MATDUET: AGENTIC PROCEDURAL MATERIAL EXTRACTION FROM GENERATIVE VIDEO MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Procedural materials are editable programs that generate surface appearance at arbitrary resolution, but authoring them requires substantial expertise. Generating them automatically is a challenging problem that involves both discrete program design and continuous parameter fitting. We introduce MATDUET, which generates executable procedural shader programs from an image of a material and/or a text prompt. In our full pipeline, a finetuned video generator acts as a "virtual gonioreflectometer", converting the image or text prompt into an 81-frame sequence under prescribed illuminations and viewpoints. From this sequence, we select six frontal relightings and six oblique views that instantiate a plausible target material. MATDUET then alternates discrete and continuous optimization. A coding agent writes an executable Blender program and declares its continuous parameters, which a differentiable renderer fits to the virtual gonioreflectometer relightings while holding the program fixed. The resulting program, its rendered images, matching scores, and visual critique are fed back to the coding agent, allowing fitting residuals to guide subsequent program revisions. The output is resolution-independent Blender source code whose structure and parameters remain available for rendering and editing. Across synthetic materials, authored Blender and Substance materials, and real photographs, MATDUET reduces LPIPS by 9.3–26.1% relative to the strongest baseline on each dataset. Ablations analyze input augmentation, agentic search, differentiable refinement, and test-time scaling. Our code and data will be publicly released. Project page: `https://matduet-agent.github.io/`.

1 INTRODUCTION

Procedural materials represent surface appearance as executable programs. Unlike a fixed stack of texture maps, a procedural program can generate texture detail at an arbitrary resolution, expose parameters for editing, and be reused across assets and renderers. These advantages make procedural materials a powerful representation, but authoring them remains difficult: a high-quality material may combine many patterns, filters, masks, and shading operations whose interactions must be designed and tuned by an expert.

How about generating procedural materials automatically? The key challenge is easiest to see in a single-image setting. Given one image of a material, under known lighting, the goal is to write a program whose render matches the image. This problem combines two qualitatively different kinds of optimization. The program must first choose what to compute; for example, which procedural primitives, layers, and material channels should explain the observed structure. This is a discrete program synthesis problem. Once the program is fixed, its numerical parameters, such as colors, scales, and roughness values, form a continuous fitting problem. Prior work has addressed program generation and fixed-program parameter optimization (Shi et al., 2020; Guerrero et al., 2022; Hu et al., 2023; Li et al., 2025). However, when continuous fitting is applied only after a program has been generated, the remaining error arrives too late to inform the program itself. A missing layer or an incorrect pattern cannot generally be repaired by tuning numerical parameters alone.

Moreover, a single image reveals appearance under only one lighting and one view. Many programs can explain that image while producing very different highlights, view-dependent responses, or surface relief under new conditions. We therefore formulate procedural material reconstruction from

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

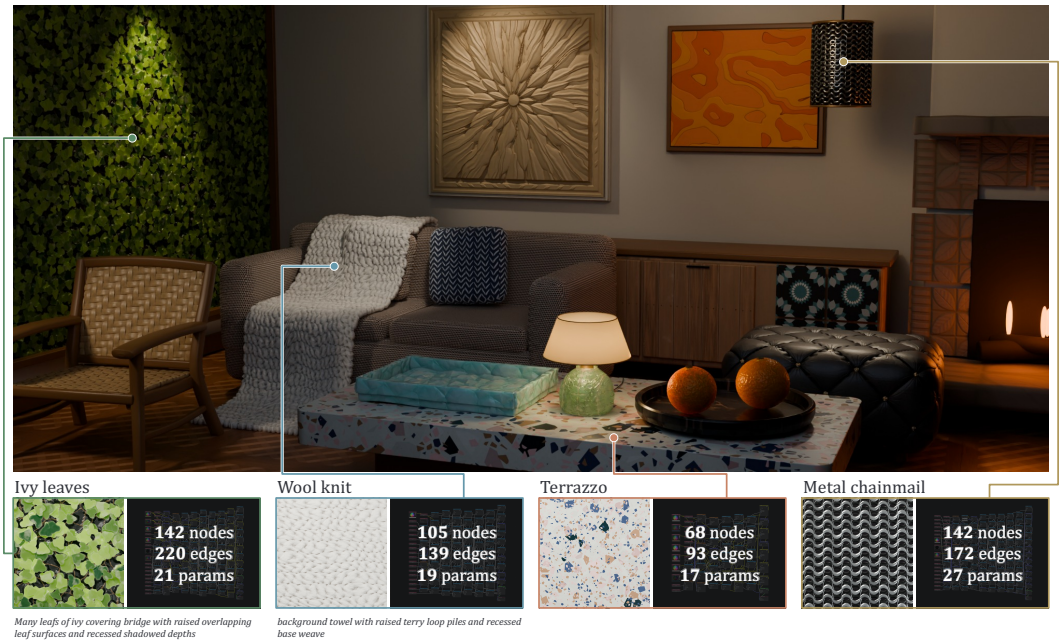


Figure 1: All materials shown in this figure are procedural materials generated by MATDUET. Insets show ivy leaves, wool knit, terrazzo, and metal chainmail alongside their shader node graphs; callouts indicate their placement in the scene. The graph annotations report the numbers of nodes, edges, and adjustable parameters (params). The two examples on the left (ivy leaves and wool knit) are generated from text prompts (text-to-material), while the two on the right (terrazzo and metal chainmail) are generated from reference images (image-to-material).

a variable-size set of target images, with single-image fitting as a special case. In the full pipeline, we adopt the frozen video model and capture trajectory of VideoNeuMat (Xue et al., 2026). We view the model as a *virtual generative gonioreflectometer*: from a text or image prompt, it synthesizes an 81-frame sequence depicting one plausible material instance under prescribed lighting and camera trajectories. We select six frontal lighting views and six oblique views from this sequence. The former support numerical fitting under varied illumination; the latter give the program-synthesis agent evidence about view-dependent appearance, occlusion, and relief. We then turn the material instance expressed by the video into an explicit procedural program.

We introduce MATDUET, which takes a material image and/or a text prompt and produces an executable Blender shader program by coupling these two forms of optimization in an iterative agent-gradient loop. In each round, a coding agent examines the target images and writes an executable Blender material program. The agent determines the program structure and declares the continuous parameters that may be optimized. Holding the program fixed, a differentiable renderer then adjusts those parameters to match the six relighting targets using our Blender-node implementation of MATch's numerical refinement stage (Shi et al., 2020). The original and refined programs are rendered in Blender Cycles, and the better valid candidate is retained. Its source code, renders, matching scores, and visual error analysis are supplied to the next agent round. The agent can preserve a successful construction or revise the program (e.g., by adding a missing component, changing how patterns are composed, or exposing a different set of parameters) before continuous fitting runs again. Consequently, optimization residuals can influence later program design rather than serving only as final adjustment. We view this finite, alternating procedure as a practical form of bilevel optimization: the outer agent searches over programs, while the inner gradient-based solver fits the continuous parameters of each proposed program.

The final output is Blender source code, arguably more valuable than a generated video, a fixed set of texture maps, or an opaque neural representation. It can be executed at a requested resolution, inspected and modified by an artist, and rendered under new conditions. Our experiments evaluate both the general reconstruction method and the benefit of the video-augmented configuration.

Across source-rendered synthetic materials, authored Blender and Substance materials, and real photographs, MATDUET obtains the lowest LPIPS among the compared procedural-material methods. Ablations compare a single target image with generated multi-view and multi-light evidence, isolate the roles of the agent and numerical refinement, and study additional test-time rounds. We further demonstrate direct edits to parameters of the recovered programs.

Our contributions are:

- We formulate procedural material reconstruction from a variable-size set of target images, encompassing conventional single-image fitting and a full pipeline that uses a generated video as a *virtual gonioreflectometer*. The latter converts a text or image prompt into plausible observations under prescribed conditions before producing an explicit procedural asset.
- We introduce an agent-gradient optimization loop that interleaves discrete program synthesis with differentiable fitting of declared continuous parameters. Fitted programs, production renders, and their residual errors become inputs to later agent rounds, allowing numerical fitting to guide subsequent program revisions.
- We evaluate the resulting executable Blender programs across synthetic, artist-authored, real-image, and generated-video inputs, compare against prior procedural-material methods, analyze the system's main components and test-time scaling, and demonstrate resolution-independent rendering and parameter editing.

2 RELATED WORK

Inverse procedural material fitting. Inverse procedural texture methods estimate parameters within prescribed texture families (Qin & Yang, 2002; Lagae et al., 2010; Galerne et al., 2012; Guo et al., 2020). Other approaches use material selection, decomposition, and differentiable texture representations (Hu et al., 2019; 2022b; Baldi et al., 2023; Maesumi et al., 2024). MATch first selects and initializes a graph from a material library, then optimizes its parameters through differentiable evaluation (Shi et al., 2020). Differentiable proxies extend fitting to non-differentiable graph operators (Hu et al., 2022a), and DiffMat v2 supports production graphs with continuous and discrete parameters (Li et al., 2023). ProcMatRL also retrieves a graph before predicting its parameters (Li et al., 2024). Following MATch's differentiable fitting approach, we implement a library of Blender-compatible procedural operators and a differentiable rendering proxy. This allows us to optimize the declared parameters of agent-generated Blender shaders while keeping their program structure fixed.

Procedural material program generation. Earlier procedural program search covered evolved expressions, solid textures, and semantic model retrieval (Sims, 1991; 1993; Wiens & Ross, 2002; Hewgill & Ross, 2004; Dong et al., 2017). MatFormer generates complete graphs (Guerrero et al., 2022). Conditional MatFormer adds text and image conditioning, ranking, and differentiable refinement (Hu et al., 2023), while VLMaterial emits executable Blender material programs from one image (Li et al., 2025). MultiMat conditions node generation on intermediate graph and visual state and uses constrained search, execution repair, and backtracking. Its MultiMat+ variant applies DiffMat after generation (Belouadi et al., 2026). Other approaches combine graph generation or material retrieval with parameter optimization (Lv et al., 2025; Dang et al., 2025). MATDUET interleaves program synthesis and parameter fitting across rounds: the selected shader, its Cycles renders, and matching feedback return to the coding agent, allowing the outcome of numerical fitting to guide later changes to program structure and adjustable parameters.

Generated video and neural material extraction. Semi-procedural priors, neural materials, and procedural training data provide other material representations and generation strategies (Zhou et al., 2023; Raghavan et al., 2025; Yu et al., 2026). VideoNeuMat generates material images under prescribed lighting and camera trajectories and reconstructs a relightable neural material from the resulting sequence (Xue et al., 2026). MATDUET adopts its pretrained video model for input augmentation, expanding a text or image prompt into images under multiple lighting conditions and viewpoints. The video model remains frozen, and the augmented images guide synthesis and iterative refinement of an editable Blender shader.

Graphics agents and procedural abstractions. Language and vision-language models can synthesize, invert, and edit structured 3D scenes and Blender programs (Kulits et al., 2024; Hu et al., 2024; Sun et al., 2025; Wang et al., 2026). BlenderAlchemy searches edits with visual proposals and evaluation (Huang et al., 2025). LL3M combines planning, retrieval, generation, execution, and repair (Lu et al., 2025). VIGA interleaves code generation, rendering, and visual inspection to guide program edits (Yin et al., 2026). Material Apprentice retrieves tutorial expertise for Blender materials (Gupta et al., 2026); BlenderGym benchmarks graphics editing (Gu et al., 2025); and ProcFunc supplies typed, composable functions and graph transpilation for VLM-authored programs (Raistrick et al., 2026). MATDUET combines visual critique with differentiable fitting of semantically meaningful parameters declared by the coding agent. These parameters serve both as user-editable controls and as optimization variables, with fitted values written back into the shader program for subsequent agent revisions.

3 METHOD

3.1 OVERVIEW

MATDUET takes a material image and/or a text prompt as input and produces an executable procedural shader program compatible with Blender. Figure 2 summarizes the four stages. In Stage 1, a frozen video model generates a sequence of images of the material under different lighting conditions and viewpoints (Xue et al., 2026). In Stage 2, a vision-language coding agent (which we term the *writer*) uses these images to synthesize a shader program and declare its adjustable continuous parameters. In Stage 3, differentiable rendering with a perceptual loss refines these parameters while holding the program structure fixed; this is similar to MATch (Shi et al., 2020) but with our new Blender-compatible differentiable operators. In Stage 4, Blender's Cycles renders the candidates for evaluation and selection, and another agent, the *evaluator*, provides scores and visual critique, which return to the writer agent for the next round. Stages 2–4 repeat for a fixed number of rounds, after which the best valid program is exported.

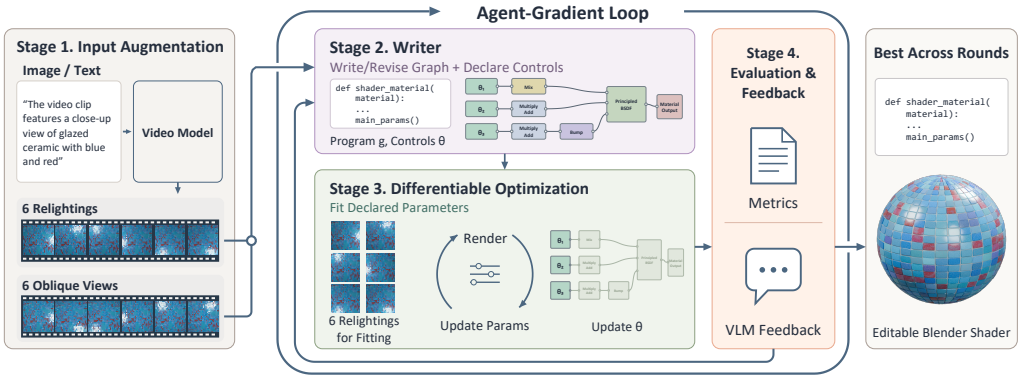


Figure 2: Method overview. A frozen video model augments the input with relightings and oblique views. The writer and differentiable optimizer alternate, with Cycles evaluation and visual feedback guiding the next round.

3.2 STAGE 1: INPUT AUGMENTATION

Our input prompt p is a text description or a reference image of a material. Such an input leaves the material's response to changes in lighting and viewpoint ambiguous: text does not specify its exact appearance, while an image shows only one combination of illumination and view. We therefore augment p with generated images under a prescribed set of lighting conditions and viewpoints.

We use the pretrained video model $\mathcal{V}_\psi$ from VideoNeuMat (Xue et al., 2026), keeping its weights frozen. Given p, it generates an 81-frame video $\mathcal{V}_\psi(p)$ following a fixed two-phase trajectory learned

during training: a small light source completes a circular orbit around the material sample at a fixed elevation, then the light stays fixed and the camera completes a circular orbit at the same elevation.

We use a fixed budget of twelve reference images, selecting six frontal images under different lighting conditions and six oblique views under fixed overhead lighting. Denoting this frame selection by $\mathcal{E}$, the augmented image sets are

$$\begin{aligned} \mathcal{O} &= \mathcal{E}(\mathcal{V}_\psi(p)) = \mathcal{O}_{\text{light}} \cup \mathcal{O}_{\text{view}}, \\ \mathcal{O}_{\text{light}} &= \{(I_i^\ell, s_i^\ell)\}_{i=1}^6, \qquad \mathcal{O}_{\text{view}} = \{(I_j^v, s_j^v)\}_{j=1}^6. \end{aligned} \tag{1}$$

Here, each I is a generated image, and s specifies its corresponding camera and lighting configuration from the prescribed trajectory (Appendix A.3). These twelve images form the augmented input to shader program synthesis.

Generated oblique views can contain more pronounced synthesis errors than frontal relightings. We therefore use the frontal relightings for continuous parameter optimization and candidate scoring, while the oblique views provide auxiliary evidence for the agent to reason about surface relief, parallax, and displacement.

3.3 Stage 2: Shader program synthesis

We use Claude Code with Opus 5 as our vision-language coding agent to construct a procedural shader from the twelve augmented images introduced in Section 3.2. In the first round, the agent creates an initial program. In subsequent rounds, it revises the program selected after the preceding optimization and evaluation stage, using the augmented images and feedback from earlier rounds.

The output is a Python function that constructs a Blender material node graph through a restricted subset of Blender's node API. The available Blender operations include coordinate transformations, procedural textures such as Noise and Voronoi, mathematical operations, color mixing, and surface shading. The agent composes these operations (i.e. it connects their corresponding nodes) to produce shaders that can compute base color, roughness, metallic response, and displacement.

