

Quality-Centric Single-Image Procedural Material Generation

by

Beichen Li

B.Eng., Tsinghua University, 2018

S.M., Massachusetts Institute of Technology, 2021

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2025

© 2025 Beichen Li. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: Beichen Li
Department of Electrical Engineering and Computer Science
January 24, 2025

Certified by: Wojciech Matusik
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: Leslie A. Kolodziejewski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee for Graduate Students

Quality-Centric Single-Image Procedural Material Generation

by

Beichen Li

Submitted to the Department of Electrical Engineering and Computer Science
on January 24, 2025 in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

ABSTRACT

Procedural materials, represented as functional node graphs, are ubiquitous in computer graphics for photorealistic material appearance design. They allow users to perform intuitive and precise editing to achieve desired visual appearances. However, even for experienced artists, creating a procedural material given an input image requires professional knowledge and significant effort. Current inverse procedural material modeling approaches enable the automatic generation of procedural materials from input images. However, the visual quality of the generated materials is fundamentally limited by insufficient high-quality training data from industry-standard procedural materials, reliance on token-space supervision without visual feedback, and the absence of approximation-free node parameter post-optimization. My thesis presents advanced dataset augmentation, model training, and parameter post-optimization algorithms to address these challenges, significantly improving the perceptual match between the generated procedural material and the input image. Furthermore, the methodologies can be applied to other inverse procedural graphics problems to expedite similar artistic creation processes.

Thesis supervisor: Wojciech Matusik

Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

My Ph.D. journey was an unprecedented adventure riddled with challenges and obstacles. In retrospect, I could never have overcome those hardships without the kindness, guidance, and unwavering support of many individuals, to whom I am deeply grateful.

First and foremost, I want to express my sincere gratitude to my advisor, Prof. Wojciech Matusik, for mentoring and supporting me throughout the years. He provided me with invaluable opportunities to explore my research interests, practice critical thinking, establish self-resilience, and venture forward against the odds. His genuine faith in my potential for a successful career was coupled with remarkable empathy and patience as I grappled with research difficulties. I am equally thankful for his professionalism and transparency when settling our occasional differences, which prompted me to reassess my approaches and exercise better judgment. Above all else, working with him has been a transformative experience that enabled me to develop into a competent independent researcher.

I would like to extend my appreciation to Prof. Armando Solar-Lezama and Prof. Jonathan Ragan-Kelley, members of my thesis committee. Armando's astute observation, profound intuition, and keen participation contributed tremendously to my research. His constructive, encouraging comments have always reassured me that I was heading on the right path. Jonathan's works are a major source of inspiration for the differentiable programming methodology as I will discuss in my thesis. It is my pleasure to invite him as a thesis reader and receive his insightful feedback.

I share my heartfelt thanks to Prof. Aleksander Madry, my academic advisor, for his constant support and valuable career advice communicated through our regular meetings before each semester starts. I am also profoundly thankful to Dr. Liang Shi, Prof. Wan Shou, Prof. Bolei Deng, Dr. Miloš Hašan, Dr. Kalyan Sunkavalli, and Dr. Yiwei Hu for their exceptional mentorship and collaboration, which have been instrumental to my accomplishments.

Throughout my Ph.D., I had the privilege of collaborating with many excellent researchers whose intellect and creativity fundamentally enriched my works. Their contributions as co-authors through discussions, experimentation, and manuscript editing are integral to this thesis: Prof. Tae-Hyun Oh, Dr. Yuanming Hu, Prof. Yiyue Luo, Dr. Radomír Měch, Dr. Tamy Boubekur, Dr. Paul Guerrero, Dr. Valentin Deschaintre, Rundi Wu, Prof. Changxi Zheng, and Prof. Bernd Bickel. Furthermore, my journey would never have been joyful and rewarding without the harmonious, inclusive, stimulating, and dynamic environment of the Computational Design and Fabrication Group (CDFG) at MIT CSAIL. I would like to thank my former and present labmates for their friendship and tolerance: Prof. Petr Kellnhofer, Dr. Changil Kim, Michael Foshey, Prof. Tao Du, Dr. Alexandre Kaspar, Prof. Andrew Spielberg, Harrison Wang, James Minor, Dr. Jie Xu, Liane Makatura, Pingchuan Ma, Yunsheng Tian, Dr. Allan Zhao, Dr. Kui Wu, Prof. Mina Konaković-Luković, Timothy Erps, Yifei Li, Minghao Guo, Prof. Bohan Wang, Prof. Ruitao Su, Donghun Ryu, Lara Zlokapa, Yichen Li, Young Joong Lee, Megan Tjandrasuwita, Prof. Yichen Chen, Dr. Crystal Owens, Joshua Jacob,

Michael Sun, Devin Murphy, Alston Lo, and Dr. Benjamin Jones.

My gratitude extends further to Ksharma Ananthapura, Erin Buckley, Roger White, Mieke Moran, Felicia Raton, and Megan Farmer for their administrative efforts. They played a pivotal role in managing all the logistics, ensuring seamless communication, and organizing recreational activities that brought our lab closer together. I am additionally thankful to the departmental staff from MIT EECS and MIT CSAIL for their assistance: Prof. Leslie Kolodziejcki, Janet Fischer, Alicia Duarte, Meredith Bittrich, Ronald Wilken, and The Infrastructure Group (TIG).

Beyond research, I am deeply grateful to my family and friends for their unwavering love, support, and understanding throughout these demanding years. I extend my heartfelt gratitude to my parents, grandparents, and my wife, Jingjing Chai, for their unconditioned belief in me. Among friends, my special thanks go to Zhiyu Zhang, Yuxuan Hou, Xiaoyu Sun, Xin Zheng, Zhanfu Yang, Dr. Shiyang Chen, Prof. Shucong Li, Dr. Yuhang Zhang, Prof. Hanrui Wang, Prof. Chao Liu, Dr. Guoqing Wang, Dr. Haojun Jia, Dr. Chenru Duan, Kaiyi Jiang, Dr. Zhifei Ge, Dr. Linsen Li, Dr. Wenqi Xian, Ruibei Li, Zhongpeng Sun, Dr. Chia-Kang Ho, and Prof. Jian Shu.

Last but not least, I would like to dedicate this thesis to the memory of my grandmother, a faithful and respected Christian Elder who persistently prayed for my achievements and well-being, and whose spirit guided me every step of the way.

Contents

<i>List of Figures</i>	9
<i>List of Tables</i>	11
1 Introduction	13
1.1 Formal Definition of Procedural Materials	15
1.2 Exemplar Procedural Material Systems	16
1.2.1 Adobe Substance 3D Designer	16
1.2.2 Blender	16
1.3 Thesis Structure	18
2 Related Works	19
2.1 Material Capture	19
2.2 Procedural Noise	20
2.3 Inverse Procedural Material Modeling	21
2.4 Visual Program Synthesis	21
2.5 RL Fine-Tuning	22
2.6 LLMs for Procedural Modeling	22
3 Background: Differentiable Procedural Material Graphs	23
3.1 Differentiability of Material Graphs	24
3.2 The Diffmat Library	25
3.3 The Procedural Material Capture Process	26
3.3.1 Image-based Parameter Optimization	27
3.3.2 Network-Based Graph Selection and Initialization	28
3.4 Results	28
3.5 Limitations	31
4 Proxy-Free Mixed-Integer Node Parameter Optimization	33
4.1 Method	34
4.1.1 FX-Map Theory and Implementation	35
4.1.2 Graph System for Pixel Processor	38
4.1.3 End-to-end Inverse Procedural Material Capture	39
4.1.4 Exposed Parameter Translation	40
4.2 Results	41
4.2.1 Validation and Benchmarking of Diffmat v2	41
4.2.2 Material Capture from Synthetic and Real Images	43

4.2.3	Exposed Parameter Translation	46
4.2.4	Failure Cases	47
4.3	Conclusion	48
5	Node Parameter Prediction with RL	49
5.1	Learning-Based Material Graph Generation	50
5.2	Method	51
5.2.1	Parameter Generator	52
5.2.2	Supervised Pre-Training	52
5.2.3	Reinforcement Learning by Program Execution	52
5.2.4	RL Implementation	55
5.3	Dataset	56
5.3.1	Synthetic Dataset	56
5.3.2	Real Photograph Dataset	56
5.4	Evaluation	57
5.4.1	Qualitative Results	57
5.4.2	Quantitative Evaluation	59
5.4.3	Graph Retrieval vs. Generation	61
5.4.4	Ablation Study	61
5.4.5	Alternative RL Settings	63
5.4.6	Matching Quality and Time Cost	64
5.5	Discussion and Conclusion	66
6	Material Graph Generation using VLMs	69
6.1	Method	69
6.1.1	Data Collection	70
6.1.2	Data Augmentation	71
6.1.3	VLM Fine-Tuning	73
6.1.4	Post-Optimization	74
6.2	Experiment	75
6.2.1	Evaluation Setup	75
6.2.2	Results	77
6.2.3	Ablation Study	79
6.2.4	User study	80
6.3	Discussion and Conclusion	82
7	Conclusion and Future Work	83
A	Additional Results for RL Fine-Tuning	85
B	Additional Results for VLMaterial	89
	<i>References</i>	93

List of Figures

1.1	Procedural material editing in application.	14
1.2	Thesis overview.	14
1.3	An example procedural material in Adobe Substance 3D Designer.	17
1.4	An example procedural material in Blender.	17
3.1	An illustration of procedural materials created by MATch.	24
3.2	Rendered images of 88 automatically translated procedural materials by Diffmat.	26
3.3	Overview of MATch’s fully differentiable procedural material capture pipeline.	27
3.4	Optimization results for real-world materials using MATch.	29
3.5	Comparison between MATch and Hu et al.	30
3.6	Comparison between MATch and non-procedural SVBRDF capture methods.	31
3.7	Failure cases of MATch.	32
4.1	Overview of proxy-free mixed-integer node parameter optimization.	34
4.2	Visualization of an example FX-Map graph.	35
4.3	Reproduction of FX-Map, Pixel Processor, and other generator nodes.	37
4.4	Example of a Pixel Processor node.	39
4.5	Comparison of noise generator outputs by Diffmat v2 and AS3D.	42
4.6	A gallery of differentiable procedural materials translated by Diffmat v2.	42
4.7	Additional comparisons with MATch.	45
4.8	Additional comparisons with Neural Proxies.	46
4.9	Results of exposed parameter transfer between material graphs.	47
4.10	Demonstration of failure cases.	48
5.1	Overview of the RL framework.	51
5.2	Workflow of RL implementation.	55
5.3	Qualitative evaluation on the synthetic dataset.	57
5.4	Qualitative evaluation on the real photograph dataset (top-1).	58
5.5	Qualitative evaluation on the real photograph dataset (top-5).	58
5.6	Parameter predictions using retrieved or predicted graph structures.	60
5.7	Ablation study: choice of dataset.	62
5.8	Ablation study: pre-training and loss function.	62
5.9	Comparison with alternative RL settings.	64
5.10	Qualitative evaluation with increasing random sampling budgets.	65
5.11	Trade-off between matching quality and time consumption.	65

5.12	Average length of predicted parameter sequences w.r.t. training epochs.	66
5.13	Failure cases of the RL-fine-tuned model.	67
6.1	Application of procedural materials captured by VLMaterial to a 3D scene.	70
6.2	Overview of the VLMaterial workflow.	70
6.3	Qualitative comparison of VLMaterial with baselines.	78
6.4	Average material generation quality with varied graph complexity.	79
6.5	Visual examples for ablation study.	80
6.6	Failure cases.	82
A.1	Additional qualitative results of RL fine-tuning on synthetic images.	86
A.2	Additional qualitative results of RL fine-tuning on real images.	87
A.3	RL fine-tuning on synthetic images with increasing sample budgets.	88
A.4	RL fine-tuning on real images with increasing sample budgets.	88
B.1	More qualitative comparisons with baselines.	91
B.2	More examples for ablation study.	92

List of Tables

4.1	Runtime comparison of generator nodes in Diffmat v2 and AS3D.	43
4.2	Optimization runtimes of production-grade material graphs.	44
5.1	Hyperparameter values for RL fine-tuning.	56
5.2	Quantitative evaluation on the synthetic test set.	59
5.3	Quantitative evaluation on the test set of real photographs.	60
5.4	RL fine-tuning using retrieved or predicted graph structures.	61
5.5	Ablation study: choice of dataset.	63
5.6	Ablation study: pre-training and loss function.	63
6.1	Quantitative comparison of VLMaterial with baselines.	77
6.2	The inference time consumption of our method and other baselines.	79
6.3	Ablation study.	80
6.4	Guidelines for interpreting the usability ratings.	81
6.5	User study on the practical usability of generated materials.	81
B.1	Comparison under a tighter sample budget.	90

Chapter 1

Introduction

Physics-based rendering (PBR) materials are essential for creating photorealistic object appearances in modern 3D computer graphics (CG) production, including movies, animations, video games, and professional visualization. The demand for high-quality PBR materials has increased significantly in recent years, driven by emerging applications such as building virtual environments for VR/AR, creating digital twins for training robotic AI agents, and generating synthetic data for learning world models. Conventional, image-based PBR materials encode the material attributes of object surfaces, i.e., the spatially varying bidirectional reflectance distribution functions (SVBRDFs), as pixelated texture maps. Although extensively used in commercial rendering pipelines, these materials are limited by image resolution and are not intuitively editable to generate semantically meaningful appearance variations.

