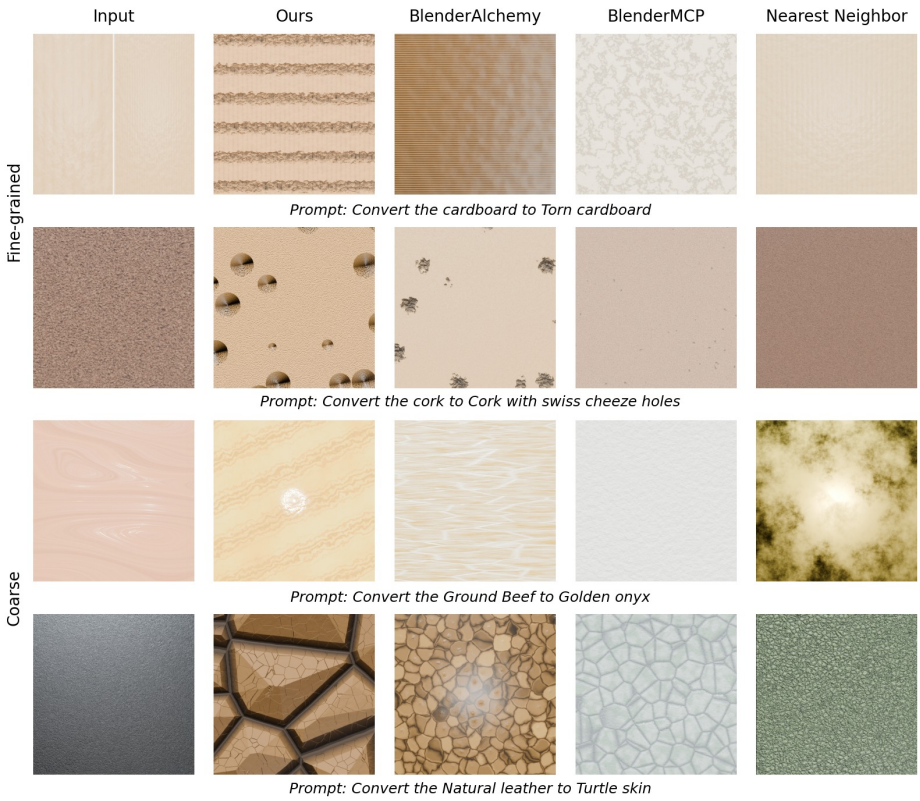


Fig. 25: Fine and Coarse editing on a base material given certain prompts.

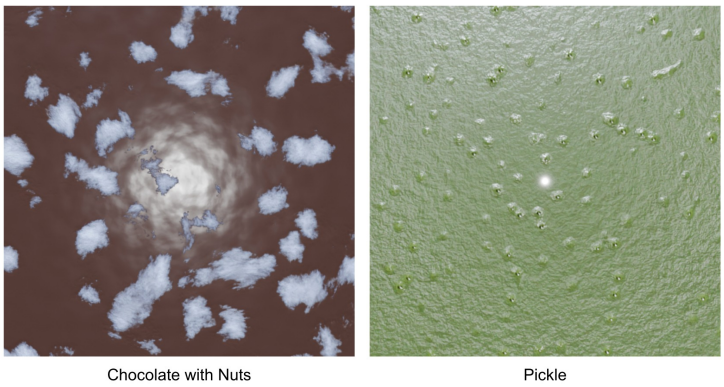


Fig. 26: Limitations : MaterialApprentice stuggles to generate materials requiring more than 25 nodes. Example shown here for text to material generation for chocolate with nuts and pickle, both of which require significant number of shader nodes to express adequate realism.