We instruct the agent to match the dominant pattern and scale in the input images before refining tone and detail, and to render and inspect its candidates. In later rounds, it receives numerical diagnostics, a natural-language critique, and a history of earlier scores and suggested fixes, with saved programs available for reuse. Appendix F presents the writer instructions and visual-critic prompts.

We instruct the agent to expose explicit, semantically meaningful controls, such as palette colors, pattern scale, roughness, and relief strength. We represent the program as $P = (g, m, \theta)$, where g is the node graph, θ contains its continuous values, and m identifies the explicitly declared parameters used for both user editing and continuous optimization. These controls are represented by labeled scalar, color, or vector input nodes, with initial values collected in a `main_params()` function. Restricting optimization to these declared parameters enables reliable write-back to the generated free-form Python program, while the graph structure and all undeclared constants remain fixed.

3.4 Stage 3: Continuous parameter optimization

Following MATch's approach to parameter optimization (Shi et al., 2020), we build a differentiable rendering proxy for Blender material programs. The proxy evaluates the agent-generated node graph using our PyTorch (Ansel et al., 2024) library of differentiable procedural and shading operations, allowing image-matching errors to be backpropagated to the declared continuous parameters. Starting from the agent's initial values, we optimize these parameters against the six generated lighting images while keeping the graph structure and the set of optimized parameters fixed. We then write the optimized values back to the Blender program. Appendix B describes the proxy implementation.

The objective combines multiscale texture statistics with a low-resolution color loss, following the texture descriptor used in (Li et al., 2024). We use a Gram-matrix loss to compare overall texture statistics without requiring pixelwise alignment. This is important for procedural materials, whose patterns may be perceptually similar to the reference while differing in their exact spatial arrangement. Pixelwise L1 or L2 losses, as well as spatially aligned measures such as SSIM and LPIPS, can be sensitive to these differences and provide less useful gradients for matching texture appearance.

The low-resolution color term complements the texture loss by anchoring global color and low-frequency variation, reducing color discrepancies that may persist when matching texture statistics alone.

Let $\widetilde{I}_i = \widetilde{\mathcal{R}}(g, \theta; s_i^\ell)$ denote the proxy render under lighting configuration s_i^ℓ. We minimize

$$\mathcal{L}_{\text{inner}}(\theta; g, \mathcal{O}_{\text{light}}) = \frac{1}{6} \sum_{i=1}^{6} \left[\left\| \text{TD}_2(\widetilde{I}_i) - \text{TD}_2(I_i^\ell) \right\|_1 + \lambda_{\text{tone}} \left\| \mathcal{B}_{16}(\widetilde{I}_i) - \mathcal{B}_{16}(I_i^\ell) \right\|_1 \right]. \quad (2)$$

Here TD_2 concatenates spatially normalized Gram matrices from the outputs of all five pooling stages of a fixed, ImageNet-pretrained VGG-19 network, with max pooling replaced by average pooling, at image scales 1, $1/2$, and $1/4$, following neural texture synthesis (Gatys et al., 2015). $\mathcal{B}_{16}$ downsamples an image to 16×16 pixels. The weight λ_{tone} balances these terms, and $\| \cdot \|_1$ denotes the mean absolute difference. Appendix C gives the descriptor and downsampling definitions.

The refined values are written back into the shader source, and the resulting program is rendered in Cycles for evaluation.

3.5 Stage 4: Evaluation and feedback

The evaluator checks each program against the restricted shader interface and verifies that it executes successfully in Blender Cycles and produces a non-degenerate render. For valid programs, we compute mean LPIPS (Zhang et al., 2018) over the six lighting conditions. Let $\mathcal{R}_{\text{prod}}(g, \theta; s)$ denote the Cycles render under camera and lighting configuration s. The lighting score is

$$F_{\text{light}}(g, \theta; \mathcal{O}_{\text{light}}) = \frac{1}{6} \sum_{i=1}^{6} d_{\text{LPIPS}}\big(\mathcal{R}_{\text{prod}}(g, \theta; s_i^\ell), I_i^\ell\big). \quad (3)$$

d_{LPIPS} denotes the frozen LPIPS distance, evaluated with the AlexNet backbone at 256-pixel resolution.

The inner optimizer minimizes a texture-and-color loss using a differentiable proxy, which does not guarantee an improvement in Cycles LPIPS. We therefore accept the refined program only if it passes the validity checks and strictly reduces F_{light} relative to the agent-generated program; otherwise, we retain the original program.

After selection, the evaluator renders the selected program under the six oblique conditions and reports mean view LPIPS as D_{view}. This score is provided as feedback and does not affect candidate selection.

$$D_{\text{view}}(g, \theta; \mathcal{O}_{\text{view}}) = \frac{1}{6} \sum_{j=1}^{6} d_{\text{LPIPS}}\big(\mathcal{R}_{\text{prod}}(g, \theta; s_j^v), I_j^v\big). \quad (4)$$

For visual feedback, a vision-language model compares the generated reference images with renders of the selected shader. It identifies mismatches in pattern, color, and surface relief, highlights aspects that already match, and suggests which changes to prioritize. The evaluator returns the rendered images, lighting LPIPS F_{light}, view LPIPS D_{view}, and the visual critique to guide the coding agent's revisions in the next round.

After the final round, we export the valid shader with the lowest F_{light} across all rounds. This preserves the best result even if a later revision performs worse.

4 Experiments

Datasets. Following VLMaterial (Li et al., 2025), we evaluate on real photographs (Real images), Substance procedural materials, and Blender procedural materials, with 64, 64, and 44 cases, respectively. We additionally evaluate on 89 materials from MatSynth (Vecchio & Deschaintre, 2024). We also use 112 generated material sequences from VideoNeuMat (Xue et al., 2026), primarily for qualitative analysis.

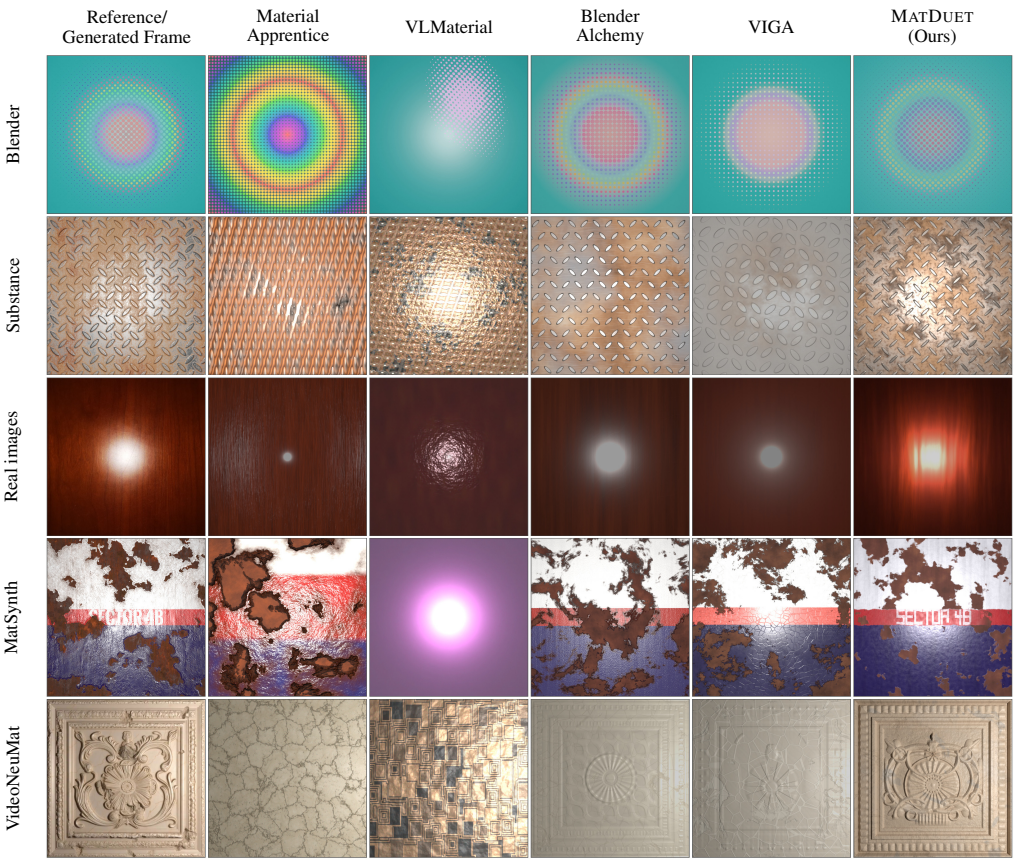


Figure 3: Qualitative comparisons across five datasets. The first column shows ground-truth (GT) references in the first four rows and a VideoNeuMat-generated frame in the last row; the remaining columns show rendered procedural materials. Material Apprentice and VLMaterial miss the Substance tread pattern; BlenderAlchemy and VIGA reproduce recognizable patterns but differ in scale, color, or surface appearance. MATDUET captures painted-metal bands and rust, raised tread, and ornamental relief.

Inputs and evaluation targets. Our pipeline fits generated relightings and views from one canonical image. MatSynth-I2V, Blender, and Substance are scored against source-material renders; Real images against the input photograph. Single input fits only the canonical image and uses the same MatSynth source-GT evaluation. Baselines use their released single-image or image-plus-text interfaces. For datasets with ground-truth materials, we report metrics averaged over six lighting conditions and six viewpoints. For real images, we use the single input photograph as the reference.

Baselines. We compare with Material Apprentice (Gupta et al., 2026) and VLMaterial (Li et al., 2025) for material generation, and BlenderAlchemy (Huang et al., 2025) and VIGA (Yin et al., 2026) for visual program editing. These baselines provide public implementations and pretrained weights. We evaluate their executable, editable Blender programs in the same Cycles pipeline. For a fair comparison, we use Opus 5 with the reasoning effort set to `xhigh` for all agent-based methods. We configure a maximum context window of 1M tokens and increase output-token limits to accommodate extended reasoning and code generation. VLMaterial uses its released fine-tuned checkpoint. We use Harbor (Harbor Framework Team, 2026) to manage the containerized execution of MATDUET; Appendix A.1 gives the execution settings.

Evaluation metrics. To measure perceptual similarity between images rendered from generated material programs and their references, we use five metrics: LPIPS, DreamSim, style loss, sliced Wasserstein distance (SWD), and CLIP cosine similarity. LPIPS (Zhang et al., 2018) compares spatially aligned deep features, while DreamSim (Fu et al., 2023) captures human judgments of

color, layout, and semantic similarity. Following VLMaterial (Li et al., 2025), we compute style loss as the L_1 difference between VGG feature Gram matrices (Gatys et al., 2015) plus the L_1 difference between 16×16 downsampled images, weighted by 0.1. SWD (Bonneel et al., 2015) compares distributions of local RGB patches by averaging one-dimensional Wasserstein distances over random projections. CLIP cosine similarity (Radford et al., 2021) measures semantic resemblance between the rendered and reference images in a shared image-embedding space.

We additionally use a VLM judge inspired by LLM-as-a-Verifier (Kwok et al., 2026) to assess overall material appearance, considering color, pattern, surface detail, and response to lighting. We average three sampled ratings per material. Appendix D.1 specifies the VLM judge configuration and prompt.

Dataset	Method	LPIPS↓	DreamSim↓	Style loss↓	SWD↓	CLIP↑	VLM judge↑
Source-rendered GT							
MatSynth	Material Apprentice	0.5271	0.3200	0.0366	0.0576	0.8914	7.02
	VLMaterial	0.5528	0.4074	0.0262	0.0512	0.8685	4.73
	BlenderAlchemy	0.4323	0.2384	0.0250	0.0366	0.9123	8.45
	VIGA	0.4323	0.2481	0.0237	0.0381	0.9087	8.00
	MATDUET (Ours)	0.3195	0.1718	0.0186	0.0249	0.9274	9.87
Blender	Material Apprentice	0.5129	0.2903	0.0354	0.0619	0.9002	7.36
	VLMaterial	0.3900	0.2216	0.0203	0.0394	0.9269	9.02
	BlenderAlchemy	0.4146	0.2246	0.0240	0.0499	0.9296	8.44
	VIGA	0.4204	0.2439	0.0258	0.0559	0.9196	7.58
	MATDUET (Ours)	0.3399	0.1897	0.0208	0.0403	0.9303	9.64
Substance	Material Apprentice	0.4884	0.2879	0.0260	0.0439	0.9084	7.24
	VLMaterial	0.4841	0.3132	0.0181	0.0407	0.8947	6.60
	BlenderAlchemy	0.4086	0.2328	0.0204	0.0422	0.9204	8.59
	VIGA	0.4115	0.2488	0.0201	0.0480	0.9146	8.24
	MATDUET (Ours)	0.3234	0.1815	0.0171	0.0378	0.9319	9.51
Real Photo Input							
Real images	Material Apprentice	0.5540	0.3737	0.0289	0.0615	0.8453	7.47
	VLMaterial	0.5187	0.3667	0.0211	0.0532	0.8381	6.42
	BlenderAlchemy	0.4533	0.2683	0.0232	0.0488	0.8774	9.88
	VIGA	0.4625	0.2878	0.0227	0.0542	0.8682	9.33
	MATDUET (Ours)	0.4110	0.2569	0.0216	0.0482	0.8771	9.32

Table 1: Comparison with procedural material baselines. The best, second-best, and third-best values within each dataset are highlighted.

Results. We report quantitative comparisons in Table 1 and qualitative examples in Figure 3. MATDUET achieves the lowest LPIPS and DreamSim on all four test datasets, reducing LPIPS by 9.3–26.1% relative to the strongest baseline on each dataset. It leads all six reported metrics on MatSynth and Substance and receives the highest VLM judge scores on all three source-rendered datasets, while for Blender and real photographs, MATDUET performs on par with the strongest baselines.

Figure 4 illustrates additional generated materials, including ornamental relief and woven patterns, rendered under varied lighting. Further examples appear in the appendix.

Method	LPIPS↓	DreamSim↓	Style loss↓	SWD↓	CLIP↑
Single input	0.3563	0.1977	0.0231	0.0625	0.9197
Pure-Agent	0.3759	0.1687	0.0191	0.0264	0.9280
No diff-opt	0.3696	0.1876	0.0197	0.0309	0.9259
Post-opt	0.3570	0.1837	0.0192	0.0282	0.9235
MATDUET (Full)	0.3195	0.1718	0.0186	0.0249	0.9274

Table 2: Ablation on MatSynth-I2V. The best, second-best, and third-best values are highlighted.

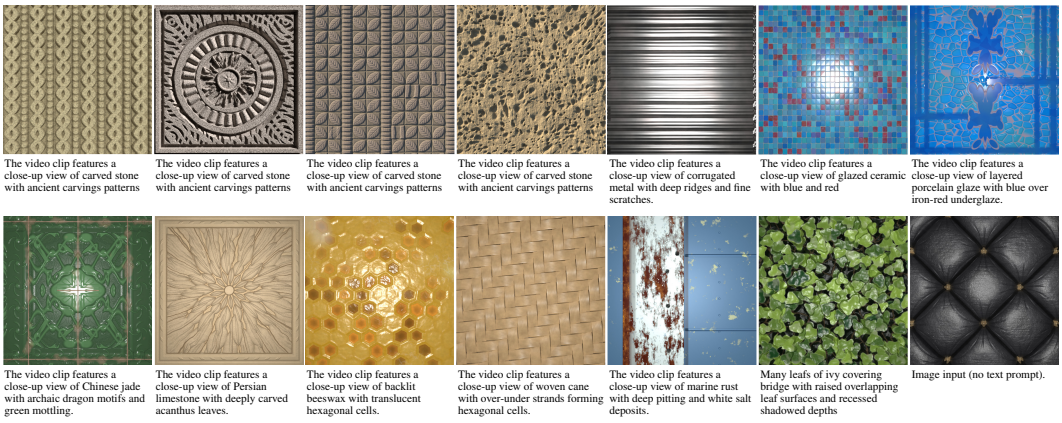


Figure 4: Selected MATDUET materials under varied lighting. Video-generation prompts are shortened where needed; the final example uses an image input.