As an alternative, procedural materials have become particularly popular in the CG industry due to an interpretable, controllable, and resolution-independent material representation. Specifically, they represent material SVBRDFs using generative algorithms that can be visualized as directed acyclic computation graphs, where the nodes correspond to texture generation, image processing, and mathematical operators. Users can easily edit a procedural material in industrial material authoring software (e.g., Adobe Substance 3D Designer [1] and Blender [2]) by adding or removing nodes, manipulating edge connections, and modifying the control parameters of individual nodes (Figure 1.1). However, mastering these tools requires comprehensive knowledge of texture decomposition and considerable training. Even professional artists may spend hours or even days creating a production-grade procedural material and fine-tuning a multitude of parameters to adjust intricate details.

To alleviate the intensive labor of artists, previous research has focused on inverse procedural material modeling, i.e., automatically generating a procedural material to match a photographed material appearance. The input appearance is typically assumed to be captured on a flat surface, using a cell phone flash camera, and from an upright angle. Given an input image, the inverse procedural material modeling workflow proceeds in two steps (Figure 1.2, top).

1. **Conditional material graph generation.** The first step predicts a procedural material graph with proper nodes, edges, and node parameters to perceptually match the input image. As materials with intricate details may have extremely complex graph structures, the common approach either retrieves an existing graph from a repository of artist-created materials or trains a neural network for end-to-end conditional generation.



Figure 1.1: An exemplar procedural material applied in an indoor 3D scene (left). Users can edit the material graph to change its color and texture pattern (right).

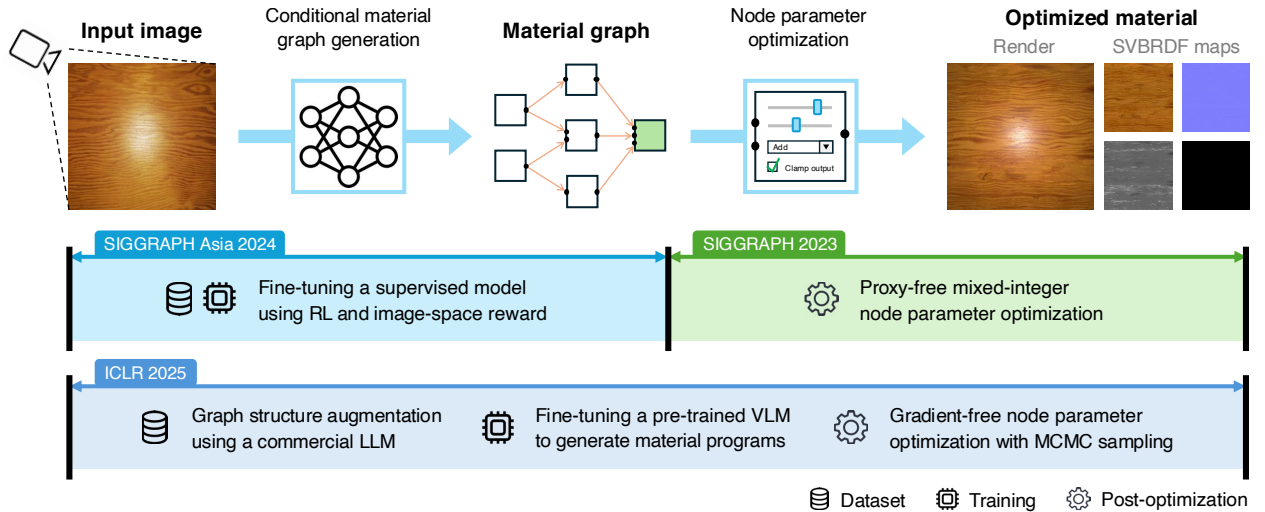


Figure 1.2: An overview of the inverse procedural material modeling workflow (top) and the projects included in my thesis (bottom).

2. **Node parameter optimization.** The second step freezes the structure of the predicted material graph and post-optimizes its node parameters to improve the match with the input image. This constitutes a challenging mixed-integer nonlinear programming (MINLP) problem due to the heterogeneity in parameter types, which involves both continuous and discrete parameters.

The state-of-the-art implementation of this workflow, however, tends to produce materials with notable mismatches in color or texture pattern when evaluated on diverse input images, especially real flash photos. The suboptimal matching quality severely undermines their practical value in accelerating the artistic creation processes of technical artists.

To fundamentally enhance the quality and robustness of generated materials, I address the limitations of existing inverse procedural material modeling approaches in dataset preparation, training methodology, and node parameter optimization. This requires resolving the following

technical challenges:

- How to approach the scarcity of high-quality artist-created procedural materials as training data for conditional material graph generation models?
- How to improve the generalization of these generative models to out-of-distribution synthetic and real input images?
- How to extend the node parameter optimization step to co-optimize both continuous and discrete node parameters, notably for advanced texture generator nodes?

My thesis investigates novel solutions to these research problems with three projects (Figure 1.2, bottom). The projects are conducted in the opposite order with gradually increasing scopes within the entire workflow.

1. **Proxy-free mixed-integer node parameter optimization.** I present a multi-stage node parameter optimization algorithm that combines gradient-based and gradient-free methods to co-optimize the continuous and discrete node parameters in a procedural material graph.
2. **Node parameter prediction with reinforcement learning (RL).** I fine-tune a pre-trained image-conditioned node parameter prediction model with RL, introducing visual feedback using an image similarity reward between the predicted material and the input image.
3. **Material graph generation using large vision-language models (VLMs).** I demonstrate procedural material generation in the programming language format by fine-tuning a pre-trained VLM on a synthetic material dataset. The dataset is augmented using a commercial large language model (LLM) for a broader variation in material graph structures.

1.1 Formal Definition of Procedural Materials

Procedural materials are inherently functional programs that produce material SVBRDFs, typically visualized as node graphs in industrial rendering software. A procedural material M is mathematically equivalent to a directed acyclic computation graph $G = (\mathcal{V}, \mathcal{E})$. Each node $v \in \mathcal{V}$ represents a parameterized texture generation, image filtering, or mathematical operator

$$x_{\text{out}}^+ = f(x_{\text{in}}^*, \phi) \quad (1.1)$$

that receives zero or multiple input arguments x_{in}^* and generates one or more outputs x_{out}^+ . The functionality of each node is additionally controlled by a set of heterogeneous node parameters $\phi = (p_1, p_2, \dots)$ that are either continuous (e.g., floats, vectors, colors, and color ramps) or discrete (e.g., integers, categories, and Booleans) by nature. Each edge $e \in \mathcal{E}$ represents the data flow from an output connector of one node to an input connector of another node. Depending on the implementation of the material graph system, the edges may carry intermediate texture images, shading models, or generic data types. Notably, adjacent nodes with several input and output connectors can be linked by multiple edges. Users can edit a procedural material via a graphical user interface (GUI) in industry-standard authoring tools by adding or removing nodes, changing edge connections, and tweaking node parameters, to produce a wide range of appearance variations.

Besides constructing material graphs from scratch, most procedural material authoring tools support hierarchical design approaches. An artist can package pre-defined subgraphs into “macro-nodes” and embed them into a larger graph to generate more intricate materials. Therefore, although a material graph system might only implement a collection of atomic nodes for basic operations, they can be combined to construct significantly more complex macro-nodes that are reusable in other material designs. In addition, some procedural materials can be semantically controlled by high-level exposed parameters, which associate groups of low-level node parameters to intuitive semantics. For example, an exposed parameter that specifies the number of annual rings in a tree trunk material may control the number of circles in a circular pattern generator node and their relative positions simultaneously. This allows artists to apply semantically meaningful changes without manipulating the corresponding low-level node parameters in a cumbersome manner.

1.2 Exemplar Procedural Material Systems

Most commercial 3D modeling and rendering platforms implement dedicated material graph systems such as Adobe Substance 3D Designer (AS3D) [1], Blender [2], Maya [3], and Unreal Engine [4], etc. In my thesis, I use procedural materials in AS3D and Blender as examples, which are described in the following subsections.

1.2.1 Adobe Substance 3D Designer

The procedural materials in AS3D consist of two types of nodes: generators and filters. Generator nodes create primitive spatial textures including noise textures (e.g., Perlin noise) and structured geometric patterns. Filter nodes manipulate input texture maps using various image processing operations ranging from per-pixel calculation (like blending and color mapping) to regional effects (like blurring, warping, 2D transformation, etc.). These nodes are parameterized by the control parameters of the functions they implement (for instance, the kernel size of a box filter or the foreground opacity in alpha blending). The graph connectivity defines a sequence of image-space operations that typically start with grayscale noise or pattern textures and eventually output a set of material SVBRDF maps after intermediate filtering. My thesis focuses on the four most commonly used material maps: albedo (or base color), normal, roughness, and metallicity. These maps specify the parameters of the simplified Disney BRDF model for real-time and offline rendering. While AS3D procedural materials generate per-pixel texture maps, the output is resolution-independent—modifying the resolution parameter automatically changes the final texture map resolution without introducing artifacts in conventional image scaling.

Figure 1.3 shows an example leather procedural material graph. This graph has 43 filter nodes and 127 node parameters with fairly convoluted connectivity, generating a wide range of photorealistic leather material appearances. Most production-grade procedural materials in AS3D are even more complex than this example, potentially exceeding 100 nodes and 1,000 node parameters.

1.2.2 Blender

Blender procedural materials utilize texture nodes to generate primitive noise or geometric patterns (such as bricks, checker, Voronoi, and fractal noise), shader nodes to specify surface properties

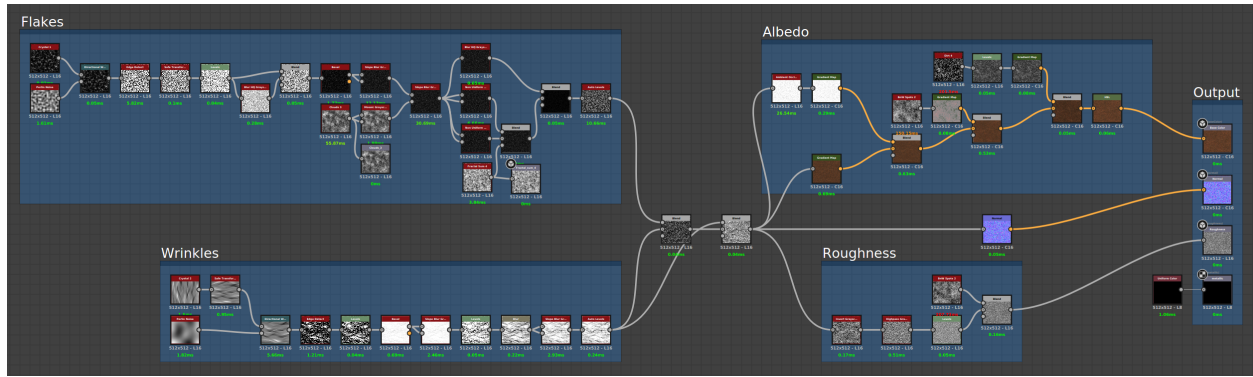


Figure 1.3: An example procedural material in Adobe Substance 3D Designer for leather appearances.

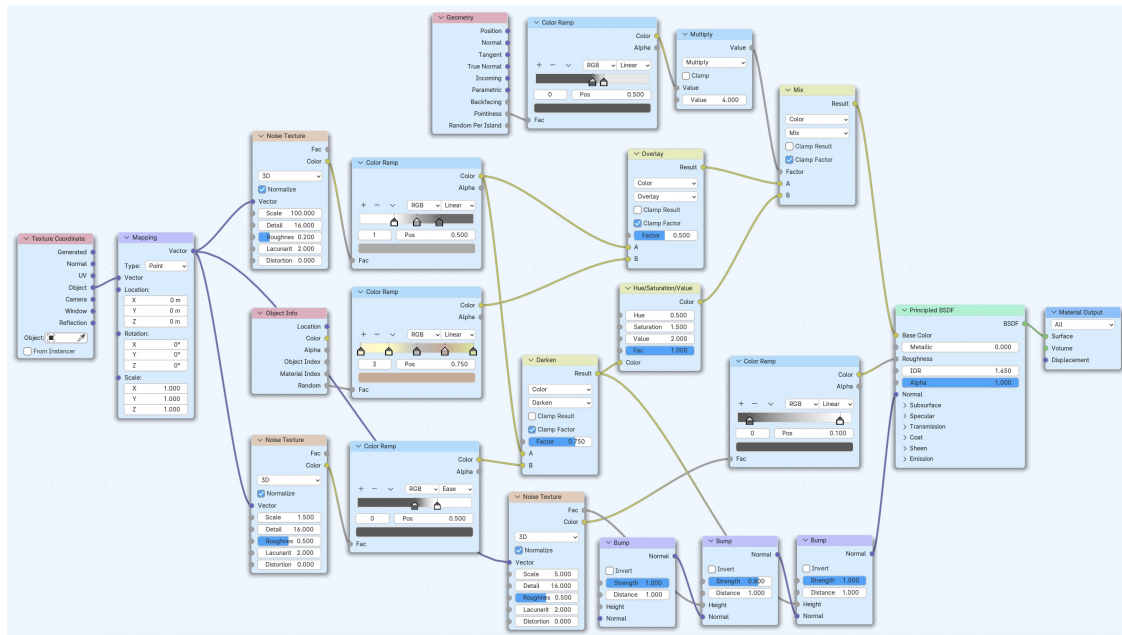


Figure 1.4: An example procedural material in Blender.