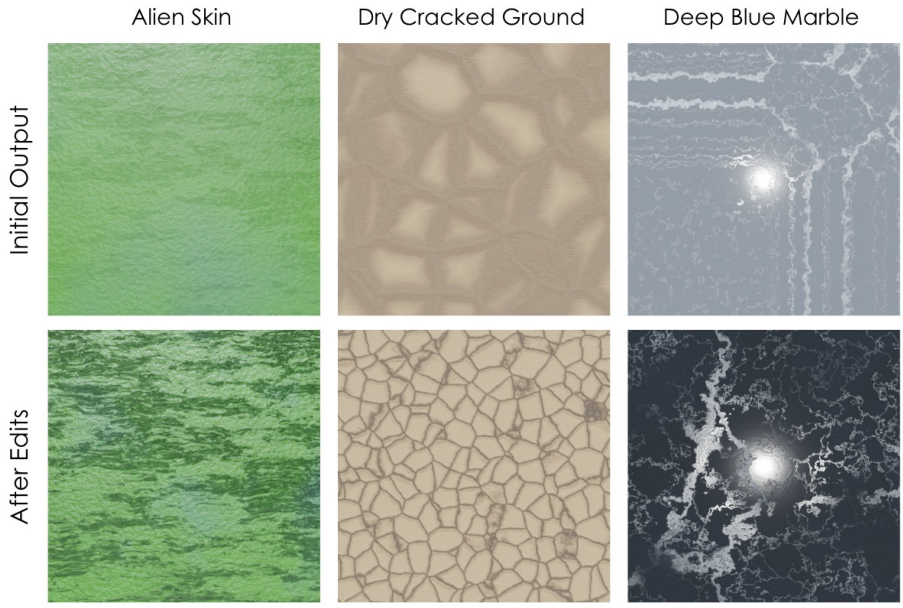


Fig. 27: Limitations and simple refinements. Our method can occasionally produce materials that lack clear structural detail or exhibit imperfect texture organization. The top row shows the initial outputs generated by the pipeline for three example materials (Alien Skin, Dry Cracked Ground, and Deep Blue Marble). While the overall color and coarse appearance are captured, important meso-scale structures such as cracks, veins, or directional patterns may be weak or poorly formed. The bottom row shows the same materials after applying small parameter edits to the generated procedural graph. These lightweight adjustments recover sharper structural features and more realistic appearance, indicating that the generated materials are often close to a correct solution but may require minor refinement.

```
You will be given narration from expert Blender artists. Based on it, you should compare the two processes for creating a material and answer the following questions.
For each question, select one of:
- "Process1"
- "Process2"
Also provide a short justification (1–3 sentences).
Q1 — Implementation-Level Clarity
Which process provides clearer, more explicit step-by-step instructions for constructing the node graph, including node connections and parameter settings?
Q2 — Procedural Strategy Quality
Which process demonstrated a more sophisticated and well-structured procedural design strategy?
Q3 — Outcome Quality
Which process will likely result in a higher-quality material after implementation and editing?
Q4 — Fine-Grained Parameter Control
Which process provides more independent controls and adjustable parameters for modifying specific aspects of the material (such as crack depth, height layering, color masking, and bump detail) without rewriting the overall shader logic?
Q5 — Physical Plausibility
Which process better reflects correct physical reasoning about the material's structure and appearance?
Q6 — Reusable Procedural Techniques
Which process introduces procedural node patterns and techniques (e.g., layered height fields, crack masking, displacement integration) that could be reused as building blocks when constructing a wide range of other procedural materials?
Q7 — Pedagogical Value for Intermediate/Advanced Users
If you were teaching procedural material generation to an intermediate or advanced Blender user, which process would serve as the stronger instructional example by better exposing the underlying procedural reasoning, graph organization, design tradeoffs, and material-specific construction logic?
Return EXACTLY one valid JSON object in this format:
{
  "comparison": {
    "Q1_clarity_readability": {
      "winner": "Process1",
      "reason": ""
    },
...
    "Q7_pedagogical_clarity": {
      "winner": "Process1",
      "reason": ""
    }
  }
}
Output ONLY JSON.
""",
```

Fig. 28: Prompt used to analyze artist narrations and perform the structured forced-choice evaluation across the defined criteria (Q1–Q7).

You will be given two editing process narrations along with the edits that were made in each process. Your task is to estimate the emotional and cognitive state reflected in each process. Focus on four workflow sentiments that reflect the editing experience. For each sentiment, assign one of the following levels: "low", "medium", "high"
Return EXACTLY one valid JSON object in the following format:
{ "process1": {"confusion": "low", "hesitation": "low", "satisfaction": "low","completion": "low"},
"process2": { "confusion": "low", "hesitation": "low", "satisfaction": "low", "completion": "low"} }
Do not include any additional text outside the JSON.

Sentiment Definitions
1. Confusion
Meaning: difficulty understanding the graph structure or what an edit is doing.
Examples of HIGH confusion: "I don't know why this is happening.", "What does this node do?", "I'm not sure where this value comes from."
Examples of LOW confusion: "This controls the displacement.", "This node drives the cracks."