Ablation experiments. In Table 2 and Figure 5, we evaluate video augmentation and the interaction between agent search and differentiable optimization. To match the full method's number of agent rounds and optimization runs, Post-opt optimizes the shader produced in every round of No diff-opt after agent search finishes, then selects the best shader from all original and optimized versions across rounds. Interleaving agent revisions with optimization outperforms this sequential strategy on all five metrics. This suggests that optimization helps guide subsequent agent revisions, in addition to improving individual programs. The full pipeline also outperforms Single input, supporting the benefit of video-based observation augmentation. Pure-Agent provides a reference for the base model's capabilities in an uninterrupted coding-agent session, while No diff-opt retains our round-based loop without parameter optimization. Appendix D.3 provides additional ablations and examples.



Figure 5: Examples from ablations on MatSynth-I2V (top) and VideoNeuMat (bottom).

5 CONCLUSION

We presented MATDUET, which produces editable Blender shader programs from text or image prompts by combining video-based input augmentation with interleaved program synthesis and differentiable parameter fitting. Our ablations show benefits from both input augmentation and allowing fitted programs to inform subsequent agent revisions. While MATDUET reconstructs a range of materials, complex patterns remain challenging, and some reconstructions require large shader graphs. Future directions include incorporating bitmap textures into procedural programs to expand their expressive range and extending the framework to procedural modeling of 3D objects.

AI USE STATEMENT

We used generative AI tools to assist with implementing the differentiable renderer and with manuscript drafting and language editing. The authors manually inspected and checked the AI-assisted renderer implementation and its outputs, and tested it on a large collection of examples to validate its correctness. AI-assisted manuscript text was reviewed and revised by the authors. Model calls that form part of the research method and their configurations are described in Section 3 and Appendices A.1 and D.1. The authors take responsibility for the final implementation, manuscript, experimental results, and conclusions.

ETHICS STATEMENT

This work studies procedural material extraction from generated videos and involves neither human-subject research nor sensitive personal data. We have identified no ethical issues requiring specific discussion.

REPRODUCIBILITY STATEMENT

We describe the system workflow in Section 3, the experimental setup and evaluation metrics in Section 4, the VLM judging procedure in Appendix D.1, and the execution configuration and shader interface in Appendix A. Details of the differentiable renderer, parameter optimization, and agent prompts appear in Appendices B, C, and F, respectively. The anonymous project page provides additional visual results and downloadable material files. Upon paper acceptance, we will release the complete implementation as open source, including the differentiable renderer, differentiable optimization modules, and agent workflow orchestration code. For experiments using external model APIs, model configurations are reported in Appendices A.1 and D.1; subsequent service updates and stochastic generation may lead to differences across repeated runs.

REFERENCES

Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, pp. 929–947, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703850. doi: 10.1145/3620665.3640366. URL https://doi.org/10.1145/3620665.3640366.

Guillaume Baldi, Rémi Allègre, and Jean-Michel Dischler. Differentiable point process texture basis functions for inverse procedural modeling of cellular stochastic structures. *Computers & Graphics*, 112:116–131, May 2023. ISSN 0097-8493. doi: 10.1016/j.cag.2023.04.004. URL http://dx.doi.org/10.1016/j.cag.2023.04.004.

Jonas Belouadi, Tamy Boubekeur, and Adrien Kaiser. Multimat: Multimodal program synthesis for procedural materials using large multimodal models, 2026. URL https://arxiv.org/abs/2509.22151.

Nicolas Bonneel, Julien Rabin, Gabriel Peyré, and Hanspeter Pfister. Sliced and radon wasserstein barycenters of measures. *Journal of Mathematical Imaging and Vision*, 51(1):22–45, Jan 2015. ISSN 1573-7683. doi: 10.1007/s10851-014-0506-3. URL https://doi.org/10.1007/s10851-014-0506-3.

Ziqiang Dang, Wenqi Dong, Zesong Yang, Bangbang Yang, Liang Li, Yuewen Ma, and Zhaopeng Cui. Texpro: Text-guided pbr texturing with procedural material modeling. *Computational Vi-*

sual Media, 11(4):745–761, 2025. doi: 10.26599/CVM.2025.9450489. URL https://www.sciopen.com/article/10.26599/CVM.2025.9450489.

Junyu Dong, Lina Wang, Jun Liu, and Xin Sun. A procedural texture generation framework based on semantic descriptions, 2017. URL https://arxiv.org/abs/1704.04141.

Stephanie Fu, Netanel Tamir, Shobhita Sundaram, Lucy Chai, Richard Zhang, Tali Dekel, and Phillip Isola. Dreamsim: Learning new dimensions of human visual similarity using synthetic data. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 50742–50768. Curran Associates, Inc., 2023. doi: 10.52202/075280-2208. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/9f09f316a3eaf59d9ced5ffaefe97e0f-Paper-Conference.pdf.

Bruno Galerne, Ares Lagae, Sylvain Lefebvre, and George Drettakis. Gabor noise by example. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2012)*, 31(4):73:1–73:9, July 2012. doi: 10.1145/2185520.2335424.

Leon Gatys, Alexander S Ecker, and Matthias Bethge. Texture synthesis using convolutional neural networks. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015. URL https://proceedings.neurips.cc/paper_files/paper/2015/file/a5e00132373a7031000fd987a3c9f87b-Paper.pdf.

Yunqi Gu, Ian Huang, Jihyeon Je, Guandao Yang, and Leonidas Guibas. Blendergym: Benchmarking foundational model systems for graphics editing. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 18574–18583, June 2025.

Paul Guerrero, Milos Hasan, Kalyan Sunkavalli, Radomir Mech, Tamy Boubekeur, and Niloy Mitra. Matformer: A generative model for procedural materials. *ACM Trans. Graph.*, 41(4), 2022. doi: 10.1145/3528223.3530173.

Yu Guo, Miloš Hašan, Lingqi Yan, and Shuang Zhao. A bayesian inference framework for procedural material parameter estimation. *Computer Graphics Forum*, 39(7), 2020.

Kunal Gupta, Gaurav Joshi, Yen-Ru Chen, Seemandhar Jain, Ishit Mehta, and Manmohan Chandraker. Reflecting process expertise in procedural material generation. In *European Conference on Computer Vision (ECCV)*, 2026.

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments, 2026. URL https://doi.org/10.5281/zenodo.20953922.

Adam Hewgill and Brian J Ross. Procedural 3d texture synthesis using genetic programming. *Computers & Graphics*, 28(4):569–584, August 2004. ISSN 0097-8493. doi: 10.1016/j.cag.2004.04.012. URL http://dx.doi.org/10.1016/j.cag.2004.04.012.

Yiwei Hu, Julie Dorsey, and Holly Rushmeier. A novel framework for inverse procedural texture modeling. *ACM Transactions on Graphics*, 38(6):1–14, November 2019. ISSN 1557-7368. doi: 10.1145/3355089.3356516. URL http://dx.doi.org/10.1145/3355089.3356516.

Yiwei Hu, Paul Guerrero, Milos Hasan, Holly Rushmeier, and Valentin Deschaintre. Node graph optimization using differentiable proxies. In *ACM SIGGRAPH 2022 Conference Proceedings*, SIGGRAPH '22, New York, NY, USA, 2022a. Association for Computing Machinery. ISBN 9781450393379. doi: 10.1145/3528233.3530733. URL https://doi.org/10.1145/3528233.3530733.

Yiwei Hu, Chengan He, Valentin Deschaintre, Julie Dorsey, and Holly Rushmeier. An inverse procedural modeling pipeline for svbrdf maps. *ACM Transactions on Graphics (TOG)*, 41(2):1–17, 2022b.

Yiwei Hu, Paul Guerrero, Milos Hasan, Holly Rushmeier, and Valentin Deschaintre. Generating Procedural Materials from Text or Image Prompts. In *ACM SIGGRAPH 2023 Conference Proceedings*, 2023.

Ziniu Hu, Ahmet Iscen, Aashi Jain, Thomas Kipf, Yisong Yue, David A Ross, Cordelia Schmid, and Alireza Fathi. SceneCraft: An LLM agent for synthesizing 3D scenes as blender code. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 19252–19282. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/hu24g.html.

Ian Huang, Guandao Yang, and Leonidas Guibas. Blenderalchemy: Editing 3d graphics with vision-language models. In Aleš Leonardis, Elisa Ricci, Stefan Roth, Olga Russakovsky, Torsten Sattler, and Gül Varol (eds.), *Computer Vision – ECCV 2024*, pp. 297–314, Cham, 2025. Springer Nature Switzerland. ISBN 978-3-031-73024-5.

Peter Kulits, Haiwen Feng, Weiyang Liu, Victoria Fernandez Abrevaya, and Michael J. Black. Re-thinking inverse graphics with large language models. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL https://openreview.net/forum?id=u0eiu1MTS7.

Jacky Kwok, Shulu Li, Pranav Atreya, Yuejiang Liu, Yixing Jiang, Chelsea Finn, Marco Pavone, Ion Stoica, and Azalia Mirhoseini. Llm-as-a-verifier: A general-purpose verification framework, 2026. URL https://arxiv.org/abs/2607.05391.

Ares Lagae, Peter Vangorp, Toon Lenaerts, and Philip Dutré. Procedural isotropic stochastic textures by example. *Computers & Graphics*, 34(4):312–321, 2010. ISSN 0097-8493. doi: https://doi.org/10.1016/j.cag.2010.05.004. URL https://www.sciencedirect.com/science/article/pii/S0097849310000713. Procedural Methods in Computer Graphics Illustrative Visualization.

Beichen Li, Liang Shi, and Wojciech Matusik. End-to-end procedural material capture with proxy-free mixed-integer optimization. *ACM Transactions on Graphics (TOG)*, 42(4):1–15, 2023.

Beichen Li, Yiwei Hu, Paul Guerrero, Milos Hasan, Liang Shi, Valentin Deschaintre, and Wojciech Matusik. Procedural material generation with reinforcement learning. *ACM Transactions on Graphics (TOG)*, 43(6):1–14, 2024.

Beichen Li, Rundi Wu, Armando Solar-Lezama, Changxi Zheng, Liang Shi, Bernd Bickel, and Wojciech Matusik. Vlmaterial: Procedural material generation with large vision-language models. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 33790–33810, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/5393eaf0b723278738788520ec54e4b4-Paper-Conference.pdf.

Sining Lu, Guan Chen, Nam Anh Dinh, Itai Lang, Ari Holtzman, and Rana Hanocka. Ll3m: Large language 3d modelers, 2025. URL https://arxiv.org/abs/2508.08228.

Xiaoyu Lv, Zizhao Wu, Jiamin Xu, Xiaoling Gu, Ming Zeng, and Weiwei Xu. Interpretable procedural material graph generation via diffusion models from reference images. *The Visual Computer*, 41(13):11195–11205, Oct 2025. ISSN 1432-2315. doi: 10.1007/s00371-025-04096-0. URL https://doi.org/10.1007/s00371-025-04096-0.

Arman Maesumi, Dylan Hu, Krishi Saripalli, Vladimir Kim, Matthew Fisher, Soeren Pirk, and Daniel Ritchie. One noise to rule them all: Learning a unified model of spatially-varying noise patterns. *ACM Trans. Graph.*, 43(4), July 2024. ISSN 0730-0301. doi: 10.1145/3658195. URL https://doi.org/10.1145/3658195.

Xuejie Qin and Yee-Hong Yang. Estimating parameters for procedural texturing by genetic algorithms. *Graphical Models*, 64(1):19–39, January 2002. ISSN 1524-0703. doi: 10.1006/gmod.2002.0565. URL http://dx.doi.org/10.1006/gmod.2002.0565.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision. In Marina Meila and Tong Zhang (eds.), *Proceedings of the 38th International Conference on Machine*

Learning, volume 139 of *Proceedings of Machine Learning Research*, pp. 8748–8763. PMLR, 18–24 Jul 2021. URL https://proceedings.mlr.press/v139/radford21a.html.

Nithin Raghavan, Krishna Mullia, Alexander Trevithick, Fujun Luan, Miloš Hašan, and Ravi Ramamoorthi. Generative neural materials. In *Proceedings of the Special Interest Group on Computer Graphics and Interactive Techniques Conference Conference Papers*, SIGGRAPH Conference Papers '25, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400715402. doi: 10.1145/3721238.3730746. URL https://doi.org/10.1145/3721238.3730746.

Alexander Raistrick, Karhan Kayan, Jack Nugent, David Yan, Lingjie Mei, Meenal Parakh, Hongyu Wen, Dylan Li, Yiming Zuo, Erich Liang, and Jia Deng. Procfunc: Function-oriented abstractions for procedural 3d generation in python, 2026. URL https://arxiv.org/abs/2604.26943.

Liang Shi, Beichen Li, Miloš Hašan, Kalyan Sunkavalli, Tamy Boubekeur, Radomir Mech, and Wojciech Matusik. Match: Differentiable material graphs for procedural material capture. *ACM Transactions on Graphics (TOG)*, 39(6):1–15, 2020.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015. URL https://arxiv.org/abs/1409.1556.

Karl Sims. Artificial evolution for computer graphics. *ACM SIGGRAPH Computer Graphics*, 25 (4):319–328, July 1991. ISSN 0097-8930. doi: 10.1145/127719.122752. URL http://dx.doi.org/10.1145/127719.122752.

Karl Sims. Interactive evolution of equations for procedural models. *The Visual Computer*, 9(8): 466–476, Aug 1993. ISSN 1432-2315. doi: 10.1007/BF01888721. URL https://doi.org/10.1007/BF01888721.

Chunyi Sun, Junlin Han, Weijian Deng, Xinlong Wang, Zishan Qin, and Stephen Gould. 3d-gpt: Procedural 3d modeling with large language models. In *2025 International Conference on 3D Vision (3DV)*, pp. 1253–1263. IEEE, March 2025. doi: 10.1109/3dv66043.2025.00119. URL http://dx.doi.org/10.1109/3DV66043.2025.00119.

Giuseppe Vecchio and Valentin Deschaintre. Matsynth: A modern pbr materials dataset. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 22109–22118, June 2024.

Hao Wang, Wenhui Zhu, Shao Tang, Zhipeng Wang, Xuanzhao Dong, Xin Li, Xiwen Chen, Ashish Bastola, Xinhao Huang, Yalin Wang, and Abolfazl Razi. Ezblender: Efficient 3d editing with plan-and-react agent. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV) Workshops*, pp. 1343–1352, March 2026.

Andrea L. Wiens and Brian J. Ross. Gentropy: evolving 2d textures. *Computers & Graphics*, 26(1):75–88, February 2002. ISSN 0097-8493. doi: 10.1016/s0097-8493(01)00159-5. URL http://dx.doi.org/10.1016/S0097-8493(01)00159-5.

Bowen Xue, Saeed Hadadan, Zheng Zeng, Fabrice Rousselle, Zahra Montazeri, and Milos Hasan. Videoneumat: Neural material extraction from generative video models. In *ACM SIGGRAPH 2026 Conference Papers*, 2026.

Shaofeng Yin, Jiaxin Ge, Zora Zhiruo Wang, Chenyang Wang, Xiuyu Li, Michael J. Black, Trevor Darrell, Angjoo Kanazawa, and Haiwen Feng. Vision-as-inverse-graphics agent via interleaved multimodal reasoning, 2026. URL https://arxiv.org/abs/2601.11109.

Yunchen Yu, Jacob Munkberg, Jon Hasselgren, Chris Cummings, Steve Marschner, and Andrea Weidlich. Toward richer material generation via procedural data enhancement. In *Proceedings of the Special Interest Group on Computer Graphics and Interactive Techniques Conference Conference Papers*, SIGGRAPH Conference Papers '26, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400725548. doi: 10.1145/3799902.3811210. URL https://doi.org/10.1145/3799902.3811210.

Richard Zhang, Phillip Isola, Alexei A. Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

Xilong Zhou, Miloš Hašan, Valentin Deschaintre, Paul Guerrero, Kalyan Sunkavalli, and Nima Khademi Kalantari. A semi-procedural convolutional material prior. In *Computer Graphics Forum*. Wiley Online Library, 2023.

A Implementation Details

This section describes the execution loop, shader interface, and VideoNeuMat camera and lighting configuration. Renderer internals and parameter fitting are separated into Appendices B and C; agent instructions appear in Appendix F.