(e.g., roughness, normals, and metallic), and converter nodes for color mapping and general mathematical operations. Unlike AS3D, a material graph defines a shader program that takes object coordinates as input and directly computes the SVBRDF function instead of deriving SVBRDF maps. Therefore, similar to general-purpose programming languages, the intermediate results typically have generic data types (e.g., scalars, vectors, and shaders).

The node graph of an example Blender procedural material is shown in Figure 1.4. Blender also allows users to package node subgraphs into custom macro-nodes with user-specified inputs and outputs, referred to as node groups. Based on various node groups, Blender procedural materials can represent sophisticated appearances with compact graph structures.

1.3 Thesis Structure

The remainder of my thesis is organized as follows. Chapter 2 includes a comprehensive literature review of related works. Chapter 3 introduces a differentiable procedural material graph framework, MATch [5], as the background for all thesis projects. Then, Chapter 4 describes the proxy-free mixed-integer node parameter optimization workflow for AS3D materials. Chapter 5 demonstrates the application of RL fine-tuning in node parameter prediction for AS3D materials. Chapter 6 elaborates on the capability of adapting pre-trained VLMs to Blender procedural material generation using a carefully augmented procedural material dataset. Finally, Chapter 7 summarizes the thesis and explores several promising future directions for inverse procedural material modeling.

Chapter 2

Related Works

While my thesis focuses on inverse procedural material modeling for node-graph-based materials, the topic is closely related to material capture methods in general. The image-conditioned procedural material generation task is also an instance of visual program synthesis which extends to symbolic representations of generic visual data. From the methodological perspective, the idea of fine-tuning a model trained with supervision using RL is extensively applied in machine learning for content generation. Furthermore, there has been a rapidly growing interest in incorporating multi-modal LLMs in procedural modeling due to their emerging reasoning and coding capabilities. The following sections delve deep into the previous works in these categories, constituting a broad picture of the thesis direction.

2.1 Material Capture

Capturing the parameters of SVBRDFs is a classic problem in computer graphics, where a large body of work uses non-procedural material representations. Guarnera et al. [6] provides a comprehensive survey for this research area. We summarize the previous works below based on the capture and reconstruction method.

Appearance Capture with Specialized Hardware. Physically accurate SVBRDF capture requires dense spatial and angular sampling to disambiguate material parameters. Achieving adequate sampling at large scales often necessitates a specialized captured environment that includes multiple lights and cameras. Many prior works have devised custom capture setups, including the use of a robot arm [7], matrix light pattern (LCD, DLP projector) [8], spherical gantry [9–11], light dome [12–14], light cube [15], spherical harmonic illumination [16, 17], LED array [18–20], or moving linear light source [21, 22]. While these methods generally exhibit satisfactory reconstruction accuracy, the entailed hardware costs, human labor in environment setup, operation complexity, and acquisition time can be prohibitive for amateur users and casual purposes.

Prior-based Mobile Appearance Capture. The rapid advancement in cell phone imaging has ushered in a new era of mobile appearance capture based on casually captured flash photographs. Aittala et al. [23] first demonstrated such a system using a pair of flash/no-flash photographs and later reduced the requirement to a single photo [24]. A contemporary work by Riviere et al. [25]

used a flash video captured over a hemisphere. However, the scarcity of captured views and lighting variations in these methods demands domain-specific priors to constrain the results within a plausible solution space. These priors may require the results to be a linear combination of certain basis materials [21, 26–30], exhibit self-similarity (containing stationary spatial structures) [23], be illuminated under sparse incident lighting [31], or exist within a low-dimensional manifold [32]. Unfortunately, these assumptions do not always hold for various material families, which often compromises the complexity and fidelity of the reconstructed SVBRDFs.

Learning-based Mobile Appearance Capture. The revolutionary knowledge distillation and generalization capabilities of convolutional neural networks (CNNs) make them an attractive alternative to hand-crafted priors. Li et al. [33] trained a CNN-based SVBRDF predictor utilizing a small labeled dataset of measured SVBRDFs with self-augmentation. Ye et al. [34] extended this method to enable training with unlabelled data. Later on, the introduction of differentiable rendering layers and large-scale synthetic SVBRDF datasets allowed for per-pixel albedo, normal, and roughness predictions [35–37], as well as the joint recovery of 3D shape [38]. Alternatively, CNNs were trained to extract a low-dimensional latent space as a neural prior, enabling the test-time optimization of more plausible SVBRDFs [39–41]. For large-scale materials, Deschaintre et al. [42] fine-tuned a trained network on close-up flash photographs of the test sample to achieve better generalization across the entire material. Subsequent works continued to improve the quality and efficiency of material generation. Zhou et al. [43], Vecchio et al. [44], and Guo et al. [45] handled saturated specular highlights in reconstructed SVBRDFs using generative adversarial networks (GANs) or highlight-aware convolution. Zhou et al. [46] and Fischer et al. [47] further applied meta-learning techniques to improve material capture quality and reduce the cost of test-time optimization. The most recent advances in SVBRDF capture target challenging, in-the-wild settings [48], tileable materials [49], and controllable generation with diffusion models [50, 51]. Zhou et al. [52] published a large dataset of real material captures, bridging the distribution gap between synthetic materials for model training and real flash photos.

2.2 Procedural Noise

Procedural noise is a fundamental tool for generating realistic textures for natural materials with rich visual details [53]. It is largely classified into lattice, explicit, and sparse convolution noises [54]. Lattice noises are generated by interpolating random values/gradients defined at the grid points of an integer lattice, the most famous example being Perlin noise [55, 56]. Meanwhile, alternatives to integer lattices were proposed to mitigate axis-aligned artifacts and reduce the computational cost, such as lattice convolution noise [57] and simplex noise [58]. Explicit noises are generated in a more deterministic process [59–62]. A prominent category of explicit noises is generated through Fourier spectral synthesis, which creates noises with distinct power spectra by filtering white noise in the frequency domain [63–68]. Finally, sparse convolution noises are produced as the sum of randomly positioned and weighted kernel functions [69–72]. We refer readers to the survey by Lagae et al. [54] for a more detailed discussion.

2.3 Inverse Procedural Material Modeling

Inverse procedural appearance capture optimizes a procedural appearance model to match materials against user-specified examples. Early attempts used hand-crafted templates and rules to extract structural and local information from the source examples [73–76]. The effectiveness of these methods primarily depends on the quality of the templates, and their applications are often restricted to certain material categories. For more stochastic and isotropic texture examples, procedural noises are employed to directly match their appearances by optimizing the power spectrum in both the image and frequency domains [77–79]. Example-based noises such as local random phase noises [80], Texton noise [81], and histogram-preserving blending operator [82] were proposed to accommodate more appearance variations.

The advances of node-based procedural material authoring tools facilitated the creation of complex procedural materials. Simultaneously, these tools fostered an online community where artists can freely share and reuse high-quality assets. Such a phenomenon has inspired a new research direction that leverages artist-designed production-grade graphs to match photographed material appearances. Hu et al. [83] proposed to train a CNN per material graph, learning to regress the best parameters to match a photograph. Alternatively, Shi et al. [5] made the graph execution partially differentiable, enabling gradient descent for parameter search. The set of differentiable nodes was extended to pattern generators through differentiable proxies [84] albeit with approximation errors. Yeh et al. [85] and Yan et al. [86] further applied differentiable material graphs to capture materials from 3D indoor scenes. These methods, however, require ground-truth material graphs to be provided for training and optimization. Therefore, different methods were proposed to create variations of a flexible graph [87] using scribble-driven material segmentation and mask synthesis [88, 89]. Subsequently, Guerrero et al. [90] linearized the material graphs into token sequences and leveraged three Transformers [91] to generate graph nodes, edges, and parameters. Hu et al. [92] conditioned this generation process on text or image prompts [92], but requires differentiable node parameter post-optimization [5] to rectify the mismatches between the generated material and the input image. Most recently, Zhang et al. [93] demonstrated text-driven material graph generation for texture inpainting on 3D shapes, exemplifying the practical value of procedural material generation in 3D modeling.

2.4 Visual Program Synthesis

Our work is also related to visual program synthesis, i.e., inferring the underlying programs from input visual signals. Prior works under this topic span various domains, such as synthesizing drawing programs from images [94–98], converting raster images into SVGs [99–101], reconstructing CSG trees from 3D shapes [102–106], and inferring CAD construction programs from 3D shapes [107–110]. In contrast to some of these works that define their domain-specific language (DSL), my thesis focuses on generating procedural material programs already widely used in the 3D CG industry.

2.5 RL Fine-Tuning

RL has been used in various applications to fine-tune models initially trained with supervision. Unlike supervised training which requires ground truth targets, RL only needs a reward function to guide model training, and the reward does not have to be differentiable. Consequently, RL-based fine-tuning is less limited by available data or a need for technically convenient loss functions. This was notably demonstrated with RL from human feedback (RLHF) in InstructGPT [111], the paper that laid the foundation for ChatGPT [112]. Other LLMs followed a similar approach [113, 114] to align model outputs with human preferences. RL fine-tuning has also been applied to code generation [115–117] using different rewards based on compile errors or runtime test results.

RL fine-tuning in vision and graphics. Several tasks in vision and graphics have been shown to benefit from eliminating the need for ground truth data or differentiable score functions. Early work used RL to train a simple network that can retouch a photo in a sequence of interpretable steps [118] using an adversarial loss as the reward function. RL fine-tuning also enhanced the performance of vision tasks [119] by directly improving the quality metrics for evaluation instead of training losses. Recently, RL has been applied to fine-tuning text-to-image diffusion models [120–122] using human feedback and other non-differentiable scores. A similar approach has been used to improve multi-view consistency in 3D generation [123].

RL for procedural content generation RL demonstrated great potential in procedural content generation tasks due to the limited availability of training data. Several RL-based methods have been proposed to train procedural generators for virtual environments in education or video games [124–127]. Other methods tackle the more challenging task of generating procedures for 2D or 3D shapes that reconstruct given target outputs [102, 128] using RL. Some works propose alternatives to RL [129] but operate on relatively simplistic domains. My thesis considers industrial-grade procedural materials with a significantly more complex representation than previous works. Materials are represented as functional programs with a large number of parametric operations and convoluted dependencies. This requires careful dataset construction, a good reward definition, and a more advanced RL approach [130] using a value network for increased robustness.

2.6 LLMs for Procedural Modeling

State-of-the-art commercial LLMs like OpenAI o1 [131], LLaMA [113] and Gemini [132] have demonstrated impressive results in code generation and program synthesis [133, 134]. As a procedural modeling system essentially defines a DSL, some recent works explore procedural modeling using LLMs [135–138]. 3D-GPT [135] uses an LLM as a planner to combine pre-defined procedural generators to create a 3D scene corresponding to the given text description. BlenderAlchemy [136] leverages VLMs to iteratively refine a procedural modeling program to achieve the user’s design intent. These works all rely on the zero-shot reasoning ability of LLMs to modify existing procedural modeling programs. In my thesis, I generate procedural material programs from scratch based on the input images, discovering that state-of-the-art VLMs fail to perform this task reliably via zero-shot prompting.

Chapter 3

Background: Differentiable Procedural Material Graphs

The MATch framework [5], short for “MATerial Match”, enables a lightweight material acquisition method that automatically creates AS3D procedural materials from single captured images. While this would be extremely challenging to do from scratch, some datasets of high-quality procedural materials already exist as a starting point. Therefore, MATch converts a target RGB image (for example, but not limited to, a flash photograph taken with a cellphone) into a procedural material by identifying an appropriate procedural material graph from an AS3D asset repository¹, and estimating the node parameters of that graph to match the target appearance. In particular, MATch focuses on estimating the continuous node parameters in the filter nodes of a procedural material graph, while keeping the discrete parameters and generator nodes fixed (these typically govern the types and random seeds of primitive noises and patterns). As illustrated in Figure 3.1, this is already sufficient to reproduce a broad range of real-world materials.

MATch is an optimization-based method that treats a procedural material graph as analogous to a neural network whose “parameters” can be estimated. More specifically, the “parameters” are not actual network weights but the collection of node parameters in the graph. Such an approach requires a key element: a differentiable version of a given material graph that can be evaluated forward and backward for gradient computation. This is accomplished using a procedural material modeling library, *Diffmat*², to translate procedural material node graphs into differentiable programs. Diffmat can handle production-grade procedural material graphs representing sophisticated SVBRDFs, potentially with hundreds of nodes and thousands of parameters.

As the second author of the MATch paper [5], I was deeply involved in developing the Diffmat library and testing the method on real images. This chapter explains the methodology behind MATch and demonstrates its advantages relative to previous material capture methods. I also include a discussion about its limitations to motivate the thesis projects.