2. Hesitation
Meaning: uncertainty about what edit to perform next.
Examples of HIGH hesitation: "Maybe I should try this.", "Let's see what happens if I change this.", "I'm not sure what to do next."
Examples of LOW hesitation: "Next I'll add a noise texture.", "Now I'll adjust the roughness."

3. Satisfaction
Meaning: positive reaction when the material improves.
Examples of HIGH satisfaction: "That looks better.", "Now the cracks are clearer.", "This is closer to the reference.", "Nice, that fixed it."
Examples of LOW satisfaction: "That didn't help.", "This still looks wrong."

4. Completion
Meaning: the user feels the material is finished or good enough.
Examples of HIGH completion: "This is good enough.", "I think this is close to the reference.", "I'll stop here.", "This looks finished."
Examples of LOW completion: "I still want to tweak more", "This isn't quite right yet"

Important Instructions
- Use the narration as the primary signal.
- You may also consider the edit behavior if narration is minimal.
- Be conservative: if there is little evidence for a sentiment, choose "medium".
- The two processes should be evaluated independently.

Fig. 29: Prompt used for sentiment analysis of the editing narrations, identifying signals such as confusion, hesitation, satisfaction, and completion.

We're going to drag the first tab down.

We're just going to click on it.

And I think they're called swatches, but we're going to take the first one.

I'm going to make it kind of like a greenish color.

I'm going to drag this value down to make it darker.

And maybe it'll make it a bit less saturated, maybe a little bit darker in the value.

Something like this, kind of like an avocado green.

----- Image Frame -----

-- Node Details

Node 0: ShaderNodeValToRGB
Parameters: ["Parameters: {'Color Ramp': [{'Pos': 0.0, 'Color': '#010202'}, {'Pos': 0.89, 'Color': '#283d28'}], 'Mode': 'RGB', 'Interpolation': 'Linear'}"]

Node 1: ShaderNodeBsdfPrincipled
Parameters: ["Parameters: {'Base Color': None, 'Metallic': None, 'Roughness': None, 'IOR': None, 'Alpha': None, 'Normal': None, 'Subsurface': None, 'Specular': None, 'Transmission': None}"]

-- Socket Connections
ShaderNodeValToRGB.Color --> ShaderNodeBsdfPrincipled.Base Color

And then we're going to go plus to create another slider, drag it up.

And this one will go ahead and make it lighter in value.

Fig. 30: Example raw narration transcript extracted from a tutorial video (Dinosaur skin material). The transcript captures the full verbal explanation provided by the artist and forms the initial verbose procedural trace before summarization.

- Create a new material: add Principled BSDF and Material Output. Connect Principled BSDF BSDF → Material Output Surface.

Part 1 – Base color (noisy reptile palette)
1) ColorRamp driving Base Color
- Add a ColorRamp and connect its Color → Principled Base Color.
- Add a Noise Texture and connect its Fac → ColorRamp Fac.
- Add a Texture Coordinate node and connect Generated → Noise Texture Vector. (He tried Object but settled on Generated; feel free to experiment.)
- Noise Texture settings for color:
- Scale: 8.0 (adjust to taste; he tweaks this later)
- Detail: 4.0
- Roughness: 0.6
- Distortion: 0.2 (he initially tried 0.3, then reduced to 0.2)
- Lacunarity: 2.0 (default; fine to leave)
- Set the ColorRamp to a green–brown reptile palette (Linear). Example 4-stop palette (adjust to taste; he shows several variations):
- Stop 1 Pos ~0.00: light sandy, e.g. #d1a592
- Stop 2 Pos ~0.29–0.35: dark brown-green, e.g. #45462b
- Stop 3 Pos ~0.60–0.67: olive green, e.g. #4a5c40
- Stop 4 Pos ~0.85–0.92: deep green, e.g. #283d28

Part 2 – Scales bump (Noise + Voronoi "distance to edge" mixed with Darken)
2) Bump node chain
- Add a Bump node. Connect Bump Normal → Principled Normal. We will feed Bump Height with a mixed mask next.

3) Mix (Darken) to combine two textures for the scale mask
- Add a MixRGB node. Set Blend Type: Darken. Set Factor initially to 0.75 (he later increases it; 0.90 gives a stronger scale look).
- Connect MixRGB Color (Result) → Bump Height.

Fig. 31: Example summarized process trace generated by the Summarizer for the Dinosaur skin material. The summarization condenses the verbose transcript and reconstructed graph states into a textual process trace suitable for conditioning the ProcessSynthesizer.