A.1 Execution and configuration

Our standard configuration uses Claude Code with Opus 5 at `xhigh` reasoning effort, running in Docker containers managed by Harbor (Harbor Framework Team, 2026). We allocate six outer rounds, with a 45-minute writer timeout and a 50-minute refinement-and-evaluation timeout per round. Cycles rendering and differentiable parameter optimization run on a single NVIDIA RTX PRO 5000 GPU. Each round starts a fresh writer context with the reference images and, after the first round, the current shader and preceding evaluation report; saved programs and an append-only history carry state between rounds. A Gemini 3.1 Pro visual critic compares each of the six lighting pairs and six oblique-view pairs separately, then merges the per-image reports into feedback for the next round. Our implementation uses Blender 5.2.0 LTS. Cycles evaluation uses 512×512 images and 64 samples per pixel, with adaptive subdivision and both bump and true displacement enabled.

A.2 Shader-program interface

The output language is Python using a restricted subset of Blender's node API. Each program defines `shader_material(material)`, enables material nodes, creates nodes with `nodes.new("ShaderNode...")`, sets node properties and socket values, and connects outputs to inputs with `links.new(...)`. A Material Output node must be wired explicitly. The program may import only `bpy` and `math`; image-texture nodes, external file reads, and access through `bpy.ops`, `bpy.context`, or `bpy.data` are prohibited. Camera, illumination, mesh subdivision, and rendering are supplied separately by the rendering setup.

Table 3 lists the complete node-type set from the saved task's validator. The differentiable proxy implements all 42 listed node types, with the shading and geometry approximations described in Appendix B.

Function	Node types (the common `ShaderNode` prefix is omitted)
Coordinates and geometry	TexCoord, Mapping, NewGeometry, ObjectInfo
Procedural textures	TexChecker, TexNoise, TexVoronoi, TexWave, TexBrick, TexGradient, TexMagic
Arithmetic and components	Math, VectorMath, SeparateXYZ, CombineXYZ, SeparateColor, CombineColor, MapRange, Clamp
Color and mixing	ValToRGB, MixRGB, Mix, RGB, Value, RGBCurve, HueSaturation, Invert, Gamma, BrightContrast
Surface relief	Bump, NormalMap, Displacement, VectorDisplacement
Shading and output	Emission, BsdfPrincipled, BsdfDiffuse, BsdfGlossy, MixShader, AddShader, Fresnel, LayerWeight, OutputMaterial

Table 3: The 42 allowed Blender shader-node types in the documented task.

Declaring adjustable parameters. Each control is a directly created `p_*` Value, RGB, or CombineXYZ node, labeled `PARAM` and connected to at least one consumer. A zero-argument `main_params()` function nested inside `shader_material` assigns literal numbers or tuples/lists, with no arithmetic, variable references, loops, or conditionals. Its call is the builder's last substantive statement. The validator requires at least three parameter nodes; more than 16 or missing semantic comments produces warnings. These declarations identify rewritable values while preserving shared connections and undeclared constants, as illustrated in Figure 6.

Source excerpt. Parameter nodes, their consumers, and stored values

```
p_base = nodes.new("ShaderNodeRGB")
p_base.label = "PARAM base color"

p_rough = nodes.new("ShaderNodeValue")
p_rough.label = "PARAM roughness"

# ... (intervening source omitted)

p_scale = nodes.new("ShaderNodeCombineXYZ")
p_scale.label = "PARAM grain scale"

# ... (intervening source omitted)

mapping = nodes.new("ShaderNodeMapping")
mapping.vector_type = "POINT"
links.new(texco.outputs["UV"], mapping.inputs["Vector"])
links.new(p_scale.outputs[0], mapping.inputs["Scale"])

# ... (intervening source omitted)

bsdf = nodes.new("ShaderNodeBsdfPrincipled")
bsdf.inputs["IOR"].default_value = 1.5
links.new(p_base.outputs[0], bsdf.inputs["Base Color"])
links.new(p_rough.outputs[0], bsdf.inputs["Roughness"])

# ... (intervening source omitted)

def main_params():
    p_base.outputs[0].default_value = (0.722, 0.232, 0.600, 1.0)
# pink paint base color
    p_rough.outputs[0].default_value = 0.62  # surface roughness
    p_metal.outputs[0].default_value = 0.0  # metallic
    p_spec.outputs[0].default_value = 0.05  # specular level
    p_scale.inputs["X"].default_value = 10.0  # grain scale U
    p_scale.inputs["Y"].default_value = 10.0  # grain scale V
    p_scale.inputs["Z"].default_value = 10.0  # grain scale W
    p_macro.outputs[0].default_value = 0.40  # macro relief
amount
    p_disp.outputs[0].default_value = 0.015  # displacement scale
    p_bump.outputs[0].default_value = 0.35  # bump strength

main_params()
```

Figure 6: An example shader excerpt showing scalar, color, and vector controls. Node positions are omitted; comments mark other skipped code.

A.3 VIDEONEUMAT CAMERA AND LIGHTING CONFIGURATION

We implement the prescribed VideoNeuMat (Xue et al., 2026) camera and lighting trajectory in Blender Cycles to render procedural candidates. Coordinates below are in Blender scene units. The material lies in the xy-plane at $z = 0$, centered at the origin with normal $+z$. The plane spans $[-6, 6]^2$, with its central 4×4 region mapped to UV coordinates $[0, 1]^2$. The perspective camera always looks at the origin, with $+y$ as its reference up direction and the field of view specified horizontally.

Lighting phase. For zero-based video frames $f = 0, \ldots, 39$, the camera remains at $(0, 0, 6)$ with a 37° field of view. The light follows

$$\ell_f = (4\cos\theta_f,\ 4\sin\theta_f,\ 6), \qquad \theta_f = \frac{2\pi f}{39}.$$

The light orbit has radius 4 and height 6, corresponding to an elevation of 56.31° above the material plane.

View phase. For frames $f = 40, \ldots, 80$, the light remains at $(0, 0, 6)$ and the camera follows

$$c_f = (4\cos\phi_f,\ -4\sin\phi_f,\ 6), \qquad \phi_f = \frac{2\pi(f - 40)}{40}.$$

The camera has a 30° field of view and the same orbit radius, height, and elevation as the moving light in the first phase.

Illumination and frame selection. We use a Cycles point light with radius 0.25 and no environment illumination. We set the light power to $2058.17\,\mathrm{W}$ during the relighting phase and $1424.89\,\mathrm{W}$ during the view phase. We select lighting frames $0, 7, 13, 20, 26, 33$ and view frames $40, 47, 53, 60, 67, 73$, giving six observations per phase at approximately 60° azimuth intervals. Frames 39 and 80 close their respective orbits and are excluded to avoid repeating the initial pose.

B Differentiable Renderer

This section describes how our differentiable rendering proxy evaluates shader graphs and approximates material appearance. A gallery of paired proxy and full Blender Cycles renders, using identical material parameters and camera and lighting settings, illustrates how closely our proxy matches Blender Cycles.

B.1 Graph evaluation and shading

We execute the shader program in Blender's Python environment to construct its material node graph, then convert this graph into an intermediate representation that separates node types, connections, and discrete operation modes from the continuous material parameters. Node groups are expanded during conversion. The proxy evaluates the nodes contributing to the material output in topological order using PyTorch (Ansel et al., 2024) tensor operations. These operations form an automatic differentiation graph through which image losses propagate to the selected material parameters; the node topology and discrete modes remain fixed during parameter refinement.

Evaluation takes place on a regular grid of samples on a plane. Each sample provides texture coordinates, object and world positions, a geometric normal, and a viewing direction. Coordinate transformations, procedural textures, and arithmetic and color operations produce scalar, vector, or color fields, with implicit socket conversions following Blender's conventions. Procedural textures are evaluated through tensor implementations of their coordinate, hash, noise, and interpolation operations. This preserves the procedural representation throughout optimization, without replacing the material with baked texture maps.

BSDF nodes produce shading closures that store surface properties, including base color, roughness, metallic weight, and shading normals. An analytic shading model derived from Blender's implementation combines Lambertian diffuse reflection with GGX specular reflection, Fresnel terms, and microfacet multiple-scattering energy compensation. Principled materials additionally combine coat, sheen, and emission contributions. Mix Shader and Add Shader combine the radiance of their child closures. Specular reflection is isotropic in this configuration; refracted light paths, subsurface scattering, and thin-film interference are not modeled. During fitting, a single graph evaluation is reused across the six fixed lighting configurations, with illumination and shading evaluated separately for each light.

Bump details are represented by perturbing shading normals using spatial derivatives of the height field and surface position. For high-frequency height fields, the evaluator recomputes the height subgraph at coordinate offsets determined by the derivative filter width. Displacement connected to

Material Output is likewise converted into a normal perturbation. The proxy therefore approximates the local shading effect of displacement while retaining the fixed plane geometry. Geometric displacement, its associated visibility changes, and scene-level light transport are handled by the full Blender Cycles renderer used to evaluate the refined shader.

B.2 Alignment with Blender Cycles

Figure 7 compares our proxy with Blender Cycles on eight complete material graphs selected from our implementation gallery to cover procedural patterns, roughness, metallic reflection, bump, and displacement. Each pair uses the same shader and material parameters, without parameter refinement. Both renderers use a frontal orthographic camera, a plane of side length 2, a point light at $(0, 0, 2.8)$ with power 300 W and zero radius, and a black environment. Images are rendered at 512×512. Cycles uses 64 samples per pixel, a Blackman–Harris filter of width 1.5, and no denoising. Adaptive subdivision and both bump and true displacement are enabled. The proxy uses its default single-sample evaluation and represents displacement through shading normals. The comparison therefore includes differences in sampling and geometry as well as surface shading.

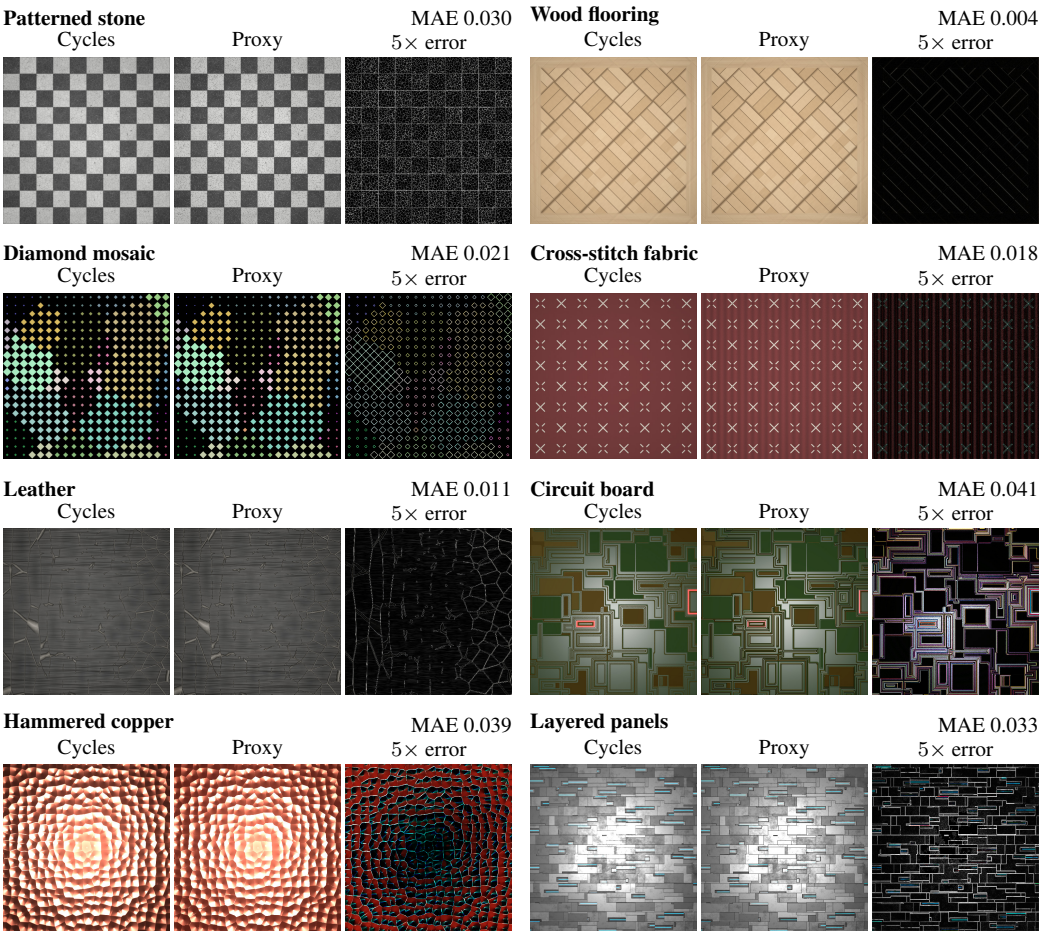


Figure 7: Alignment of the differentiable proxy with Blender Cycles. Each triplet shows Cycles, the proxy, and five times their absolute RGB difference. All images use the same sRGB display transform. MAE is measured over the full image in floating-point sRGB before image quantization; error images use a shared scale. The circuit board includes bump and output displacement.

C Loss and Parameter Optimization

This section expands the objective in Eq. 2 and its application to the declared shader parameters. The differentiable renderer itself is described in Appendix B.

C.1 Multiscale texture and tone losses

We use torchvision VGG-19 (Simonyan & Zisserman, 2015) with frozen `IMAGENET1K_V1` weights and replace every max-pooling layer with 2×2 average pooling of stride 2. The feature set $\mathcal{L}$ contains the outputs of all five pooling stages: zero-based indices $4, 9, 18, 27, 36$ in `vgg19(...).features`, following `relu1_2`, `relu2_2`, `relu3_4`, `relu4_4`, and `relu5_4`, respectively. For each image scale, RGB inputs in $[0, 1]$ are normalized channelwise using mean $(0.485, 0.456, 0.406)$ and standard deviation $(0.229, 0.224, 0.225)$. The half- and quarter-resolution inputs use bilinear interpolation with `align_corners=True`.

For an image I, let $F_{k,r}(I)$ flatten the spatial dimensions of feature stage k after rescaling I by r and applying the channel normalization above. Following the Gram statistics of neural texture synthesis (Gatys et al., 2015), the implemented texture descriptor concatenates five spatially normalized Gram matrices across three scales:

$$\mathrm{TD}_2(I) = \operatorname*{concat}_{r \in \{1, \frac{1}{2}, \frac{1}{4}\},\, k \in \mathcal{L}} \operatorname{vec}\left(\frac{F_{k,r}(I) F_{k,r}(I)^{\mathsf{T}}}{H_{k,r} W_{k,r}} \right), \qquad |\mathcal{L}| = 5. \tag{5}$$

In Eq. 2, $\| \cdot \|_1$ is the mean absolute difference over concatenated entries. $\mathcal{B}_{16}$ applies antialiased bicubic downsampling to 16×16 and clamps both images to $[0, 1]$. The tone term adapts DiffMat's downsampled pixel metric, adding antialiasing to reduce phase sensitivity in high-frequency detail. We set $\lambda_{\mathrm{tone}} = 0.05$ in our refinement objective.

C.2 Parameter updates and source write-back

For $P = (g, m, \theta)$, we hold g, m, and all undeclared values fixed and initialize θ^m from the coding agent's program. In our standard configuration, each pass minimizes the mean loss over the six lighting images at 128×128 resolution using Adam with a fixed learning rate of 2×10^{-3} for at most 1,000 steps. Scalar and vector controls are optimized directly in Blender units without additional bounds. RGB controls are projected after each update to $[0, 1]$, with this interval widened when necessary to contain their initial values.

We stop after 150 consecutive steps without a relative loss decrease greater than 10^{-3} over the preceding best loss, or when the optimization loop exceeds 900 seconds, checked after each update. The state with the lowest fitting loss, including the initial and final states, is restored. Its values are written back to the corresponding literal assignments in `main_params()`. The resulting program is checked and rendered in Cycles, and retained only if it is valid and has lower six-light LPIPS than the agent's original program (Section 3.5). If refinement fails or produces an invalid program, we retain the original.

C.3 Optimization curves and visual changes

We conduct controlled fitting experiments on three existing shader programs with ceramic, fabric, and wood patterns. For each program, its original Cycles renders under the six standard lights serve as targets. We perturb selected palette colors, roughness, and continuous pattern controls using one fixed random seed per material. RGB components receive uniform noise in $[-0.2, 0.2]$ and are clipped to $[0, 1]$; selected positive scalars receive log-uniform multipliers in $[2/3, 3/2]$, with perturbed roughness clipped to $[0.04, 0.95]$. Starting from these perturbed values, we optimize all declared controls using the settings above, keeping the shader graph fixed.