¹<https://substance3d.adobe.com/assets/allassets>

²<https://github.com/mit-gfx/Diffmat>



Figure 3.1: Every material in this rendered scene is a procedural material automatically created by MATch from a single cellphone flash photo. The corresponding target materials are visualized underneath.

3.1 Differentiability of Material Graphs

Recent work on learning-based inverse graphics has found that combining neural networks with graphics models can lead to more interpretable, and in many cases, more accurate results [35, 37, 118, 139]. However, this requires the graphics model to be differentiable for gradient back-propagation. This has prompted the development of differentiable libraries for computer vision [140], rendering [141–144], geometry processing [145], and spatially sparse computing [146]. Similarly, we use procedural node graphs as the output of material capture as they provide a compact and interpretable material representation. This would be impossible without proper tools to efficiently evaluate procedural materials (forward and backward) in an optimization framework.

Our insight is that the typical filter nodes in AS3D procedural materials consist of image processing operations closely analogous to the foundational operations in CNNs. With very few exceptions, filter nodes execute either convolutional (blurs, edge detectors, etc.) or pixel-wise operations (e.g., curve mapping, blending, quantization) to each pixel of the input map(s). This corresponds to the rudimentary components of CNNs: convolutions and point-wise activation functions (like ReLU and Sigmoid). Thus, the filter nodes in a procedural material system can be fully translated to differentiable operators using a modern machine learning framework. On the other hand, the generator nodes are not always expressible in this way. MATch freezes the parameters in generator nodes and focuses only on estimating the filter node parameters. Nonetheless, in many cases, adjusting the filter node parameters already induces fundamental changes to the material appearance (for example, from matte brown leather to glossy red leather).

Based on these observations, our implementation of the Diffmat library consists of differentiable atomic filter nodes and compound filter nodes (packaged subgraphs of atomic or other compound

nodes). Diffmat also integrates an automatic graph translator that translates complex procedural material graphs into differentiable graphs.

3.2 The Diffmat Library

Diffmat is a PyTorch-based [147] library of differentiable material node implementations that match the functionality of equivalent nodes in AS3D with up to per-pixel accuracy. Given a production-grade procedural material graph, it automatically converts the graph into a differentiable program. Then, it optimizes the node parameters in the material graph using gradient descent to match the material appearance with the input image. Specifically, Diffmat supports the following node categories.

Atomic filter nodes. Atomic filter nodes are the basic building blocks of any procedural material system. AS3D includes basic image processing operators such as Blur, Warp, Blend, Distance, Gradient Map, and gamma adjustments. Advanced nodes, like FX-Map and Pixel Processor, may execute custom, user-defined pattern generation or image filtering operations. Diffmat incorporates all 21 atomic nodes in AS3D and accurately reproduces their behaviors. As special cases, we convert atomic generator nodes that import non-procedural content, such as Bitmap, SVG, and Text, into precomputed inputs of differentiable material graphs.

Compound filter nodes. As mentioned, compound nodes are constructed using atomic filter nodes to produce specific visual effects and enable high-level artistic control over material appearance. AS3D defines more than 200 compound filter nodes, among which Diffmat implements 110 frequently used nodes. This allows Diffmat to represent a significant portion of production-grade artist-created material graphs (Figure 3.2). In particular, the inefficient calls of atomic filter nodes are fused into integral functions to improve computational efficiency.

Generator nodes with pre-cached outputs. Instead of including the generator nodes in the differentiable material graph, Diffmat precomputes their outputs using the AS3D’s command line automation tools by saving these textures as image inputs for the rest of the graph. While this simplification excludes the generator node parameters from optimization, it also avoids implementing a large set of generators with high computational costs during evaluation.

Special instances of FX-Map nodes. The FX-Map node is unique to AS3D. It allows users to repeatedly subdivide and replicate an image while applying rotations, translations, and blending. It is commonly used to create repetitive or fractal patterns, such as tiles, stripes, and various noises. As a full-fledged FX-Map node can generate extremely intricate textures, Diffmat supports special instances of the FX-Map node that tile a single input image with deterministic or random changes in position, rotation, scaling, and blending opacity.

Special instances of Pixel and Value Processor nodes. The Pixel Processor and Value Processor nodes define custom mathematical functions to manipulate per-pixel or single values. A Pixel Processor node provides shader-like control by calculating a user-specified mathematical function

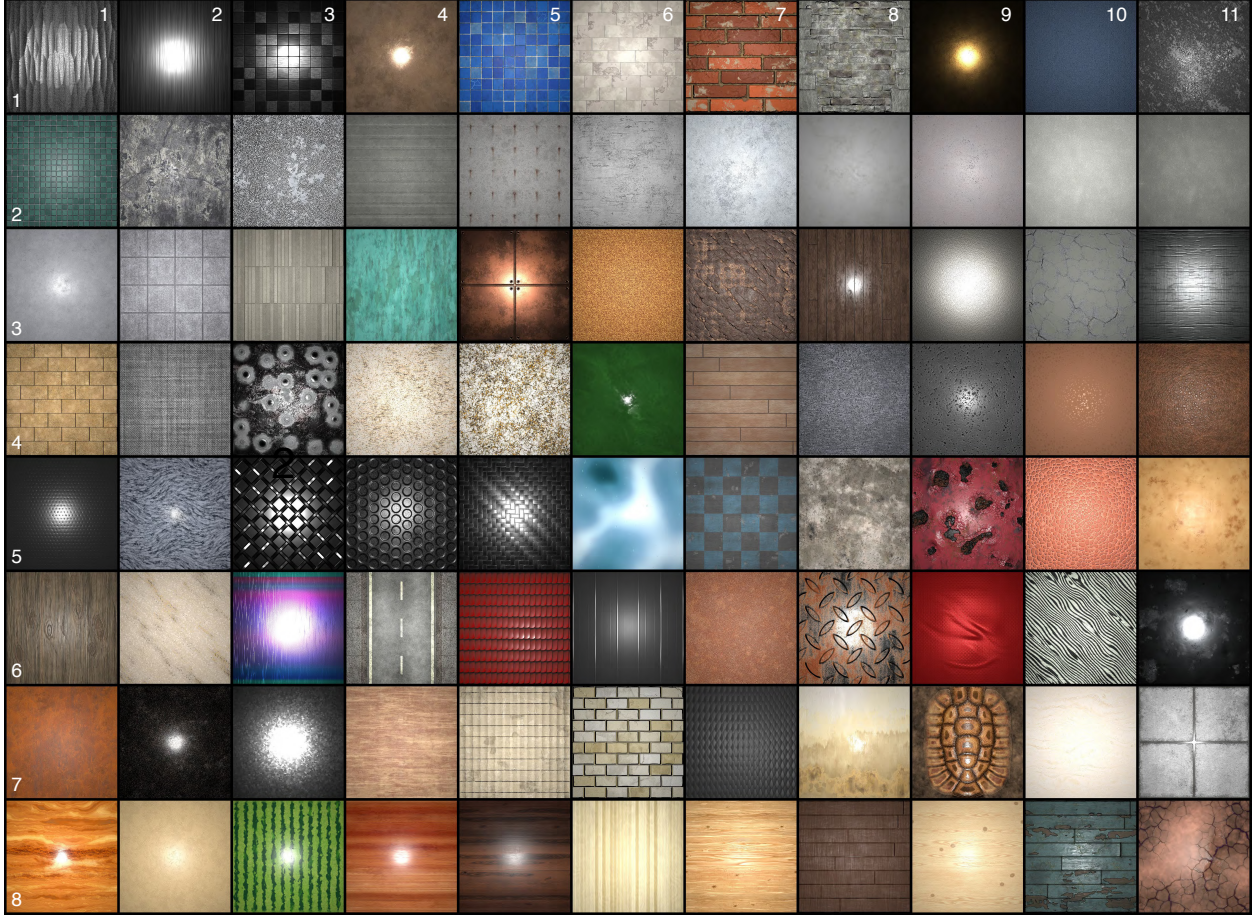


Figure 3.2: Rendered images of 88 automatically translated procedural material graphs, marked with row and column index. All images are rendered from the maps produced by Diffmat. The image resolution is 512×512 pixels.

pixel-wise, which can create advanced effects like half-toning. In contrast, a Value Processor node is typically used to convert exposed parameters into individual low-level node parameters. Like FX-Map, Diffmat includes certain instances of these nodes with non-optimizable parameters.

3.3 The Procedural Material Capture Process

Given a library of differentiable graphs, we follow a two-step process to recover a procedural material from a target photograph. First, we identify the most relevant graphs for the input photograph. We train networks to directly predict the node parameters for each graph, evaluate the error between images rendered with these predictions and the input image, and pick the top-3 graphs with the lowest errors. Then, we use gradient-based optimization to fit the parameters in each of these graphs to the target image. Our entire pipeline is illustrated in Figure 3.3.

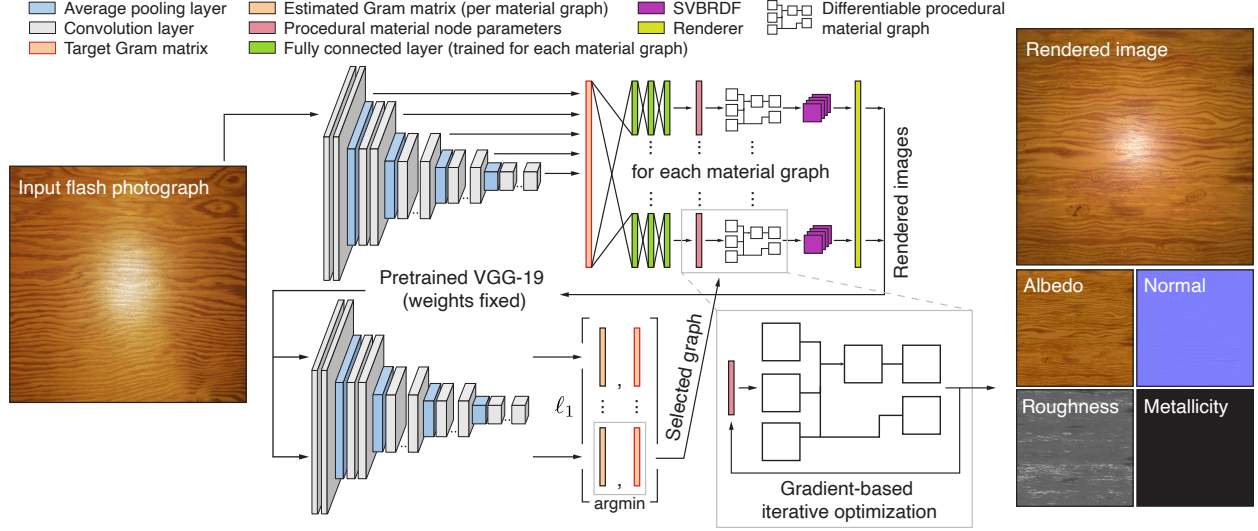


Figure 3.3: Overview of MATch’s fully differentiable procedural material capture pipeline.

3.3.1 Image-based Parameter Optimization

Given a specific translated graph G , our goal is to estimate node parameters that will produce a SVBRDF whose rendered appearance reproduces a target image, I^* . In particular, we are interested in the material parameter vector θ of length k , a concatenation of all k optimizable parameters of the node graph G .

We define the parameter map evaluation operator M that encompasses the evaluation of G . Given θ as input, it produces parameter maps of a simple BRDF model combining a microfacet and diffuse term: albedo a , normal vector n , roughness r , and metallicity m . Therefore,

$$(a, n, r, m) = M(\theta). \quad (3.1)$$

The rendering operator R takes the generated maps and computes a rendered image under known illumination. We assume a single target image captured by a co-located point light and camera from an upright angle (simulating a cellphone camera with flash). The predicted synthetic image can be written as:

$$I = R(M(\theta)) = R(a, n, r, m). \quad (3.2)$$

Note that both operators M and R are differentiable so the gradient can backpropagate through the entire expression $R(M(\theta))$.

Optimizing for θ requires us to define a loss function between the rendered image I and the target image I^* . The loss cannot rely on pixel-perfect alignment because the spatial patterns between the two images are unlikely to match exactly. We use a popular style loss function proposed by Gatys et al. [148, 149] for image style transfer, which is used by many previous methods [24, 79]. This loss function can be written as:

$$L_G = \|T_G(I) - T_G(I^*)\|_1, \quad (3.3)$$

where T_G is a Gatys texture descriptor defined by the concatenation of the Gram matrices of the five feature maps before each pooling layer of the VGG [150] network. Because Equation 3.3 is fully differentiable, we can backpropagate gradients from the loss through the rendering layer and the material graph to update the values of θ directly.

3.3.2 Network-Based Graph Selection and Initialization

The choice of the graph used to match a target image fundamentally affects the quality of reproduction. However, it is non-trivial to manually choose an appropriate graph from a large set of material graphs, since production-grade material graphs are often “over-parameterized” and can represent a wide range of material appearance beyond their semantic categories. Therefore, our method first directly predicts the parameters of each graph for the target image and compares the predictions to select the best candidate graphs. These predictions also serve as initializations to node parameter optimization and facilitate convergence to better solutions.