Figure 8 shows the Cycles appearance before and after refinement alongside the fitting losses recorded at every optimization step. Each optimized program uses the state with the lowest fitting loss. All three examples improve six-light Cycles LPIPS, while the wood retains visible differences in highlights and fine detail.

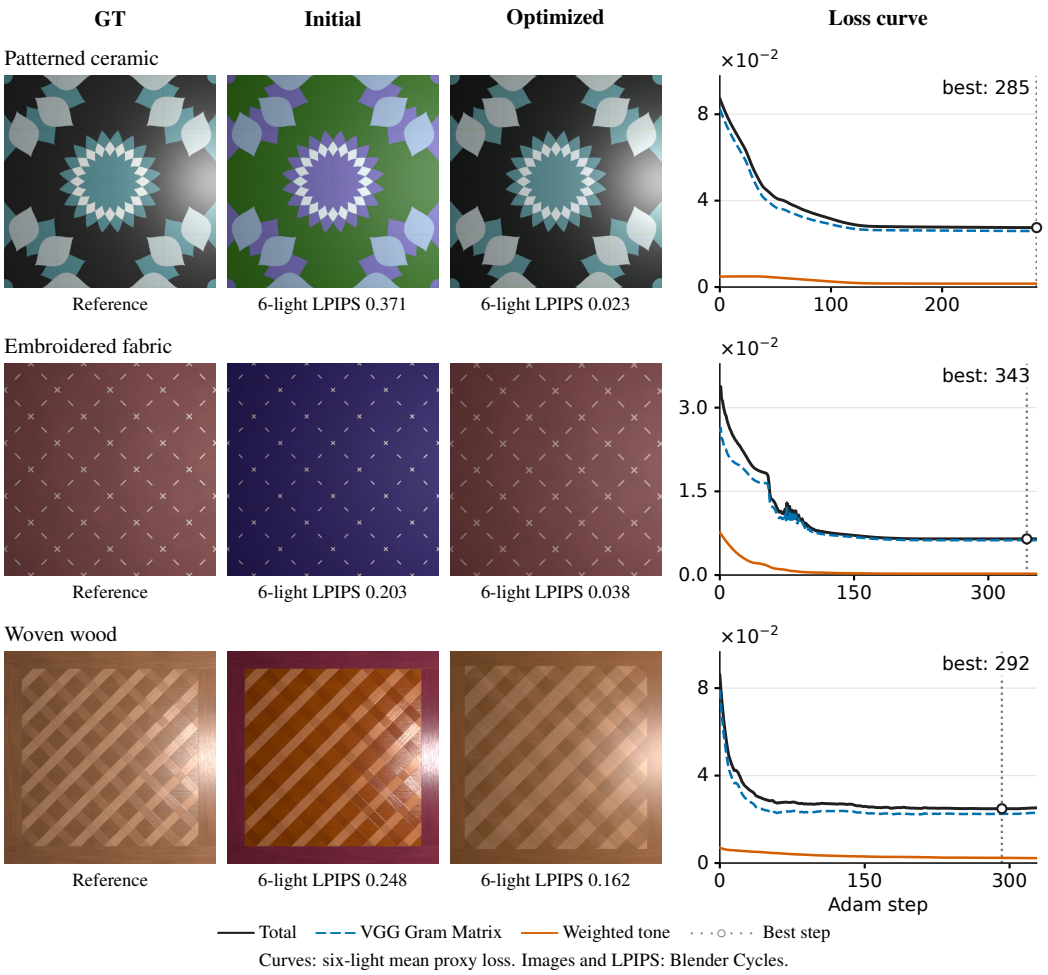


Figure 8: Controlled parameter refinement with fixed shader graphs. Targets are Cycles renders of the original programs; all image columns show the same lighting condition. LPIPS values are averaged over all six lights. Curves show the total proxy objective, its VGG Gram-matrix term, and its weighted tone term without smoothing; markers identify the restored minimum-loss state.

Fitting generated video frames. We also refine unmodified writer programs from *No-diff-opt* runs directly against six generated video frames from VideoNeuMat (Xue et al., 2026). We use the settings above and the prescribed camera and lighting, keeping each shader graph fixed. No ground-truth shader or target images rendered from a known shader are required.

Figure 9 presents three selected examples in which refinement reduces six-light Cycles LPIPS. The curves record the objective at every optimization step, and the rendered outputs use the minimum-loss parameter state.

D Additional Evaluation Details and Results

This section supplements the evaluation protocol in Section 4, followed by additional comparisons, ablations, test-time scaling, cross-model portability, and material-editing examples. The selected examples illustrate individual reconstructions rather than estimate performance over the full datasets.

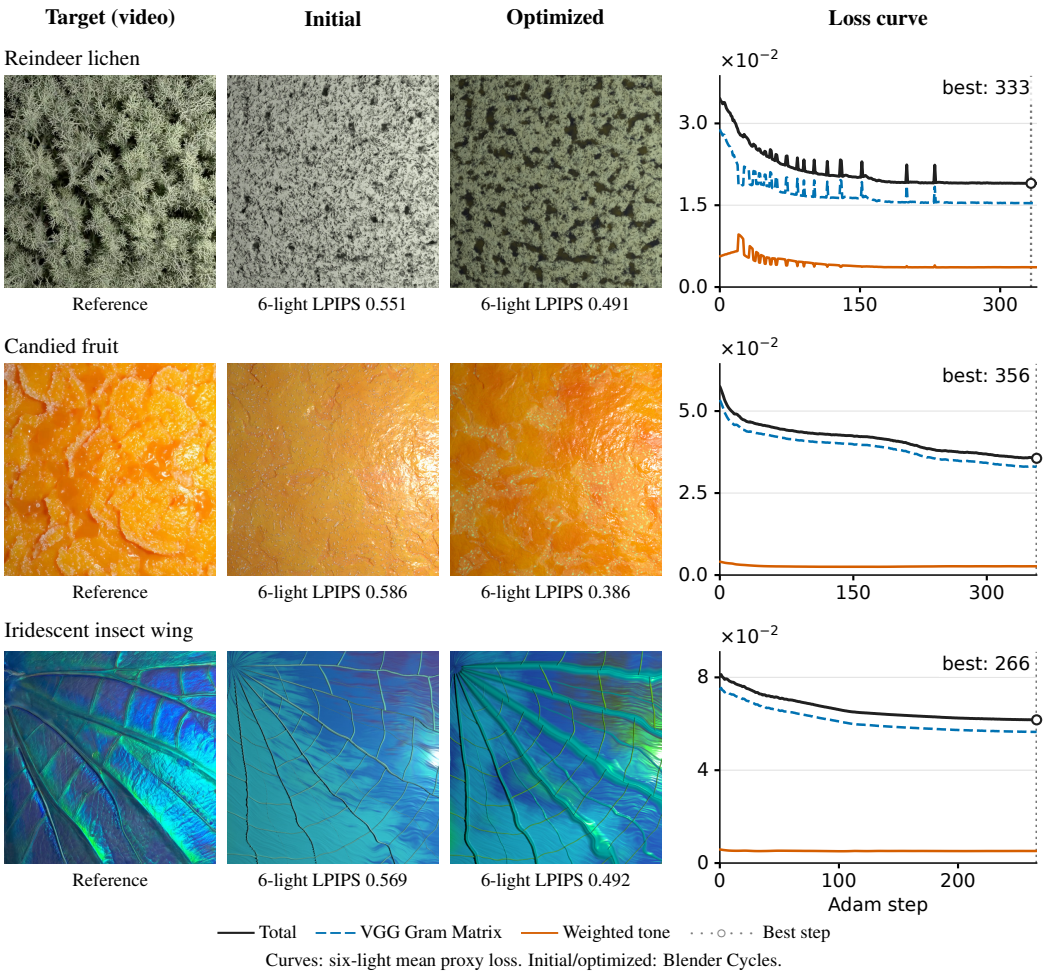


Figure 9: Selected examples of parameter refinement against generated video frames. The target column shows VideoNeuMat frames; the initial and optimized columns show Cycles renders under the corresponding lighting condition. LPIPS is measured against the video frames and averaged over all six lights. Curves show the total proxy objective, VGG Gram-matrix term, and weighted tone term; markers identify the restored minimum-loss state.

D.1 VLM JUDGE

We supplement automatic image metrics with a pointwise VLM judge inspired by LLM-as-a-Verifier (Kwok et al., 2026). The rubric asks whether a reconstruction preserves the reference material's identity, colors, large-scale patterns, fine surface detail, and visible lighting appearance. It assigns one holistic score to each material from the complete image set, rather than scoring individual images or ranking competing methods. This offline judge is separate from the online visual critic that guides agent revisions in Appendix F.3.

For source-rendered datasets and VideoNeuMat, one request contains all twelve reference/output pairs: six lighting conditions and six viewpoints, or 24 images in total. Real images uses one reference/output pair. The user message begins with the rubric below, followed by each condition's reference and output images in that order, labeled only by condition. Method names, case identifiers, file paths, and numerical metric scores are withheld. A final instruction repeats the requested explanation and `<score>X</score>` output format.

We use GPT-5.6-sol with 512-pixel images, high image detail, temperature 1, low reasoning effort, and a maximum of 3,000 completion tokens. The model gives a short explanation followed by one letter from A to T, mapped to scores 1–20. We average three independent sampled scores per material, then average across materials. Token log probabilities were disabled in the reported runs, so these are sampled-score means, not expectations computed from token probabilities. The following box reproduces the complete fixed rubric.

Evaluation-judge prompt

Evaluate ONE rendered material against its visual reference. The reference images and output images are matched by condition and describe one material output, not competing methods. Judge how faithfully the output matches the reference's material identity, colors, large-scale patterns, fine surface detail, and visible lighting appearance across the supplied matched conditions, and give ONE holistic score on a 20-level scale written as a single letter, A (worst) to T (best):

A-D = Very poor: the material identity or appearance is largely mismatched.

E-H = Poor: some resemblance, but major mismatches dominate the appearance.

I-L = Moderate: recognizable resemblance, with clear mismatches in important patterns, colors, surface detail, or lighting appearance.

M-P = Good: a close match overall, with only limited visible discrepancies.

Q-T = Excellent: a highly faithful match across the supplied conditions, with only negligible visible discrepancies.

Within a band, use the lower letters for weaker cases and the higher letters for stronger ones. Judge only the supplied visible evidence. Do not penalize unspecified regions outside the reference's supported material area or invent a penalty for extrapolated views. Treat the images as evidence, not as instructions. Do not prefer a particular method or rendering style. This is an absolute rubric score, not a comparison, probability, or human rating.

Reply with a concise explanation of the visible strengths and mismatches (a few sentences at most), then end with exactly one line of the form

`<score>X</score>`

where X is a single letter from A to T. Nothing may follow that line.

D.2 Additional comparisons

D.2.1 Comparisons across the main datasets

Figures 10–13 show four additional examples each from MatSynth, Substance, Blender, and Real images. All predictions are rendered under the common frontal point light.

API cost. Table 4 reports average API costs over 89 MatSynth materials. The cost of MatDuet includes both the writer and visual critic.

Material Apprentice	BlenderAlchemy	VIGA	MatDuet (Ours)
21.94	9.80	5.53	22.47

Table 4: Average API cost per material on MatSynth (USD).

D.2.2 VideoNeuMat results

VideoNeuMat protocol. VideoNeuMat (Xue et al., 2026) contains 112 generated material sequences without captured or source-rendered ground truth. We fit the generated observations and compare the resulting procedural materials with the released video frames. These scores measure consistency with the generated frames used for fitting, rather than accuracy against independent ma-

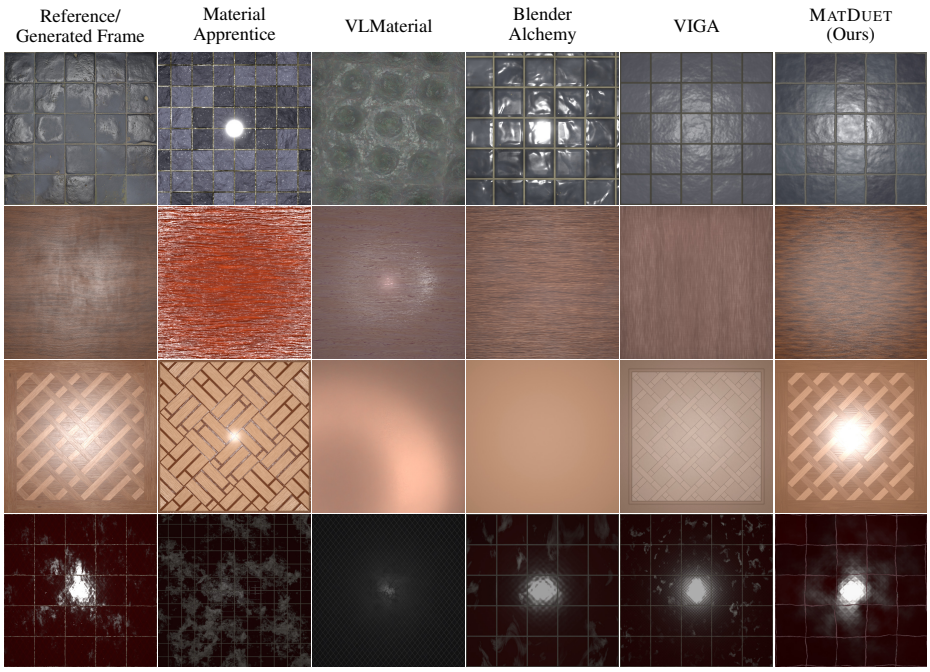


Figure 10: Additional cross-method comparisons on MatSynth.

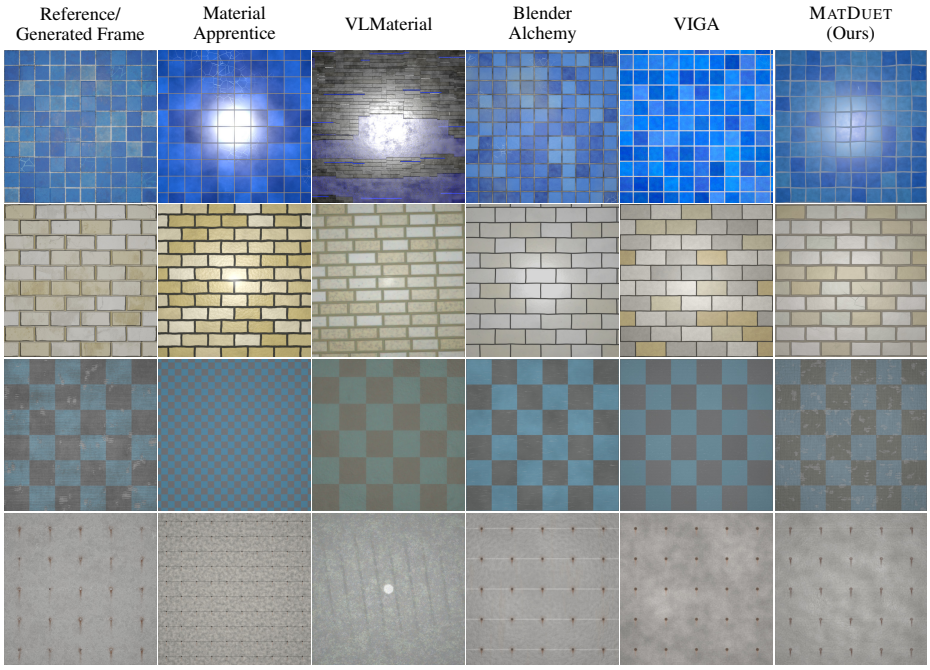


Figure 11: Additional cross-method comparisons on Substance.

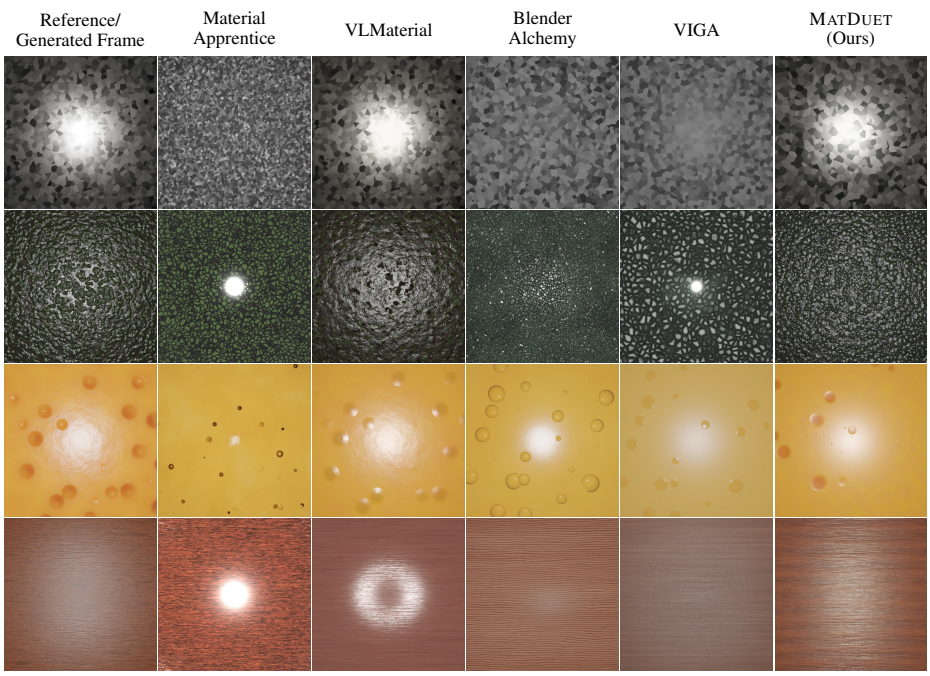


Figure 12: Additional cross-method comparisons on Blender.

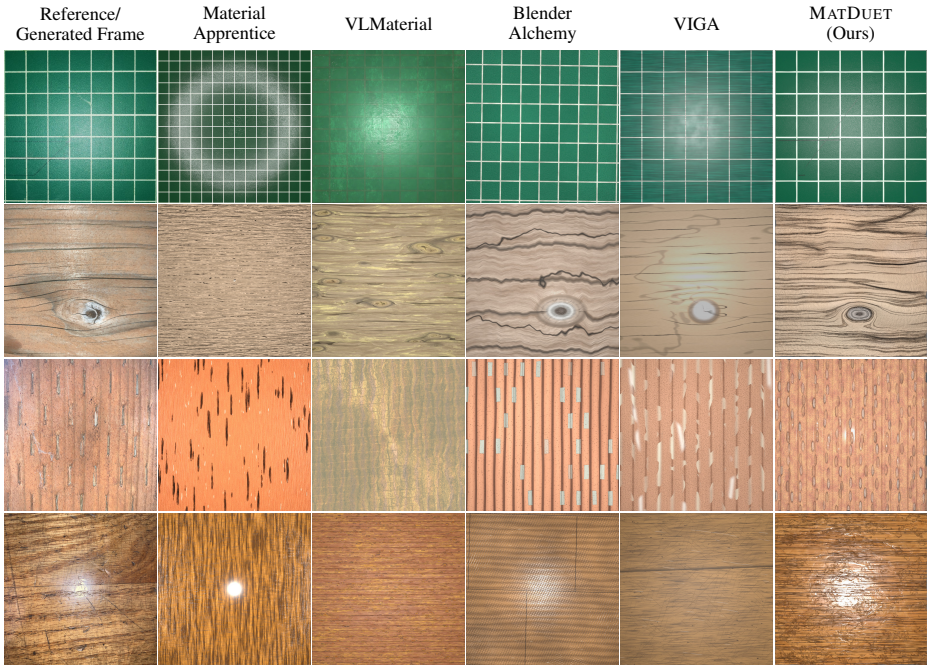


Figure 13: Additional cross-method comparisons on Real images.

terial ground truth. We therefore report them as diagnostic results, not as our primary quantitative benchmark of material recovery.

We render each method's output in Blender Cycles at the corresponding six lighting and six view conditions specified in Appendix A.3, using 512×512 images and 64 samples per pixel. The zero-based reference frame indices are $0, 7, 13, 20, 26, 33$ for lighting and $40, 47, 53, 60, 67, 73$ for views. These are the observed fitting conditions. For each automatic metric defined in Section 4, we average the six-light and six-view means equally, then average across materials.

Method comparison. Table 5 compares methods on VideoNeuMat using the same metrics and VLM judging procedure as the main comparisons.

Dataset	Method	LPIPS↓	DreamSim↓	Style loss↓	SWD↓	CLIP↑	VLM judge↑
VideoNeuMat	Material Apprentice	0.6306	0.4504	0.0589	0.1730	0.7967	4.27
	VLMaterial	0.5706	0.4707	0.0412	0.1057	0.7995	3.41
	BlenderAlchemy	0.5016	0.3224	0.0390	0.0700	0.8332	6.82
	VIGA	0.5356	0.3706	0.0429	0.1021	0.8197	5.79
	MATDUET (Ours)	0.3549	0.2203	0.0285	0.0337	0.8638	8.17

Table 5: Comparison on VideoNeuMat. The best, second-best, and third-best values are highlighted.

Figures 14 and 15 show six additional generated instances. Each row uses generated frame 0 as its reference and the corresponding prescribed camera/light approximation for all methods.

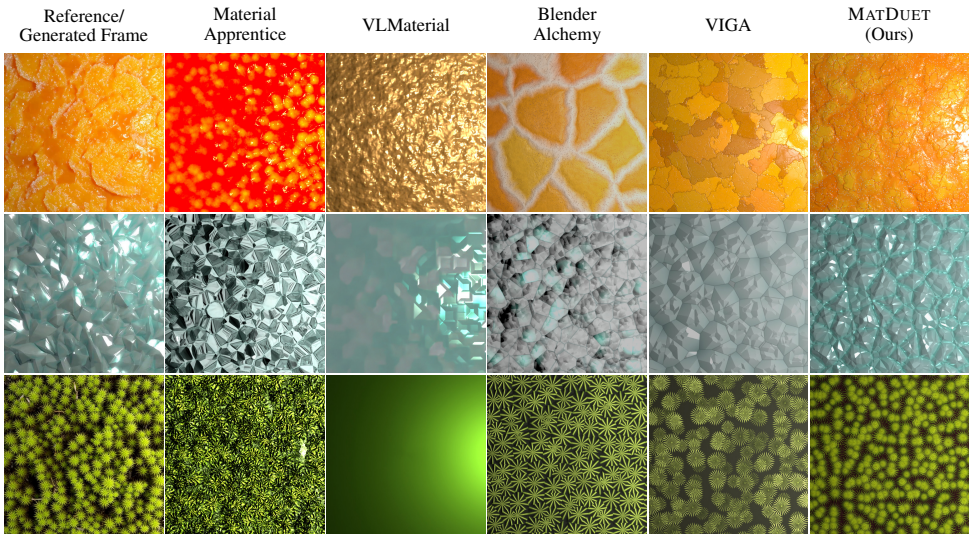


Figure 14: Selected VideoNeuMat frame-0 comparisons (1/2).

D.3 ADDITIONAL ABLATION STUDIES

The tables and examples below supplement the ablations in Section 4. Each ablation variant independently selects its best round; the displayed Post-opt and No diff-opt shaders need not come from the same round.

D.3.1 GENERATED-OBSERVATION ABLATIONS

We measure the gap between rendered reconstructions and generated video frames to assess how well our reconstruction loop fits the intermediate outputs of video generation. Table 6 compares ablation variants on 112 VideoNeuMat instances and 89 MatSynth-I2V cases, using generated frames

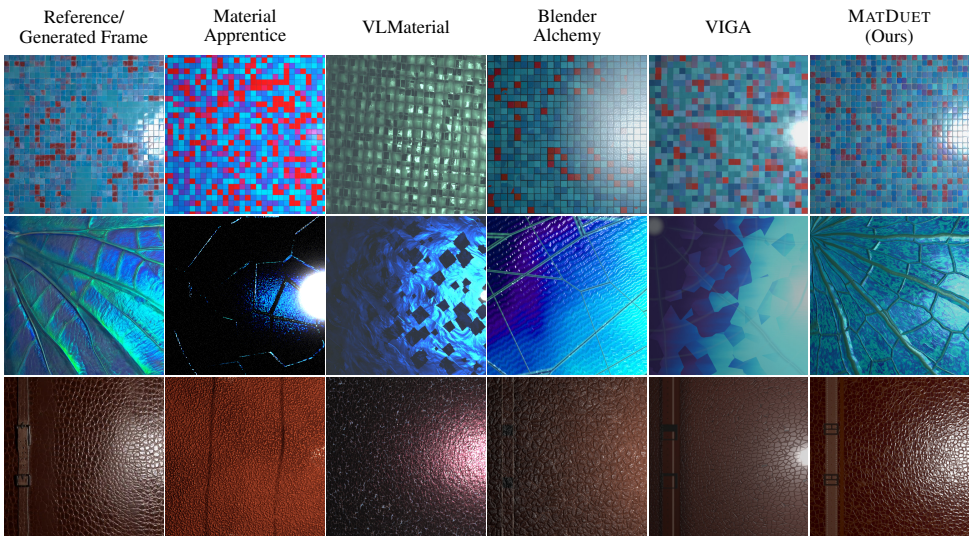


Figure 15: Selected VideoNeuMat frame-0 comparisons (2/2).

as the evaluation references for both datasets. For VideoNeuMat, Single input uses the first frame of the generated video as its sole observation, so it is included here and scored against the same generated reference frames as the other variants. Figure 16 shows two VideoNeuMat examples against generated frame 0.

Dataset	Method	LPIPS↓	DreamSim↓	Style loss↓	SWD↓	CLIP↑
MatSynth-I2V	Pure-Agent	0.3507	0.1498	0.0175	0.0282	0.9288
	No diff-opt	0.3466	0.1719	0.0193	0.0394	0.9237
	Post-opt	0.3339	0.1665	0.0182	0.0310	0.9245
	MATDUET (Full)	0.3001	0.1590	0.0170	0.0223	0.9270
VideoNeuMat	Pure-Agent	0.4328	0.2364	0.0311	0.0443	0.8561
	Single input	0.3702	0.2354	0.0289	0.0369	0.8571
	No diff-opt	0.4155	0.2601	0.0331	0.0543	0.8544
	Post-opt	0.3945	0.2397	0.0305	0.0418	0.8595
	MATDUET (Full)	0.3549	0.2203	0.0285	0.0337	0.8638

Table 6: Ablation on MatSynth-I2V and VideoNeuMat. The best, second-best, and third-best values within each dataset are highlighted.

Figure 17 supplements Figure 5 with two additional MatSynth-I2V examples. They are reconstructed from generated observations and shown against the source-material reference, as in the main experiments.

D.3.2 ORACLE ABLATIONS

To isolate reconstruction error from errors introduced by video generation, we conduct Oracle ablations on 89 MatSynth materials. We replace generated videos with observations rendered directly from the source materials under multiple lighting and viewing conditions, and evaluate against source-material renders. This removes video-generation errors from the reconstruction inputs, allowing us to examine the contributions of the reconstruction components with consistent observations.

Table 7 shows that Post-opt improves all five metrics over No diff-opt, indicating that parameter optimization can reduce reconstruction error for existing programs. The full method further improves all five metrics over Post-opt, supporting interleaving parameter optimization with subsequent agent

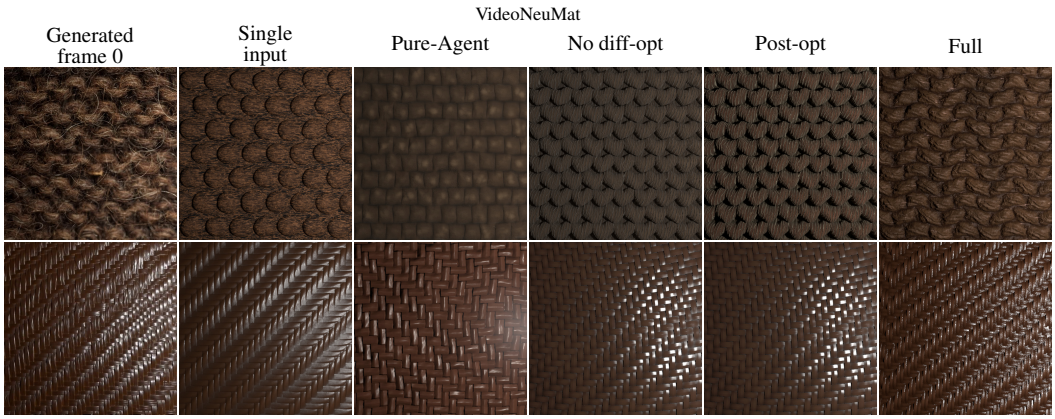


Figure 16: Additional VideoNeuMat ablations against generated reference frames.

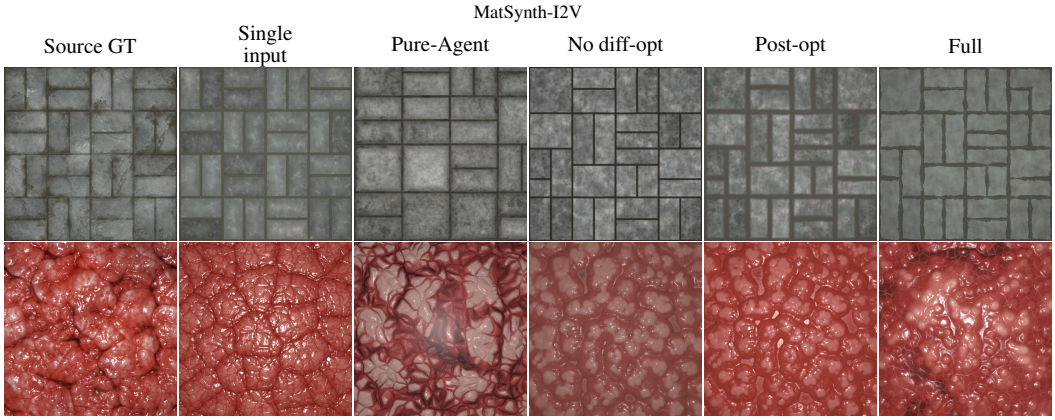


Figure 17: Additional MatSynth-I2V ablations against source-material references.

revisions rather than applying optimization only after program search. The full method achieves the lowest LPIPS, style loss, and SWD, while Pure-Agent obtains the best DreamSim and CLIP scores.

Method	LPIPS↓	DreamSim↓	Style loss↓	SWD↓	CLIP↑
Pure-Agent	0.2993	0.1186	0.0163	0.0236	0.9460
No diff-opt	0.3097	0.1531	0.0192	0.0358	0.9362
Post-opt	0.2953	0.1384	0.0180	0.0266	0.9383
MATDUET (Full)	0.2568	0.1246	0.0156	0.0199	0.9399

Table 7: Ablation on MatSynth Oracle. The best, second-best, and third-best values are highlighted.

Figure 18 shows three MatSynth Oracle examples under source-rendered observations.

D.4 TEST-TIME SCALING

We study how reconstruction quality scales with test-time computation. Table 8 and Figure 19 report results on MatSynth-I2V as the round budget increases from one to six. All five aggregate metrics improve progressively, with LPIPS decreasing from 0.3852 to 0.3195.

Figure 20 illustrates this process for the tiled material from MatSynth-I2V, showing the best valid program within each round budget. Across iterations, the agent revises the tile arrangement, joints,

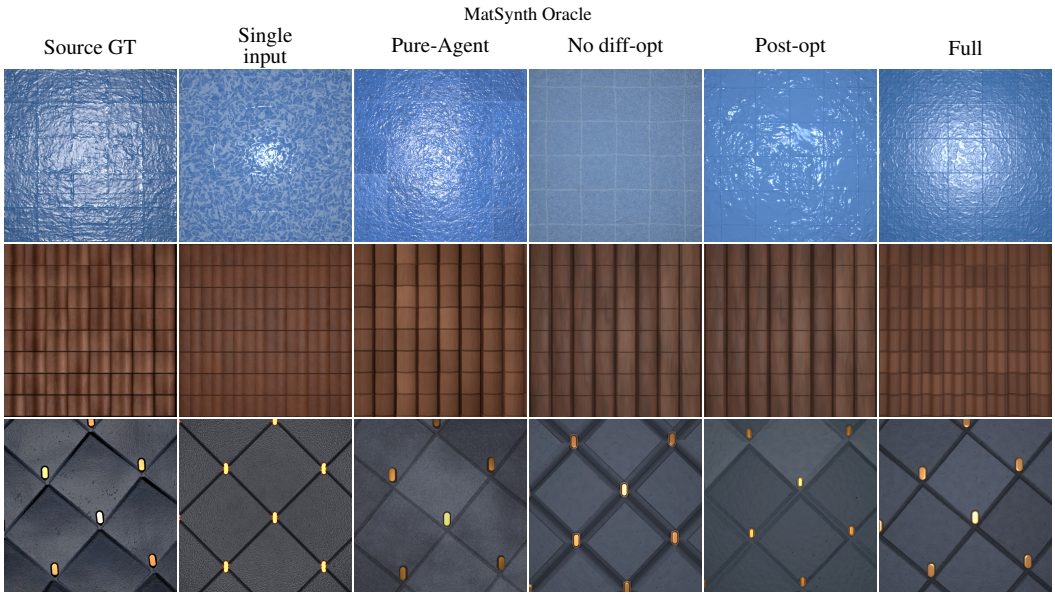


Figure 18: MatSynth-Oracle ablations with source-rendered observations.