Similar to Hu et al. [83], we predict the material parameter vector using a neural network that takes the target image I^* and returns an estimated $\hat{\theta}$. However, instead of training an individual deep network end-to-end for each material, we extract image features using a pre-trained VGG-19 network and only retrain the multi-layer perception (MLP) regression heads. This combines the advantages of a powerful feature extractor and a small compute and memory footprint during MLP training. Moreover, the same network architecture can be applied to varying input resolutions as their VGG feature vectors have identical sizes.

We train the fully connected layer weights with synthetic data generated at training time. For each iteration, we randomly generate a ground truth material vector θ^* and evaluate the corresponding maps a^*, n^*, r^*, m^* (Equation 3.1) and the ground truth synthetic image I^* (Equation 3.2). We then predict the parameter vector $\hat{\theta}$ given I^* . Using our fully differentiable pipeline, we can generate the material maps and rendered images for the predicted parameters, namely $\hat{a}, \hat{n}, \hat{r}, \hat{m}$ and image $\hat{I}$. The loss function is a weighted combination of a parameter-space loss and an image-space style transfer loss:

$$L_{\text{net}} = \lambda_{\theta} L_{\theta} + \lambda_G L_G, \quad (3.4)$$

where the parameter loss L_{θ} is defined as the L_1 difference between material parameter vectors:

$$L_{\theta} = \|\hat{\theta} - \theta^*\|_1. \quad (3.5)$$

The style transfer loss (Equation 3.3) serves as an image-based regularizer and teaches the network how the predicted parameters influence the visual appearance of the output material. Note that this term does not appear in Hu et al. [83] as they consider the material graphs non-differentiable.

3.4 Results

We analyze the performance of MATch on diverse synthetic images and real photographs. All results are computed from a single 512×512 input image captured (or rendered) under flash illumination. For illustration, we use “Default” to denote material renderings from the original, artist-defined node parameters of material graphs, and “Target” to denote the target image whose appearance we would like to reproduce. Unless otherwise specified, the gradient-based optimization covers *all* filter node parameters in the graph. We encourage readers to zoom in on the figures to observe the matching quality of texture details.

Material capture from real images. We evaluate MATch on 30 real photographs with 27 images captured by ourselves and 3 images from the dataset of Aittala et al. [24]. Figure 3.4 (top) shows

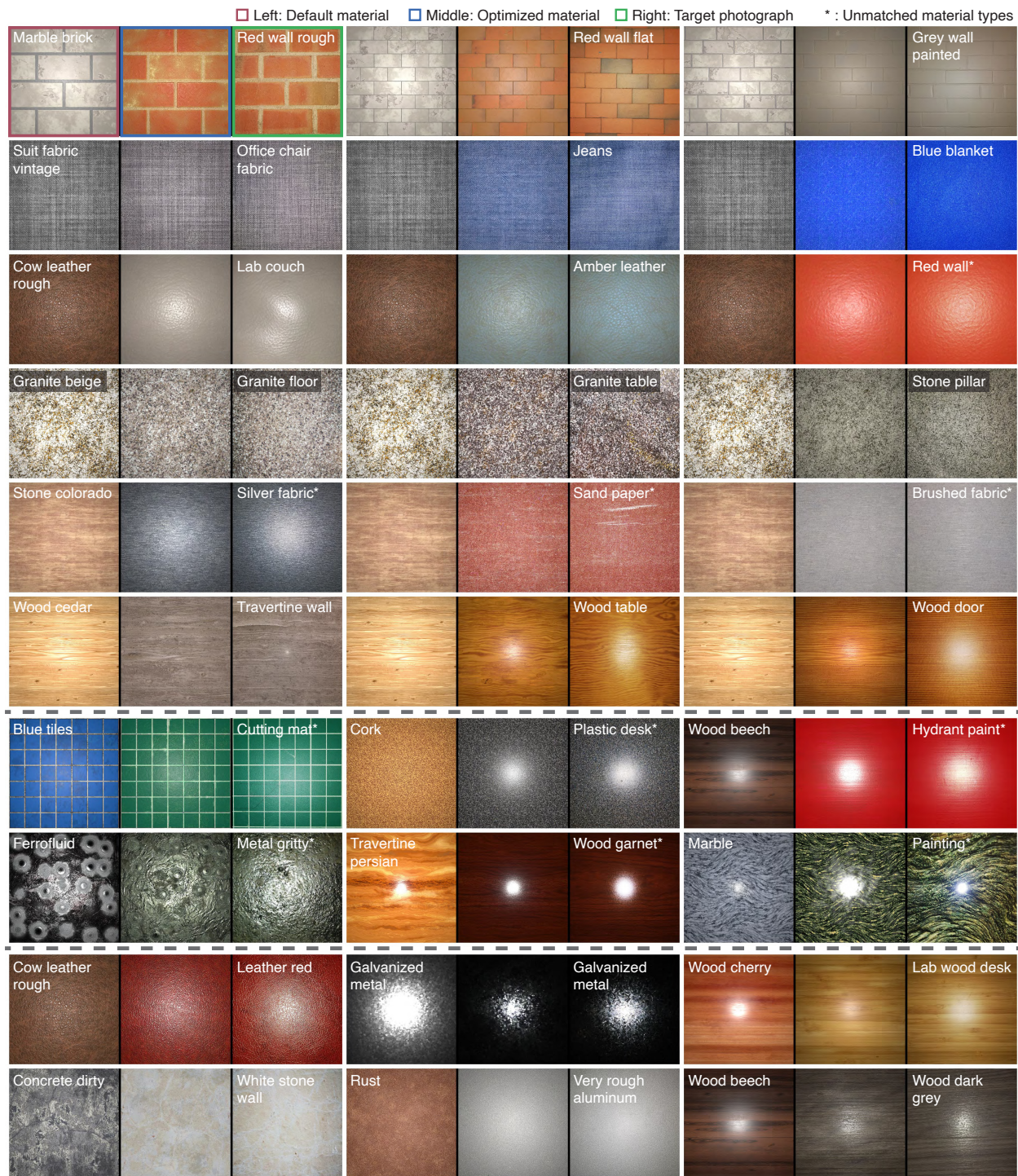
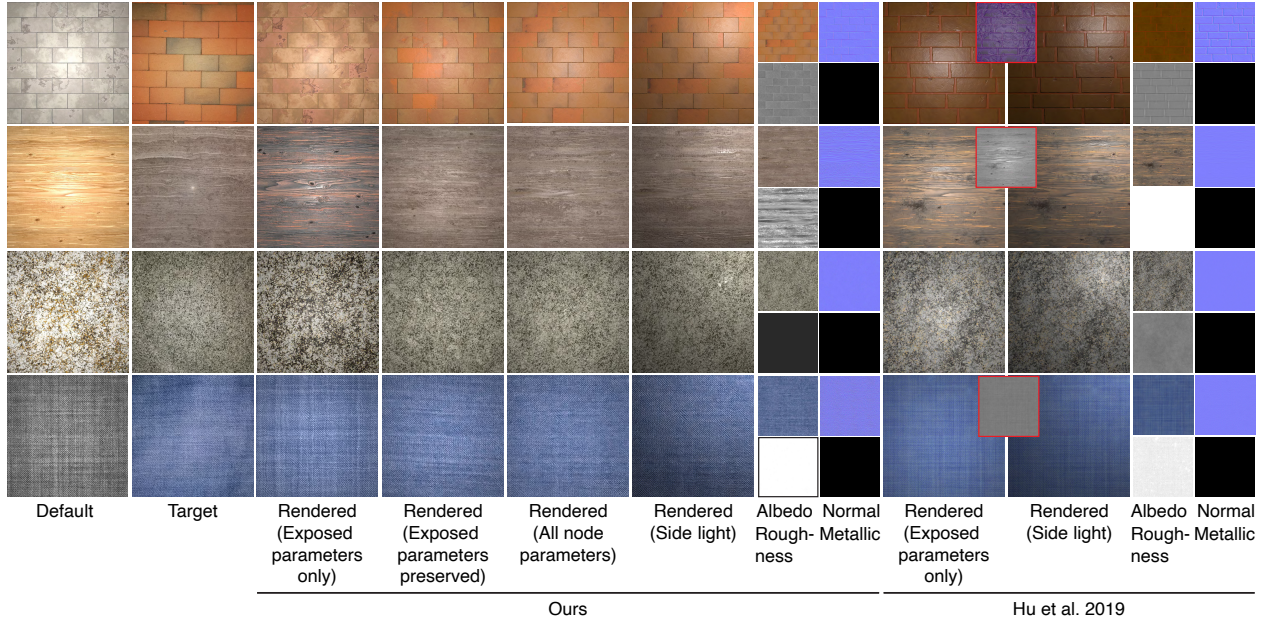


Figure 3.4: Optimization results for real-world materials. **Top:** Each row shows multiple real-world materials matched by a procedural material graph using the same initial parameters. The * symbol marks examples optimized against target materials in a different material category. **Middle:** Additional examples for cross-category material capture. **Bottom:** Additional examples for same-category material capture.



□ Prediction by Hu et al.'s network when the graph consists of highly-correlated exposed parameters (color here).

Figure 3.5: Comparison of MATch to Hu et al.'s parameter predictions (without subsequent style transfer) on real images. We show MATch optimization over exposed parameters only, exposed parameters with other independent node parameters, and all node parameters.

that we can reproduce a wide spectrum of materials starting from the same graph and initialization. This indicates the diversity of materials that can be generated by manipulating node parameters in production-grade graphs, as well as the ability of our optimization-based approach to converge to good solutions. Even when the categories of material graphs differ from the targets, MATch achieves robust matches by leveraging their adequate expressiveness (Figure 3.4, middle).

Comparison with Hu et al. In Figure 3.5, we compare MATch to the procedural material capture method of Hu et al. [83]. They use a different set of material graphs under the assumption of Lambertian BRDFs and natural illumination and only predict exposed parameters. To evaluate their approach, we retrain their AlexNet-based [151] models on our material graphs and restrict the predicted parameters to exposed parameters (except when no exposed parameter is defined). As shown in Figure 3.5, our network predictions are more accurate than Hu et al. despite a more compact size. We attribute this improvement to the image-space perceptual loss in network training, which is only possible with differentiable material graphs. MATch further optimizes the predicted parameters using gradient descent, resulting in a significant lead in matching quality. Additionally, we discover that Hu et al.'s networks have trouble disambiguating highly correlated parameters due to a lack of knowledge of the material graph. This typically happens when different exposed parameter settings create almost indistinguishable appearances, which confuses their networks while learning the inverse mapping.

Comparison with non-procedural SVBRDF capture. We also compare MATch to previous learning-based single-image SVBRDF capture methods [35, 39] in Figure 3.6. These methods can

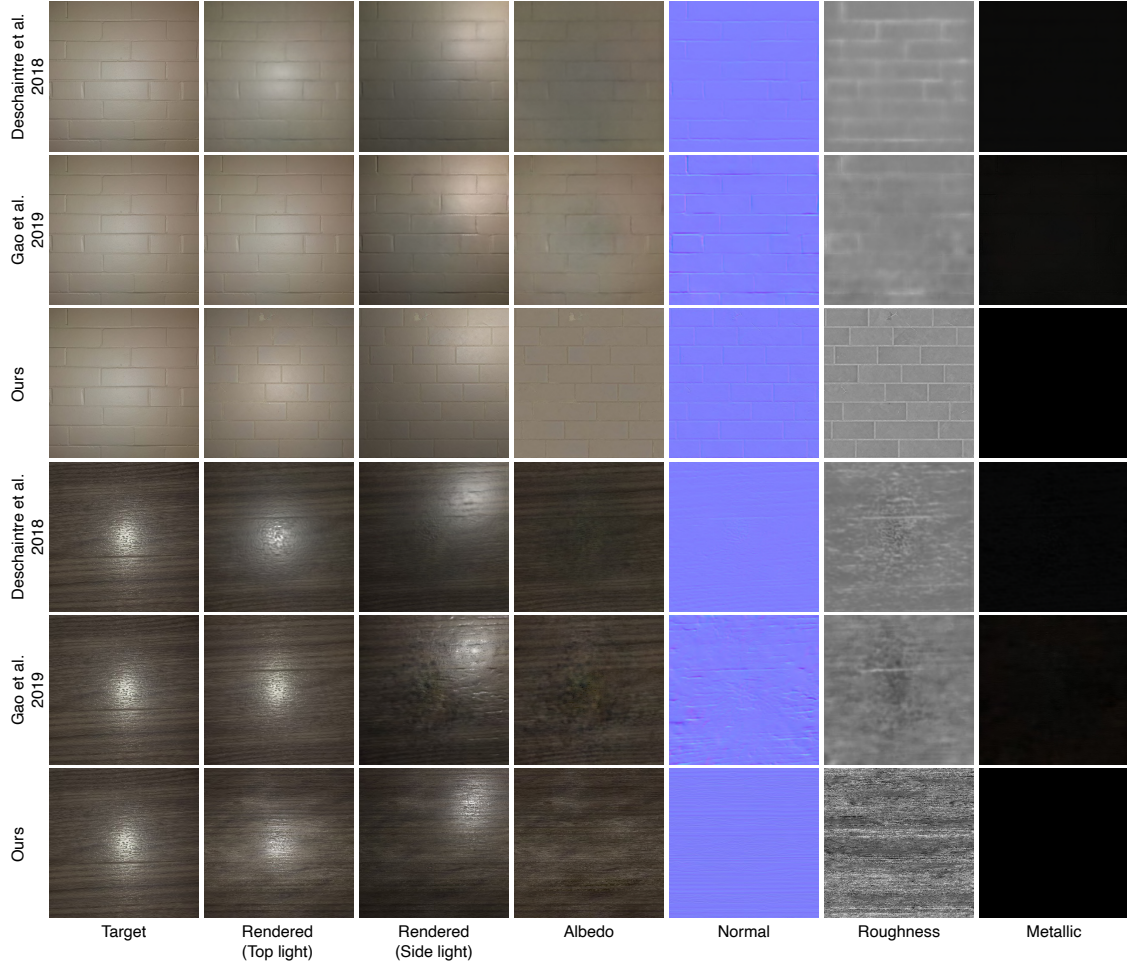


Figure 3.6: Comparison of MATch to single-image non-procedural SVBRDF capture methods. We show renderings of captured materials with default and novel lighting angles.

produce per-pixel material maps aligned with the target photo. However, due to the ill-posed nature of inverse rendering, they suffer from noticeable artifacts such as blurring, color fringing, and visual degradation with novel light positions. These methods are also limited to generating non-procedural material appearances at prespecified resolutions. Conversely, MATch produces higher-quality procedural materials that can be edited, tiled, and synthesized at arbitrary resolutions. Although the procedural materials may not perfectly match the target texture patterns, this is acceptable in most material capture scenarios where users want to capture the general appearance of the material instead of replicating the input image. In conclusion, procedural material capture has, and will always have, its trade-offs compared to image-based materials.

3.5 Limitations

Since MATch does not optimize the graph structure, its success ultimately relies on the expressiveness of the chosen graph. Consequently, it will likely fail when the input image contains intricate patterns that exceed the material graph’s modeling capability. For example, the first failure example in

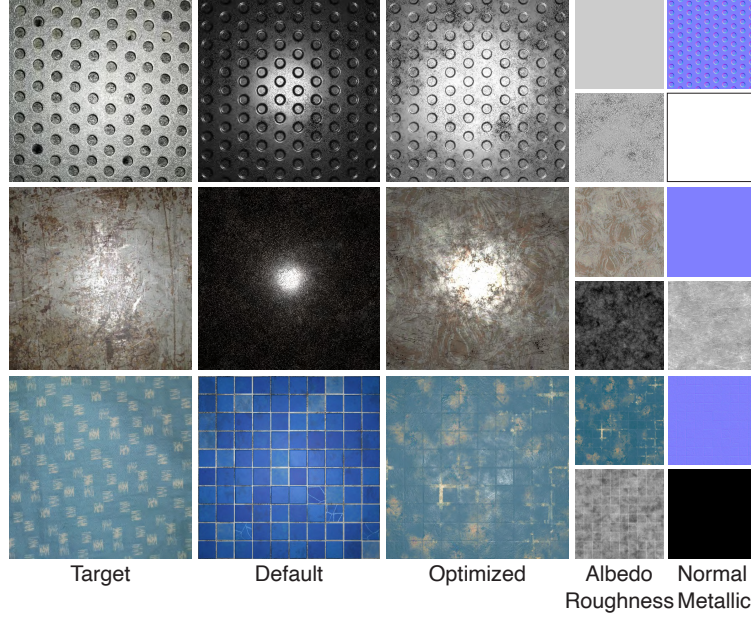


Figure 3.7: Failure cases due to unmodeled objects (Row 1), limited expressiveness of overly simple graph structures (Row 2 and 3), and inability to optimize the generator nodes to match the input pattern (Row 2 and 3).

Figure 3.7 shows a metal grate material where the material graph does not model the substrate behind the input metal grate. Another failure mode of optimization may originate from mismatched input noise patterns. Figure 3.7 Row 3 demonstrates a slightly exaggerated example, where a tile graph is used to reproduce a bed sheet with scattered square patterns. Since the tile pattern is a pre-cached input image, MATch can not adjust the layout of individual tiles, resulting in severe visual artifacts in the final result. Finally, Diffmat does not support full-fledged FX-Map, Pixel Processor, and Value Processor nodes. Thus, it can not translate graphs that utilize them as filter nodes (with input connections), which precludes optimizing certain materials with intricate random structures, e.g., grass grounds.

The next chapter will introduce my follow-up work to MATch featuring an upgraded Diffmat v2 library. The new library enables optimization for generator nodes and advanced atomic nodes, significantly improving the matching quality of the optimized materials.

Chapter 4

Proxy-Free Mixed-Integer Node Parameter Optimization

The MATch framework, developed by Shi et al. [5], translates material graphs into differentiable programs for optimizing continuous filter node parameters using gradient descent. Hu et al. [84] further extends this approach by training differentiable neural network approximations for generator nodes, enabling the joint optimization of continuous and discrete node parameters. Despite some successful examples, these inverse methods struggle to scale well against the vast spectrum of production-grade procedural materials due to the absence of accurate, computationally efficient noise/pattern generator nodes. MATch relies on pre-cached generator node outputs using AS3D’s proprietary tools, which leads to failure when initial and target textures do not match in spatial scaling. Neural Proxies [84] can optimize generator nodes but suffers from several fundamental limitations, including long training times, inaccuracies, fixed resolutions, and poor generalization outside training data. Therefore, their method has only been demonstrated on a narrow range of materials containing one or two supported generator nodes.

To address the inadequacies of current differentiable procedural material capture approaches, we introduce Diffmat v2¹, an advanced differentiable procedural material modeling library with three key additions to Diffmat [5]: generic differentiable noise generators, stochastic pattern generators, and custom differentiable pixel shading functions. Diffmat v2’s dedicated node graph system for noise and pattern generation accurately reproduces the FX-Map node in AS3D, a complex generator node that serves as the core building block for almost all procedural noises and patterns. Our reproduction achieves near-per-pixel accuracy, where the only difference lies in random number generation. The underlying implementation is carefully engineered to provide runtimes comparable to AS3D (see Table 4.1). Moreover, Diffmat v2 includes a specialized node graph system for creating custom differentiable pixel shaders, corresponding to AS3D’s Pixel Processor node. Thanks to its functional enhancements, Diffmat v2 stands as the first self-contained library capable of automatically translating production-grade procedural materials into equivalent differentiable programs without relying on AS3D’s proprietary tools.

Diffmat v2 substantially enriches the collection of existing materials that can be fully translated and optimized (Figure 4.1, right). More importantly, it enables the co-optimization of continuous and discrete (integer) generator node parameters without neural network approximations. In contrast,

¹<https://github.com/mit-gfx/diffmat>

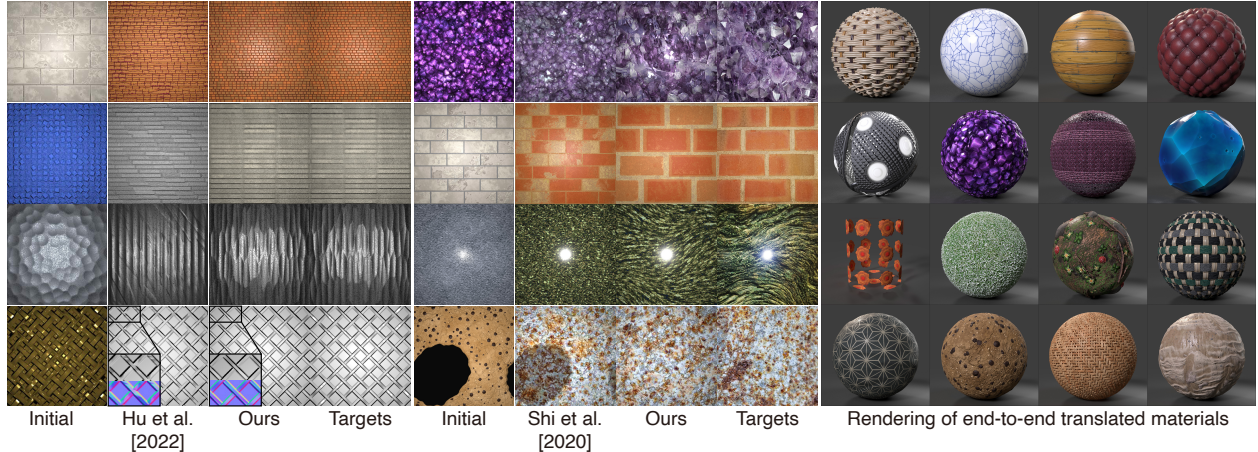


Figure 4.1: Comparison of single-image procedural material capture with Neural Proxies [84] (left) and MATch [5] (middle). See Section 4.2.2 for a detailed discussion. (Right) Our new differentiable procedural material library, Diffmat v2, translates artist-created production-grade procedural materials into differentiable programs without relying on proprietary automation tools. See Figure 4.6 for additional examples.

Neural Proxies requires training a new model for any change in texture resolution, categorical node parameters, or default parameter ranges. On the other hand, our analytical implementation produces pixel-matching results irrespective of resolution and parameter settings while computing accurate gradients during backpropagation. Harnessing these advantages, we propose an end-to-end procedural material capture pipeline that combines gradient-based and gradient-free approaches to automatically optimize the entire set of node parameters in a prespecified material graph (e.g., obtained from nearest neighbor search [5]).

Compared to previous works [5, 84], our method captures a perceptually closer appearance to the target photo across diverse material categories (Figure 4.1, left and middle). Additionally, we demonstrate the translation of high-level exposed parameters from a reference graph to a target graph for streamlined post-editing. In summary, our work incorporates new infrastructure and algorithms to enable more robust inverse procedure material capture and intuitive post-editing, offering artists enhanced support throughout the entire creative process.

4.1 Method

This section introduces the new computational infrastructure, Diffmat v2, and our end-to-end inverse procedural material capture pipeline. We first describe the theory and differentiable implementation of FX-Map (Section 4.1.1) and Pixel Processor (Section 4.1.2). They are fundamental to replicating sophisticated generator and filter nodes in AS3D. Next, we discuss how to update continuous and discrete node parameters without neural network approximations by alternating between gradient-based and gradient-free optimization (Section 4.1.3). Finally, we introduce an efficient approach to translating exposed parameters from a reference material to a target material (Section 4.1.4), providing artists knobs for high-level semantic material editing.

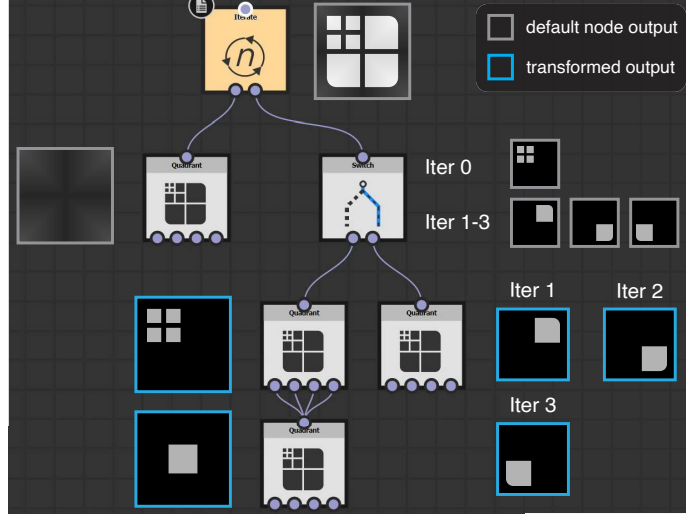


Figure 4.2: Visualization of an example FX-Map graph that combines Quadrant, Iterate, and Switch nodes to generate an approximated FX-Map icon, where the output pattern of each node is shown next to its position. A Quadrant node can apply an optional 2D transformation to its output (in blue frames). In addition, some nodes generate patterns that vary between iterations since their behaviors are contingent on the iteration number.

4.1.1 FX-Map Theory and Implementation

The FX-Map node, according to AS3D, is the most powerful and complex node that enables the procedural creation of texture images. Functionally, FX-Map represents a generalized noise generation procedure that recursively subdivides the texture canvas into smaller regions and renders atomic texture patterns in these regions. The subdivision follows a quadtree structure with a canvas area evenly split into four squares in the next quadtree layer. The 2^{2d} squares at the d -th quadtree layer constitute an octave of depth d . The rendering step draws one or multiple atomic patterns at the center of each square (d, i, j) ($i, j \in \{0, 1, \dots, 2^d - 1\}$). Each atomic pattern $F_{d,i,j,k}(x, y)$ (k is the pattern index; x, y are pixel coordinates) undergoes a user-defined spatial transformation $\xi_{d,i,j,k}[\cdot]$ that can be either deterministic or random. Deguy and Benassi [152] showed that the FX-Map procedure achieves a close approximation of intermittent locally self-similar Gaussian processes (ILSSGP), of which fractional Brownian motion (fBM) is a special case. On a $2^M \times 2^M$ canvas $\mathbb{C}$ with a max octave depth D , the FX-Map output $I(x, y)$ is defined by the following recursive formula (x, y are omitted in image-level calculation for simplicity)

$$T_{d,i,j}(x, y) = \sum_{k=1}^{R_{d,i,j}} \alpha_{d,i,j,k} \cdot \xi_{d,i,j,k}[F_{d,i,j,k}](2^d x - 2^{M-d} i, 2^d y - 2^{M-d} j) \quad (4.1)$$

$$S_{d,i,j} = \begin{cases} T_{d,i,j} & d = D \\ T_{d,i,j} + \sum_{u,v \in \{0,1\}} T_{d+1,2i+u,2j+v} & 0 \leq d < D \end{cases} \quad (4.2)$$

$$I = S_{0,0,0} \quad (4.3)$$

Here, $T_{d,i,j}(x, y)$ blends the $R_{d,i,j}$ rendered atomic patterns at square (d, i, j) , where the intensity of each pattern is parameterized by $\alpha_{d,i,j,k}$. $S_{d,i,j}$ denotes the partial texture after blending all atomic patterns in the sub-quadtrees of square (d, i, j) . Under this formulation, FX-Map’s construction process is conceptually equivalent to the depth-first traversal of the quadtree:

1. Starting from the root ($d = 0$), every quadtree node generates $R_{d,i,j}$ atomic patterns $F_{d,i,j,k}$, each transformed by $\xi_{d,i,j,k}[\cdot]$ and rendered on the canvas with intensity $\alpha_{d,i,j,k}$.
2. Enumerate four child nodes and traverse their subtrees recursively. Return to the parent node upon completion.
3. Repeat (2) until returning to the root node.