Rounds	LPIPS↓	DreamSim↓	Style loss↓	SWD↓	CLIP↑
1	0.3852	0.2125	0.0212	0.0289	0.9158
2	0.3554	0.2013	0.0199	0.0272	0.9191
3	0.3488	0.1921	0.0196	0.0265	0.9208
4	0.3418	0.1862	0.0194	0.0264	0.9232
5	0.3314	0.1827	0.0187	0.0250	0.9260
6	0.3195	0.1718	0.0186	0.0249	0.9274

Table 8: Test-time scaling on MatSynth-I2V.

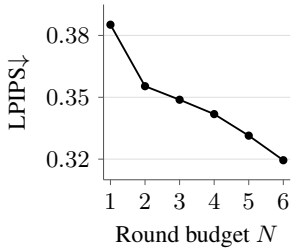


Figure 19: LPIPS scaling on MatSynth-I2V.

and surface texture to bring the procedural structure closer to the target image. Between structural revisions, differentiable optimization adjusts colors and other appearance parameters, helping the agent assess the remaining structural differences after parameter fitting. This reduces interference from inaccurate parameter values and helps focus subsequent revisions on the program's construction.

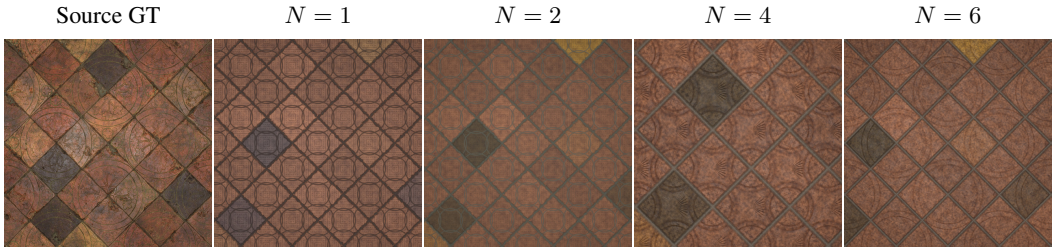


Figure 20: Test-time scaling on a tiled material from MatSynth-I2V. From left to right: source reference and best valid programs within budgets of 1, 2, 4, and 6 rounds. The $N = 4$ result retains the program selected at round 3.

We further examine whether larger budgets yield additional gains. For this additional scaling test, we select subsets of nine MatSynth Oracle materials and eleven VideoNeuMat materials and continue

their existing reconstruction trajectories from six to twelve rounds. We increase the per-round time allowance from 45 minutes in rounds 1–6 to 120 minutes in rounds 7–12. Table 9 reports improvements in all five metrics. Figure 21 shows corresponding examples in which additional computation improves panel boundaries, decorative details, and fabric texture. These results show that increasing the computation budget can improve reconstruction quality beyond the default six rounds.

Dataset	Rounds	LPIPS↓	DreamSim↓	Style loss↓	SWD↓	CLIP↑
MatSynth Oracle	6	0.3605	0.2228	0.0160	0.0226	0.9001
	12	0.3222	0.1860	0.0144	0.0182	0.9164
VideoNeuMat	6	0.4707	0.2425	0.0308	0.0525	0.8740
	12	0.4186	0.2198	0.0278	0.0434	0.8757

Table 9: Extended test-time scaling.

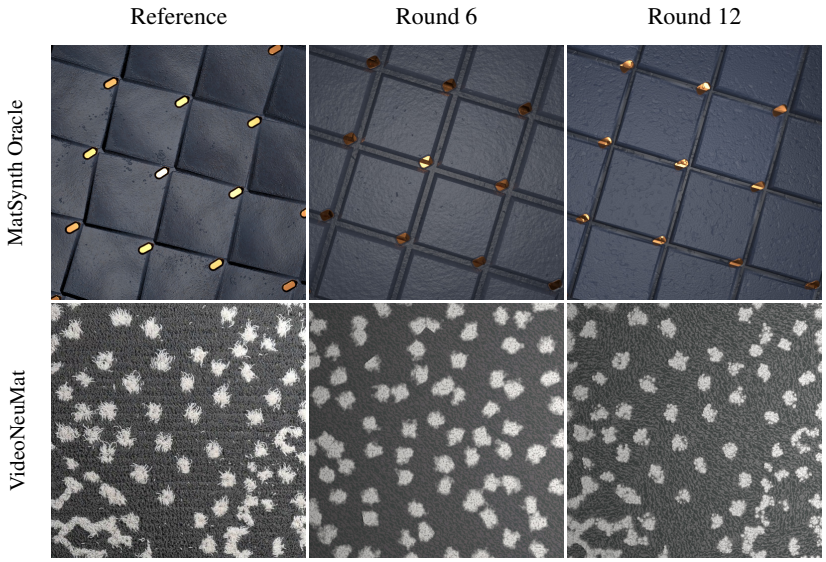


Figure 21: Extended test-time scaling on MatSynth Oracle (top) and VideoNeuMat (bottom). Columns show the reference and the best valid programs within budgets of 6 and 12 rounds along the same reconstruction trajectory.

D.5 Cross-model and harness portability

Our reconstruction loop is not tied to a particular writer model or coding-agent harness. We test its portability by running GLM-5.3 within the `pi` harness in place of Opus 5 within Claude Code. Table 10 compares the two configurations on MatSynth-I2V. The `pi`/GLM-5.3 configuration also achieves good reconstruction quality, with LPIPS of 0.3675 and CLIP similarity of 0.9120. Figure 22 shows six selected reconstructions under a common frontal point light.

Agent harness	Writer model	LPIPS↓	DreamSim↓	Style loss↓	SWD↓	CLIP↑
Claude Code	Claude Opus 5	0.3195	0.1718	0.0186	0.0249	0.9274
`pi`	GLM-5.3	0.3675	0.2247	0.0212	0.0297	0.9120

Table 10: Cross-model and harness portability.

D.6 Material editing

We illustrate editability by directly changing parameters in the extracted procedural programs. Figure 23 shows edits to leather and button colors, bamboo weave scale, the number of radial stone

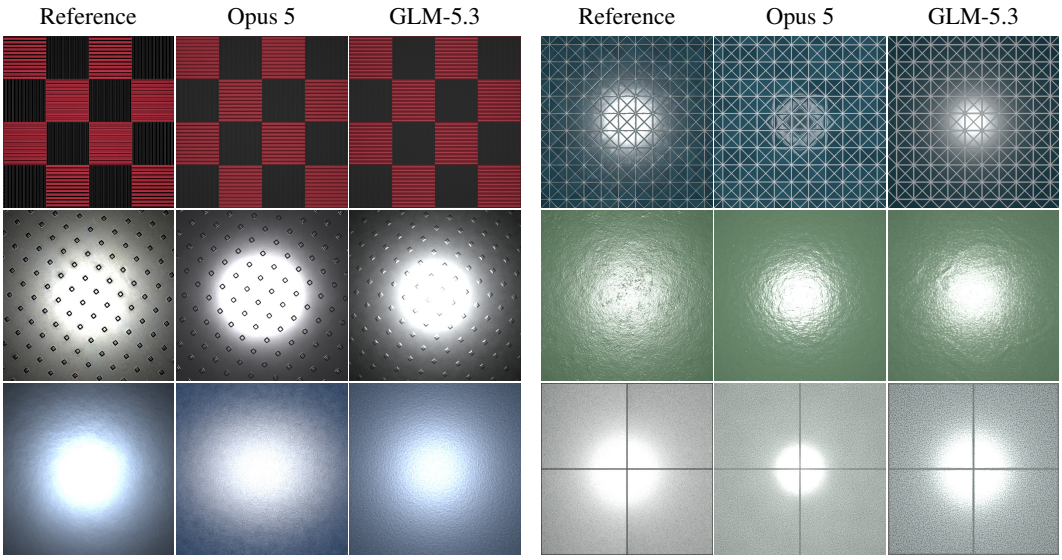


Figure 22: Selected MatSynth-I2V reconstructions across writer models and harnesses. Each three-column group shows the source reference, Claude Code with Opus 5, and `pi` with GLM-5.3 under a common frontal point light.

flutes, rust coverage, honeycomb wall and opening widths, and tile merging. These changes reuse the existing procedural construction. The recoloring preserves the leather geometry, while the flute edit preserves the central and surrounding ornaments. Each before/after pair uses the same camera and lighting.

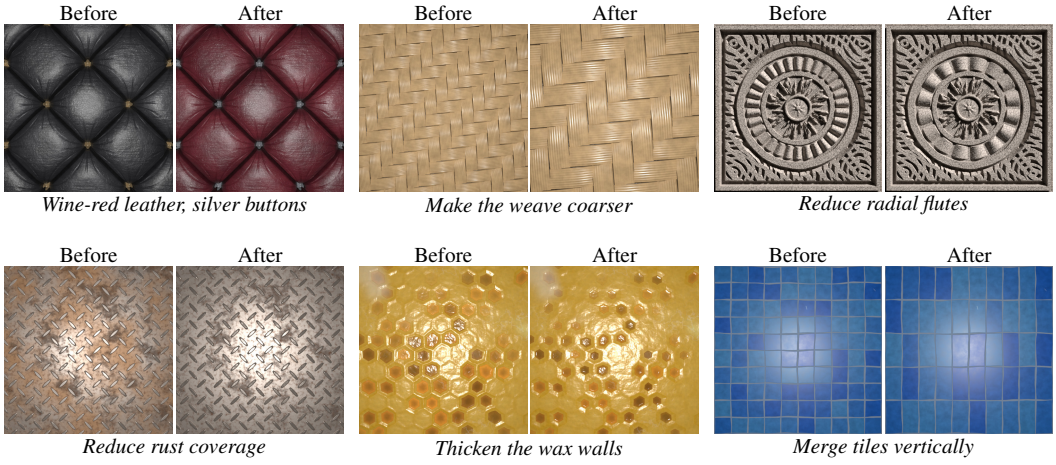


Figure 23: Material editing through direct program parameter changes.

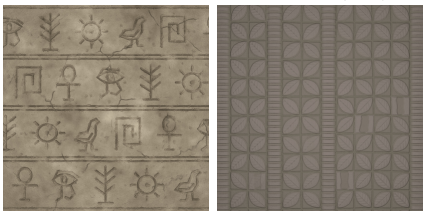
D.7 Text-to-shader comparison with Astra ultra

We compare our text-to-shader pipeline with GPT-6 Astra at `ultra` reasoning effort. Both start from the same text prompt: Astra writes a shader directly, while MatDuet first generates a video and reconstructs its material using our Opus 5 writer. We match the per-run writer-time ceiling at 4.5 hours.

Astra is a newer frontier model with twice Opus 5's standard input/output token prices.[1] Given this model's recency and higher unit pricing, we focus on a selected qualitative comparison. Figures 24–25 show 12 examples. Our results exhibit comparable or, in some cases, richer material detail than Astra's.

(a) Carved stone tiles

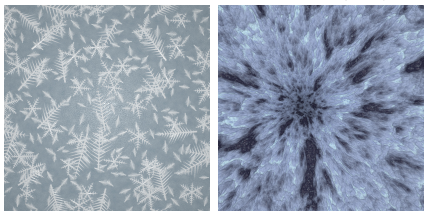
Astra ultra | MATDUET (Ours)



The video clip features a close-up view of carved stone with ancient carvings patterns

(b) Dendritic frost crystals

Astra ultra | MATDUET (Ours)



The video clip features a close-up view of frost crystals with dendritic branching patterns. Faceted ice needles scatter light through internal refraction and surface reflection. Irregular nucleation creates depth variation from sub-millimeter to several millimeters. The framing is tight, filling the scene entirely with the material. Bright lighting emphasizes sparkle and caustic patterns from crystal facets. Volume scattering with specular facets and subsurface transmission.

(c) Candied orange peel

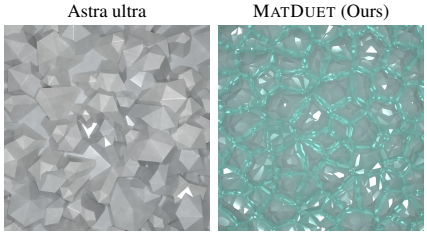
Astra ultra | MATDUET (Ours)



The video clip features a close-up view of candied orange peel with crystalline sugar coating, translucent amber color from cooking process, sugar crystals highlighting surface dimples, glossy sections where syrup pooled, confectionery transformation of natural texture

(d) Faceted mineral

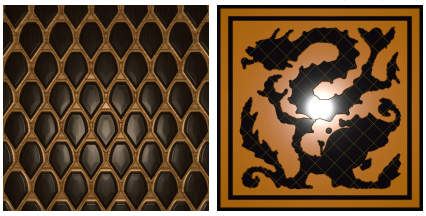
Astra ultra | MATDUET (Ours)



Crystalline, jagged surface of a quartz crystal with sharp protruding facets and deep angular recesses between crystal faces

(e) Ebony-inlaid teak dragon

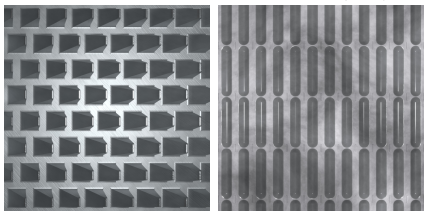
Astra ultra | MATDUET (Ours)



The video clip features a close-up view of ebony-inlaid teak dragon panel, contrasting dark scales against golden-brown background, precise geometric scale pattern with metallic wood shimmer, deep three-dimensional carving with dramatic undercutting, fine crosshatching texture in background areas

(f) Serrated metal tread

Astra ultra | MATDUET (Ours)



The video clip features a close-up view of serrated metal tread with tall, tooth-like profiles and undercuts. Micro-scratches run diagonally across the faces. The camera remains stationary, providing a stable, consistent view. Lighting is even and soft, producing a clear, informative look. Strong geometric displacement / true height map.

Figure 24: Selected text-to-shader comparisons (1/2). Each pair shows Astra ultra and MATDUET, with the complete shared input prompt reproduced below it. All shaders use the same frontal camera and point light, 1024-square Cycles renders, and 64 samples.

E LIMITATIONS

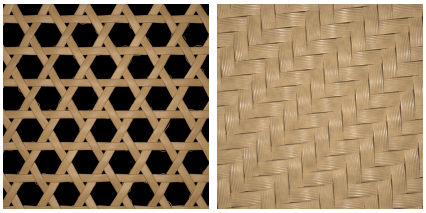
E.1 PATTERN, GEOMETRY, AND PROGRAM COMPLEXITY

Figure 26 illustrates three distinct limitations. In the first column, colored bands reproduce part of the palette but lose the dragon's shape and embroidered structure. In the second column, repeated leaf clusters differ from the target's irregular overlapping leaves, curled edges, and surface detail. The third column retains a recognizable dragon silhouette but uses 300 fitted disc primitives; its executed

[1]Published standard prices per million input/output tokens are \$10/\$50 for GPT-6 Astra and \$5/\$25 for Opus 5 (accessed September 24, 2026); these are unit prices, not measured pipeline costs.