If the same pattern function F is used for all atomic patterns and $R_{d,i,j} \equiv 1$, FX-Map effectively constructs an fBM noise by reusing the same noise map (with scaling and shifting) in all octaves instead of resampling the noise with different parameters at each octave.

The FX-Map node in Adobe Substance 3D Designer is designed using a dedicated quadtree graph system comprising three specialized nodes: *Quadrant*, *Iterate*, and *Switch*. A *Quadrant* node generates an atomic pattern and renders the pattern onto the canvas under a user-defined transformation function. This node also subdivides the rendering region into four identical quadrants, each corresponding to one child node at the next quadtree layer and accessible via an input connection. Moreover, AS3D can construct a directed multigraph from the FX-Map quadtree by collapsing identical *Quadrant* nodes at the same depth into a single node. As a result, a chained *Quadrant* node sequence is equivalent to a full quadtree during top-down traversal, allowing for the generation of highly intricate texture maps. Building on the quadtree structure defined by *Quadrant* nodes, *Iterate* and *Switch* nodes specify repetition and branching behaviors during quadtree traversal, respectively. An *Iterate* node accepts two inputs, first drawing the left subtree on the canvas, and then rendering the right subtree repeatedly for a user-specified number of iterations. Introducing randomness to the right subtree further increases texture complexity since the randomness applies to each iteration individually. In contrast, A *Switch* node renders either subtree based on a custom Boolean decision function. This node is commonly used to prune unwanted subtrees or alternate between branches during quadtree traversal. In Figure 4.2, we show a simple FX-Map instance consisting of all three node types. Intermediate results produced by each node are placed aside to visualize node behaviors.

Without access to AS3D’s proprietary codebase, we developed an in-house implementation of the FX-Map graph system using PyTorch [147] and the Taichi programming language [153] after thoroughly comprehending its functionalities. As with AS3D, our implementation includes full-featured versions of *Quadrant*, *Iterate*, and *Switch* nodes. Diffmat v2 accurately reproduces AS3D’s output for FX-Map and other noise/pattern generator nodes built upon various FX-Map instances, with only minor differences in random number generation (see Figure 4.3). Our implementation reorganizes the FX-Map construction procedure into two phases: pattern generation and rendering. In pattern generation, the FX-Map quadtree is traversed depth-first through all three types of nodes. Upon visiting each node, we first compute the node parameters by fetching constant values from a look-up dictionary or evaluating Value Processors associated with dynamically-valued parameters. Next, we execute the node’s function based on its type. Each *Quadrant* node produces a pattern rendering *job* parameterized by spatial transformation, color intensity, and blending option.

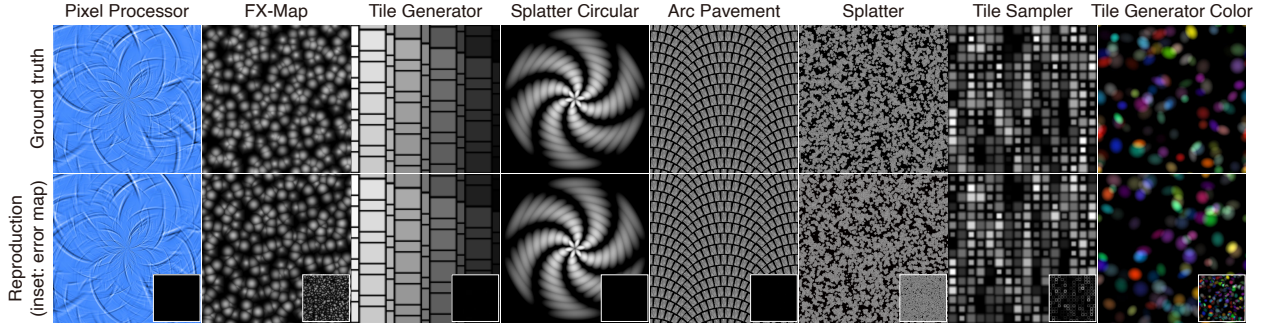


Figure 4.3: Comparison of Diffmat v2’s reproduced FX-Map, Pixel Processor, and other pattern generator node outputs with AS3D (ground truth). An error map is visualized at the bottom right corner of each reproduction. All pattern generator nodes (Column 3-8) are created from various FX-Map instances. Diffmat v2 achieves per-pixel match in nodes that do not implement any randomness or whose randomness is disabled by node parameters (Column 1, 3-5). Otherwise, our reproduction is perceptually identical but pixel-wise different (Column 2, 6-8) because AS3D’s pseudo-random number generation algorithm is non-public.

Rendering jobs are collected in a global job queue in the order of their timestamps upon creation. The rendering phase begins once the traversal is complete. During this phase, an FX-Map renderer sequentially transforms atomic patterns in the job queue and draws them on the canvas. This results in the final FX-Map image.

Besides adhering to the default FX-Map algorithm, our system implements various FX-Map nodes using different acceleration methods. This allows us to avoid the exponential scaling in computational cost due to the sequential nature of FX-Map traversal and rendering. Specifically, Our implementation classifies FX-Map nodes into *generic* and *specialized* nodes.

Generic FX-Map nodes are custom, user-defined FX-Maps that potentially enact any supported behavior. There is little room for parallel optimization since these nodes may specify a complicated random process that only allows for sequential execution, especially if Set/Sequence function nodes are extensively used to introduce data dependency. Thus, we follow the two-phase sequential approach. In the pattern generation phase, the FX-Map graph is traversed depth-first with a manual stack to track the state of each node, e.g., how many child nodes have been visited. Pushing a node into the stack represents recursion to the next octave. Popping the top node from the stack is equivalent to returning to the octave above. We also update the cumulative branch offset (a displacement effective to all atomic patterns in the subtree) and a variable dictionary containing all named variables accessible via *Get* function nodes (for referencing external parameters) in the process. Each Quadrant node submits a pattern generation job to a global rendering job queue, which sorts the atomic patterns by type and timestamp. In the rendering phase, we invoke the FX-Map renderer to batch individual pattern generation jobs into a job array. The renderer transforms these patterns in parallel and draws them onto respective regions of the canvas.

Specialized FX-Map nodes constitute most predefined noise/pattern generator nodes in AS3D, such as Tile Generator, Splatter, and Make It Tile Patch nodes. We devise corresponding acceleration strategies from their prescribed implementations to maximize data parallelism. For example, the

Make It Tile Patch node (abbreviated as MITP) makes an arbitrary input image tileable by stamping it around repetitively on an FX-Map canvas with deterministic or stochastic transformations. While an MITP node can generate thousands of atomic patterns, it only renders a single FX-Map octave, with several node parameters controlling randomness in pattern transformations. As there is no dependency among consecutive patterns, we compute all atomic patterns in parallel rather than using the job queue mechanism for generic FX-Maps. This eliminates the performance bottleneck of traversing the FX-Map quadtree. Our engineered implementation using PyTorch and Taichi [153] achieves a 300× speedup over Diffmat. Additionally, many computation-heavy stochastic noise generators in AS3D, like BnW Spots 3 and Clouds 2, only define an integer “scale” parameter and a continuous “disorder” parameter that functions like a random seed. As neither parameter is optimizable by gradient descent, these nodes do not require differentiable implementations. Diffmat v2 implements non-differentiable versions of these specialized FX-Map nodes using Taichi, leading to comparable or superior performance to AS3D (Table 4.1).

4.1.2 Graph System for Pixel Processor

Pixel Processor is a generic node-graph-based per-pixel shader in AS3D, where the user defines a function graph that computes a numerical output for each pixel. Unlike most atomic filter nodes that only conduct prescribed operations, a Pixel Processor node can execute mathematical functions of arbitrary complexities based on rudimentary numeric and logical operators. Therefore, many advanced filter nodes are represented by Pixel Processors, such as the Sobel operator, geometric warping, Cartesian-to-polar coordinates conversion, and even ray tracing. Figure 4.4 demonstrates a Pixel Processor that computes a pixel-wise scalar function to emboss a grayscale image. AS3D defines diverse function graph nodes for value definition and manipulation, and we refer readers to the official documentation² for a full catalog.

Despite the ability to parse function graphs in AS3D, Diffmat can not execute them in parallel per pixel. Thus, Diffmat v2 compiles AS3D’s function graph representation into an equivalent PyTorch program with unified support of scalar and per-pixel execution. The generated Python program is automatically differentiable whenever it references a continuous parameter or an intermediate texture map that receives gradient. Specifically, the compiler automatically translates each function node into a standard Python operation or a PyTorch routine using statically inferred operand data types (Figure 4.4, right). If a function node has mismatched input data types (e.g., a tensor and a scalar/vector), the compiler performs explicit type casting so that the operands are compatible with the referenced operation. Therefore, our function graph parser can translate Pixel Processors and Value Processors in a unified framework.

Diffmat v2 also implements many missing function nodes in Diffmat, such as Sampling nodes and Set/Sequence nodes. Sampling nodes are an essential part of Pixel Processor since they allow access to input images, without which Pixel Processor nodes can only be used as a per-pixel value generator and lose filtering capabilities. In contrast, Set/Sequence nodes are frequently used to export variables between function graphs, similar to Python’s global variables. They are indispensable in FX-Map nodes for implementing sequential control like finite-state machines. Consequently, various structured patterns can be generated with a simple Iterate node, such as tiles, weaves, and circular splatters. The additional function nodes are a cornerstone for successfully reproducing

²<https://helpx.adobe.com/substance-3d-designer/home.html>

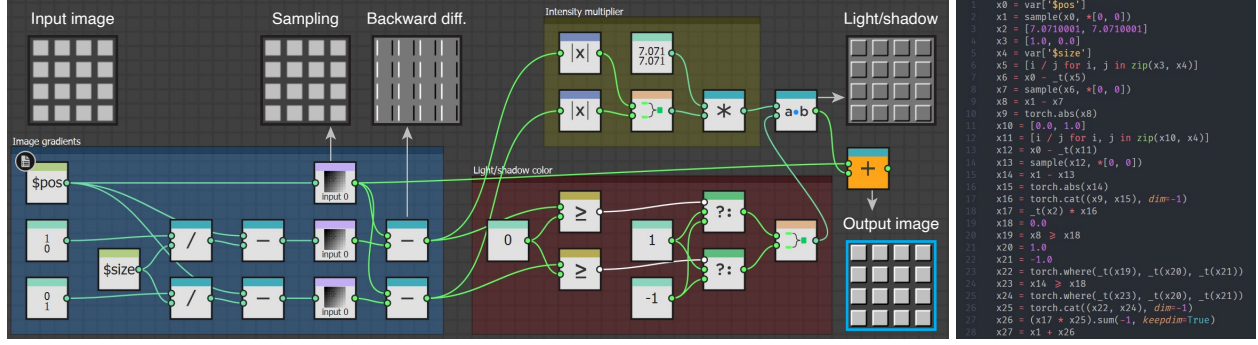


Figure 4.4: A simplified directional emboss filter implemented using Pixel Processor, which applies a simulated lighting effect to a grayscale height map. The function node graph is organized by colored frames to showcase groups of nodes that constitute an integral functionality. Intermediate results at crucial nodes are visualized, where images with negative values are normalized to $[0, 1]$. Diffmat v2 compiles the function graph into Python code sequence on the right side of the figure, where per-pixel instructions are automatically generated by referencing PyTorch API.

many noise generators in AS3D (see Figure 4.5).

4.1.3 End-to-end Inverse Procedural Material Capture

Our material capture pipeline augments MATch with integer parameter optimization. Given a user-provided or algorithm-selected material graph, let its continuous parameters be $\theta = (\theta_f, \theta_g, \bar{\theta}_g)$, where $\theta_f, \theta_g, \bar{\theta}_g$ are the vectors of differentiable continuous parameters in filter nodes, differentiable continuous parameters in generator nodes, and non-differentiable continuous parameters in generator nodes, respectively. Similarly, let its discrete (integer) parameters be $\phi = (\phi_f, \phi_g)$, where ϕ_f and ϕ_g are the vectors of discrete parameters in filter nodes and generator nodes, respectively. Finally, we set $e = (e_\theta, \bar{e}_\theta, e_\phi)$, where e_θ represents exposed, differentiable continuous parameters, $\bar{e}_\theta$ represents exposed, non-differentiable continuous parameters, and e_ϕ represents exposed discrete parameters. Similar to Hu et al. [84], our optimization consists of three stages.