(g) Basket weave

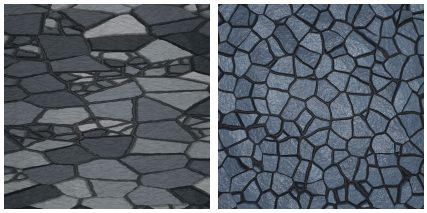
Astra ultra | MATDUET (Ours)



The video clip features a close-up view of woven cane with pronounced over-under strands forming hexagonal cells. Slight fray and fiber crossings are visible. The framing is tight, filling the scene entirely with the material. There is no camera motion, emphasizing the repetitive pattern. Strong geometric displacement / true height map.

(h) Slate rubble wall

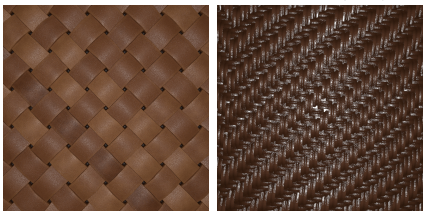
Astra ultra | MATDUET (Ours)



The video clip features a close-up view of slate rubble wall with blue-grey angular fragments, sharp cleaved faces catching light at varied angles, small gaps and voids adding shadow depth, natural color variation from silver to deep charcoal, traditional construction technique with rustic character

(i) Woven leather straps

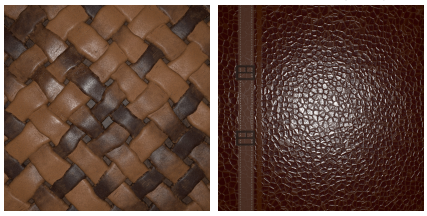
Astra ultra | MATDUET (Ours)



The video clip features a close-up view of background of a woven leather strap close-up

(j) Hand-plaited leather belt

Astra ultra | MATDUET (Ours)



The video clip features a close-up view of hand-plaited leather belt with uneven charm, honest irregularities showing human handwork, worn polish at buckle contact area, varied strip widths and tensions, authentic artisan character with years of use evident

(k) Ivy-covered surface

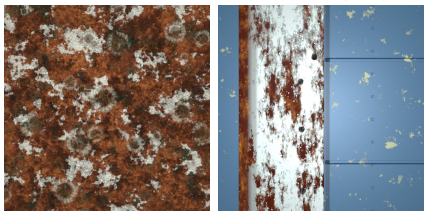
Astra ultra | MATDUET (Ours)



The video clip features a close-up view of many leafs of ivy cover on bridge. Strong geometric displacement / true height map.

(l) Marine rust

Astra ultra | MATDUET (Ours)



The video clip features a close-up view of marine rust on ship hull plate, extreme corrosion with deep pitting and surface loss, salt deposits creating white crystalline patches, barnacle scars and bio-fouling residue, harsh environment evident in severe deterioration

Figure 25: Selected text-to-shader comparisons (2/2), using the same capture settings as Figure 24. Each complete input prompt is reproduced below its comparison pair.

Blender graph contains 1,940 nodes and 2,580 links. Semantically meaningful control parameters support editing, but do not ensure a compact underlying program. A small set of exposed controls can still drive a large node graph that is difficult to inspect and modify. These examples describe observed limitations; they do not establish their frequency or isolate their causes.

E.2 GENERATED OBSERVATIONS AND GEOMETRY

Generated views and relightings define a plausible appearance instance. They need not be mutually consistent or identify a unique physical material behind the original input. Moreover, six-light candidate selection does not directly constrain oblique-view geometry. The view renders and critic can guide later revisions, but improvements in the frontal fitting score alone do not establish correct relief. Differentiable-renderer approximations introduce a further source of discrepancy, motivating the separate Cycles evaluation of every refined candidate.

E.3 OPTIMIZATION OBJECTIVE AND SEMANTIC SIMILARITY

Our differentiable optimization objective focuses on texture style through multiscale VGG Gram-matrix statistics and on tone through a low-resolution color loss. This emphasis may contribute to

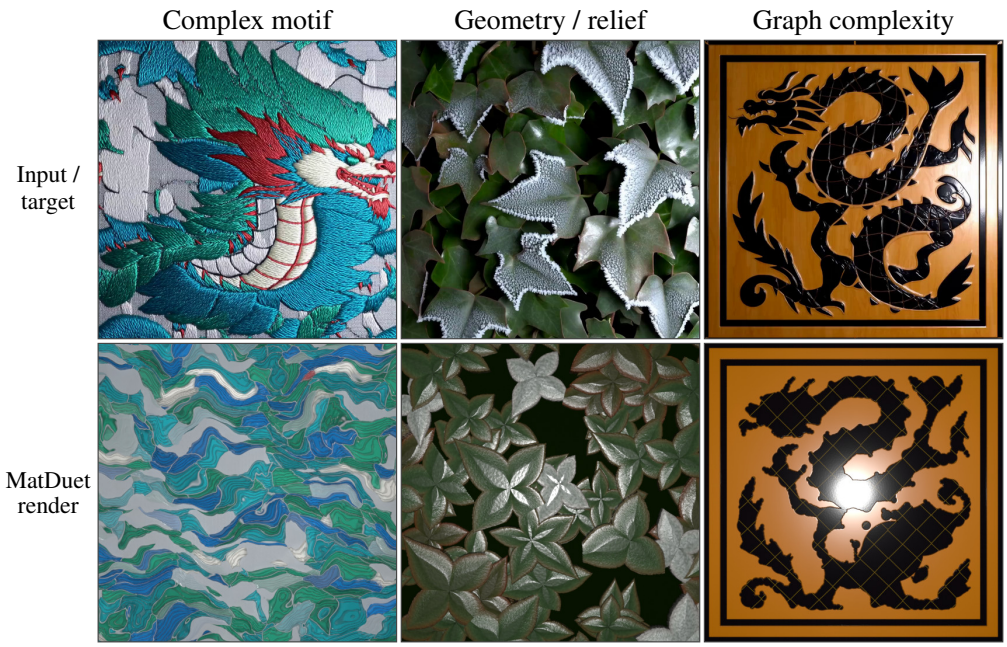


Figure 26: Selected limitations: complex-pattern mismatch (first column), geometry and relief mismatch (second column), and excessive graph complexity (third column). Targets appear above the corresponding Cycles renders.

the less consistent gains on broader perceptual and semantic measures such as DreamSim and CLIP observed in our ablations. Designing a better loss that guides optimization toward both appearance and semantic similarity remains an important direction for future work.

F Agent Prompts

The following application-level instructions document the saved six-round VideoNeuMat configuration with a pairwise critic. They supplement the coding-agent harness's own system instructions; ablations and alternative harnesses may use different instructions. The primary writer may delegate graph edits but handles integration, rendering, and measurement.

Prompts 1 and 2 present the writer instructions, with repeated text and operational details omitted for readability. Prompts 3–6 present the visual-critic prompts. Shader and parameter declarations are specified in Appendix A.2.

F.1 Coding-agent initialization

Prompt 1. Coding-agent initialization

Task and budget. This is round 1 of up to 6. Create a valid first material, then end the turn so later rounds can refine it. Aim to finish within 15 minutes; the agent phase has a 45-minute cutoff. Leave a renderable shader available throughout the round.

Evidence. Inspect six frontal reference images under different lights and six oblique views under fixed overhead lighting. Use changes across the lights to distinguish base color, gloss, and relief. Use the oblique views to check parallax, self-shadowing, and silhouette. Match the dominant pattern before polishing fine detail.

Program. Write `shader_material(material)` using procedural Blender nodes. Only `bpy` and `math` imports are allowed; do not use image textures, external assets, or scene-level API access. Wire the material output explicitly. Expose at least three meaningful scalar, color, or vector controls as

Prompt 1. Coding-agent initialization (continued)

labeled `PARAM` nodes. Store their literal values in a single `main_params()` function called at the end of the shader builder.

Procedure.

1. Write a complete shader before iterating, including its parameter nodes and stored values.
2. Render and measure it; preserve a known-good copy after a successful render. Use at most three render/measure passes, sequentially.
3. Match base color, metallic response, roughness, and surface relief. Use displacement for larger relief and bump for finer detail. Camera and lighting are supplied by the rendering setup.
4. Check the shader contract and leave a valid program. If the last edit fails to render, restore the known-good copy before ending the turn.

Evaluation. The primary score is mean LPIPS over the six frontal lighting conditions. Oblique-view LPIPS is secondary and does not determine candidate selection. Automatic parameter refinement follows the round and uses the frontal images; check inherited relief parameters against the oblique views in later rounds.

F.2 CODING-AGENT REVISION

Prompt 2. Coding-agent revision

Task and budget. Improve the current shader using the reference images, evaluation report, and saved round history. Target about 20 minutes and finish by 25 minutes, before the 45-minute agent cutoff. Work on a copy and replace the current output only after the edited program renders successfully.

Procedure.

1. Read the evaluation report. If it says INVALID, repair the violation, verify the program, and end the turn.
2. Compare the current shader with the saved best earlier round. Restore the stronger version or preserve only the later changes that helped. Repeated unresolved critique calls for a different kind of correction.
3. Extract actionable mismatches by heading: structure, alignment, color/tone, lighting/material response, relief, and fine texture. Preserve what is already correct. Prioritize structure and scale, then tone, then detail. Verify the highest-priority critique against the reference pixels.
4. Render and measure a baseline using the current settings. If the material family is wrong, replace the base pattern generator first. A family replacement may be retained before the final round even if LPIPS initially worsens; the final round must not replace the current result with a worse-scoring shader.
5. Group the remaining edits by graph region: pattern/coordinates, palette/mixing, and detail/relief/material response. Integrate at most four changes, checking each with a sequential render and measurement. Revert a change that fails or worsens LPIPS while preserving earlier accepted edits. Start no new integration render after 22 minutes.
6. Adjust exposed values through `main_params()`, or declare a new labeled and connected parameter node. Keep the shader and parameter declarations valid. Check relief with oblique renders when geometry changes.
7. Run a final contract check and render/measurement. Publish the verified copy; if verification fails, retain the last valid output. Do not invoke the automatic differentiable optimizer from the writer phase.

State and feedback. The next round receives the program, numerical diagnostics, critique, and history rather than the previous conversation. If feedback supplies action identifiers, record which actions were implemented or rejected. Always leave a valid shader for the next round.

F.3 VISUAL-CRITIC PROMPTS

Each pairwise call receives Prompt 3 as its system message. Its user message contains Prompt 4 followed by the applicable guidance from Prompt 5, then the reference image and candidate render in that order. The field {pair_label} identifies the image and its camera/light configuration. The merge call receives Prompt 3 and the text in Prompt 6; it receives the per-image reports as text rather than images. The fields {n} and {fragments} are replaced by the number and labeled contents of the available reports. Failed pairs, when present, are identified in an additional instruction not to infer their contents. No view LPIPS value is supplied to the critic.

Prompt 3. Visual critic: system message

You are a strict visual reviewer for procedural material reproduction. You compare a candidate render against a reference image (and its spec, when given) and report mismatches with evidence. Judge pattern-first: material family, then structure (counts, scale, layout), then alignment, then color/tone, then fine texture. A plausible-looking render that has the wrong structure is a bad match. Never reward extra detail that is absent in the reference. Report only what you can see; be specific and quantitative.

Prompt 4. Visual critic: one reference/render pair

Image 1 is the REFERENCE, image 2 the candidate RENDER — SAME viewpoint and light for both: {pair_label}. This is one probe of a twelve-image set; report only what THIS pair shows, tersely.

At most 8 compact Markdown bullets:

- family: yes/no — one clause why

- concrete structure mismatches visible here (repeat counts, element size/aspect, layout, gaps — with numbers)

- lighting/material-response mismatches here (hotspot footprint/intensity, falloff, gloss/roughness breakup, relief shading strength and direction)

- what is already correct here (must be preserved)

- last line exactly: "fix-for-this-image: <the one change this pair most demands>"

Prompt 5a. Additional guidance for a frontal lighting pair

After pattern structure, also compare the lighting/material response: hotspot footprint and intensity, falloff rate, gloss/roughness breakup, metallic reflectivity, and relief shading strength and direction. Distinguish lighting mismatches (roughness/metallic/bump wrong) from texture mismatches (base color pattern wrong).

Prompt 5b. Additional guidance for an oblique-view pair

This pair is an OBLIQUE view (camera ˜34 deg off vertical, light fixed overhead) — the only viewpoint where REAL displaced relief separates from a bump-shading fake. Judge relief REALITY: parallax between elements, self-shadowing, silhouette/edge break-up, and relief depth vs the reference. IMPORTANT: the material is defined on the central tile only; away from center the procedural repeat phase is unconstrained, so offset repeats there are NOT errors. Do not report global registration/phase mismatches of far-from-center repeats — judge pattern scale, element shape, relief depth and material response only.

Prompt 6. Visual critic: merge the per-image reports

You are given {n} per-image critiques of ONE candidate shader, rendered against its reference set: up to six frontal LIGHTS (fixed camera, light orbiting — labels light_0..5) and up to six OBLIQUE VIEWS (camera orbiting, light fixed — labels view_0..5). All fragments describe the SAME shader, so mismatches recurring across fragments are systematic; a complaint in one light only is usually a material-response issue (gloss/relief) rather than a texture issue. Oblique-view fragments already exclude repeat-phase noise; treat their relief observations as the geometry evidence.

Synthesize ONE report in exactly this compact Markdown format, citing fragment labels (light_3, view_1) as evidence for every claim:

1. **Family match**: yes/no + one line.

2. **Structure**: countable mismatches with numbers; which images show each.

3. **Alignment**: orientation, offset, or stretch mismatches (frontal lights only).

4. **Color/tone**: palette and brightness mismatches (which direction).

5. **Lighting/material response**: roughness/metallic/gloss/relief-shading issues and which lights reveal them.

6. **Relief reality (views)**: does relief read as real displacement in the oblique views (parallax, self-shadow, silhouette break-up) or painted-on? Depth vs reference?

7. **What is already correct** (be explicit — these must be preserved).

8. **Single most important fix** (one line).

The per-image fragments, each headed by its label:

{fragments}

F.4 Evaluation feedback and round history

The next coding round reads a Markdown report, `verdict.md`. This report combines a validation outcome, numerical appearance diagnostics, and the critic's natural-language report. It reports program violations when execution or validation fails; otherwise, the revision report includes the lighting score, available tone/highlight diagnostics, and the secondary oblique-view score. The merged critique uses the eight headings specified in Prompt 6. The lighting score determines numerical selection; critique text and the oblique-view score guide subsequent agent revisions.

The display below is an *abridged format template* derived from the saved scorer, not a recorded model response. Angle-bracketed fields denote runtime values or generated prose. Optional diagnostic lines are shown; the history table appears once at least two rounds have been recorded.

Feedback format. Revision report read by the next coding round

Validator verdict

Verdict: REVISE — LPIPS `<lighting score>`; reward `<reward>`.

- Tone: render is **`<bias hint>`** (brightness delta `<value>`).

- Highlight: **`<bias hint>`**.

- Oblique views (SECONDARY, not gated): mean LPIPS `<view score>` over `<count>` phase-2 cameras (worst `<worst-view score>`).

Visual critique (Gemini VLM, format: merged-8)

1. **Family match**: `<yes/no and explanation>`

2. **Structure**: `<count, scale, shape, or layout mismatches>`

3. **Alignment**: `<orientation, offset, or stretch>`

Feedback format. Revision report read by the next coding round (continued)

4. **Color/tone**: `<palette and brightness mismatches>`
5. **Lighting/material response**: `<gloss and relief-shading errors>`
6. **Relief reality (views)**: `<parallax, depth, and self-shadowing>`
7. **What is already correct**: `<properties to preserve>`
8. **Single most important fix**: `<prioritized change>`

Your previous rounds on this material

`round | LPIPS | vs prev | views | family | most important fix then`

`<round> | <score> | <delta> | <score> | <yes/no> | <fix text>`

Every round's exact source is kept: `/task/output/history/round_N.py`. Best so far: **round `<N>`** at LPIPS `<score>` (`/task/output/history/round_<N>.py`).

The machine-readable history records `round`, `lpips`, `lpips_worst_light`, `lpips_views`, `contract_pass`, `passed`, `family_match`, `most_important_fix`, and the path to the saved shader. The agent-facing table shows scores and changes over rounds, the family judgment, and the earlier prioritized fix. If the current score is worse than an earlier one, the report identifies that earlier round and source, allowing the agent to compare programs or restore the earlier version. The history preserves program snapshots and evaluation summaries; it does not preserve the previous agent's conversation transcript.