Stage 1: Gradient-based initialization. In this stage, we adjust the continuous parameters to approximately match the global albedo and align the geometries if the graph includes affine transformation nodes. When the initial material appearance deviates significantly from the target, optimizing integer parameters directly may not yield an ideal solution. This step aims to reduce the appearance gap and facilitate convergence to a better solution in the next stage. Our framework allows the user to determine the granularity of optimization by stopping the gradient for θ_f, θ_g , and e_θ , respectively. This is crucial for maintaining predefined material structures (e.g., preventing undesired randomness in uniform tile or weave patterns) or preserving the interpretability of optimized parameters (e.g., confining the optimization to exposed parameters). Without loss of generality, we assume all differentiable continuous parameters are optimizable. The optimization is given by

$$\theta_f^{(1)}, \theta_g^{(1)}, e_\theta^{(1)} \leftarrow \arg \min_{\theta_f, \theta_g, e_\theta} \| \mathcal{T}(I(\theta, \phi, e)) - \mathcal{T}(I) \|, \quad (4.4)$$

$\mathcal{T}$ is a texture descriptor operator that computes the Gram matrix of extracted VGG19 features [154]. $\mathcal{I}(\cdot)$ denotes the rendered texture image under an input node parameter configuration. I refers to the target photo. $\|\cdot\|$ represents L1 loss. We run no more than 2000 iterations of gradient descent.

Stage 2: Gradient-free mixed integer optimization. After gradient-based initialization, we employ gradient-free methods to optimize integer parameters ϕ , continuous generator node parameters $(\theta_g, \bar{\theta}_g)$, and exposed parameters e . We optionally include a portion of differentiable continuous parameters here because they either control input noises/patterns alongside discrete parameters (i.e., θ_g) or specify semantic constraints among node parameters (i.e., e). The user may choose to ignore them to reduce the granularity of optimization.

Mathematically, this stage constitutes a mixed-integer non-linear programming (MINLP) problem that entails NP-hard combinatorial optimization [155]. Since the concatenation of a procedural material graph, a differentiable rendering layer, and a neural network texture descriptor creates an exceedingly sophisticated objective landscape that precludes analytical simplifications [156], we turn to generic mixed-integer optimizers such as Bayesian optimization (BO), simulated annealing (SA), and genetic algorithm (GA). BO is computationally expensive when optimizing numerous continuous and integer parameters in a production-grade material graph. SA and GA are hindered by poor sample efficiency and may produce implausible material appearances due to aggressive random exploration. To balance exploration, stability, and computational cost, we ultimately opt for a coordinate descent approach based on the intuition that most node parameters affect relatively orthogonal properties (e.g., color, size, and offset) and can be optimized independently. Our method alternates between individual parameter optimization using linear grid search. Specifically, we cycle through all active parameters in each epoch and update each parameter while holding others constant. We use a search step of 1 for integer parameters and 0.01 for normalized continuous parameters within their default parameter ranges. Unlike Neural Proxies, which generalizes poorly beyond training data, our method allows the user to reconfigure the search ranges arbitrarily. The gradient-free optimization is guided by a weighted loss function

$$\phi^{(2)}, \theta_g^{(2)}, \bar{\theta}_g^{(2)}, e^{(2)} \leftarrow \arg \min_{\phi, \theta_g, \bar{\theta}_g, e} \|\mathcal{T}(\mathcal{I}(\theta, \phi, e)) - \mathcal{T}(I)\| + w_{\mathcal{F}} \|\mathcal{F}(\mathcal{I}(\theta, \phi, e)) - \mathcal{F}(I)\| \quad (4.5)$$

where $\mathcal{F}$ is the 2D Fast Fourier Transform (FFT) operator, and $w_{\mathcal{F}}$ is the corresponding weight. The FFT loss is sensitive to structural changes and helps identify desired values for integer parameters. The optimization typically converges within five epochs.

Stage 3: Gradient-based fine-tuning. With a closer estimation for integer parameters from the second stage, we revert to gradient-based optimization (using the same formula as Equation (4.4)) to fine-tune the differentiable continuous parameters. We optionally repeat stages 2 and 3 to improve the result further, but in most cases, the process converges in no more than two repetitions.

4.1.4 Exposed Parameter Translation

Procedural materials are not always created with labeled subgraphs and exposed high-level parameters. These graphs are challenging to reuse because artists have to deduce the purpose of low-level nodes and manually modify their parameters. However, many other graphs within the same material

category may have well-defined exposed parameters. We present a straightforward yet effective method for distilling and translating continuous exposed parameters from a reference graph to a target graph. Our method stems from a fundamental insight—the semantics of an exposed parameter are characterized by its impact on material appearance. Therefore, an exposed parameter e_r in a reference graph G_r should induce the same perceptual change in the output texture after being migrated to a target graph $G_t \neq G_r$. This premise enables us to pinpoint the node parameters in G_t that correlate the most with e_r by matching texture descriptor changes.

Let $\theta_r(e_r)$ denote the vector of continuous node parameters in the reference graph controlled by e_r . We first grid-sample e_r in G_r to compute a set of Gatys texture descriptors (the texture rendering function is denoted by $\mathcal{I}_{G_r}(\cdot)$). Assuming that all continuous parameters have a normalized domain $[0, 1]$, we use a step size of $\delta = 1/N$ and compute the i -th texture descriptor $\mathcal{T}_{G_r}^{(i)}$ using

$$\mathcal{T}_{G_r}^{(i)} = \mathcal{T}(\mathcal{I}_{G_r}(\theta_r(e_r = i\delta))) \quad (i = 0, 1, \dots, N) \quad (4.6)$$

The appearance change in the reference graph from $e_r = 0$ to $e_r = i\delta$ is then derived by

$$\Delta\mathcal{T}_{G_r}^{(i)} = \mathcal{T}_{G_r}^{(i)} - \mathcal{T}_{G_r}^{(0)} \quad (4.7)$$

Our method transfers these texture descriptor changes to G_t through node parameter optimization. Here, we reuse the notations above to define θ_t and $\mathcal{I}_{G_t}(\cdot)$ for G_t . Additionally, θ_t^* denotes G_t 's default parameter values. The following problems are solved to match the impact of e_r in G_r at each sampled value ($i = 0, 1, \dots, N$):

$$\hat{\theta}_t^{(i)} = \arg \min_{\theta_t} \left(\left\| \mathcal{T}(\mathcal{I}_{G_t}(\theta_t)) - \mathcal{T}(\mathcal{I}_{G_t}(\theta_t^*)) - s\Delta\mathcal{T}_{G_r}^{(i)} \right\| + \mathcal{L}_{\text{reg}}^{(i)} \right) \quad (4.8)$$

where s is a hyperparameter amplifying (or attenuating) the transferred texture descriptor difference. $\mathcal{L}_{\text{reg}}^{(i)}$ is a regularization term that compares θ_t against the previous best solution $\hat{\theta}_t^{(i-1)}$ to encourage a smooth transition in material appearance

$$\begin{aligned} \mathcal{L}_{\text{reg}}^{(i)}(\theta_t) &= 0 & (i = 0) \\ \mathcal{L}_{\text{reg}}^{(i)}(\theta_t) &= \lambda \left\| \mathcal{I}_{G_t}(\theta_t) - \mathcal{I}_{G_t}(\hat{\theta}_t^{(i-1)}) \right\| + \eta \left\| \mathcal{T}(\mathcal{I}_{G_t}(\theta_t)) - \mathcal{T}(\mathcal{I}_{G_t}(\hat{\theta}_t^{(i-1)})) \right\| & (i > 0) \end{aligned} \quad (4.9)$$

where λ and η are the weights for texture image and texture descriptor losses, respectively.

Finally, we select the top K node parameters in G_t based on the magnitude of texture descriptor difference (between $\hat{\theta}_t^{(N)}$ and θ_t^*) contributed by each node parameter. This step discerns the essential node parameters underlying the transferred exposed parameter e_r . For each selected parameter k , we extract its optimized value sequence from $\hat{\theta}_t^{(i)}$ ($i = 0, 1, \dots, N$) and fit an M -th degree polynomial function $\mathcal{P}_k(e_t) = \sum_{m=0}^M \beta_{k,m} e_t^m$ from e_t using least squares, where $\beta_k = \{\beta_{k,0}, \dots, \beta_{k,M}\}$ is the set of polynomial coefficients for the k -th selected parameter. These polynomial functions can be easily constructed using AS3D's function graphs.

4.2 Results

4.2.1 Validation and Benchmarking of Diffmat v2

We validate the reproduction quality of Diffmat v2's new modalities by comparing them side-by-side with AS3D results. Figure 4.3 compares the output of custom FX-Map and Pixel Processor instances

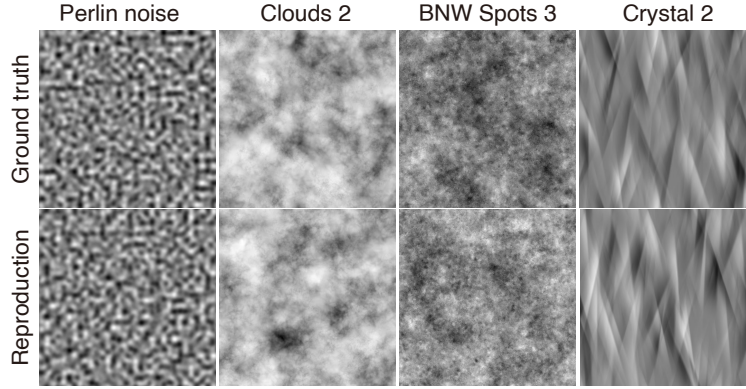


Figure 4.5: Comparison of noise generator outputs by Diffmat v2 and AS3D (ground truth). We analyze the statistics of noise usage in the Adobe Substance 3D Assets Library and visualize four representative noises as examples. Owing to the utilization of a non-public pseudo-random number generator in AS3D, our reproduction is statistically indistinguishable from the ground truth, though not pixel-perfect.

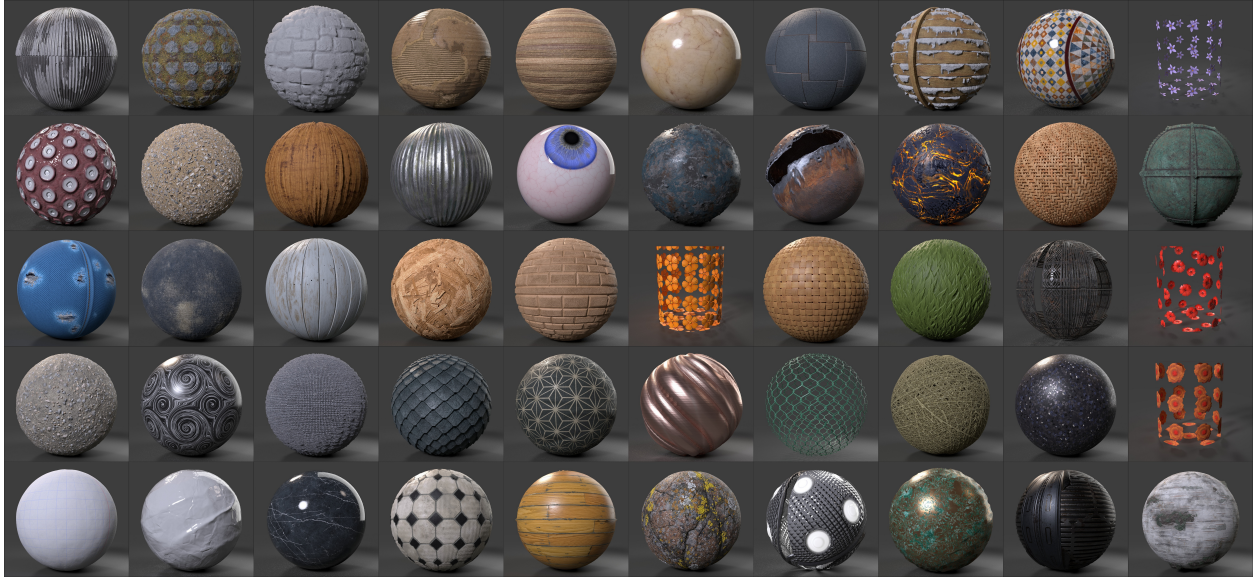


Figure 4.6: A gallery of 50 differentiable procedural materials end-to-end translated from artist-authored material graphs in Adobe Substance 3D Designer using Diffmat v2. Most materials exhibit regular structures (e.g., bricks, tiles, waves, and polygons) or complex, stochastic patterns (e.g., foliage, flower, and irregular geometries). All images are rendered using SVBRDF texture maps generated by Diffmat v2 at a 1024×1024 resolution.

and other pattern generator nodes; Figure 4.5 compares a set of frequently used noise generators. Diffmat v2 matches the ground truth precisely in nodes that do not implement randomness or whose randomness is disabled by node parameters. The reproductions of nodes with internal randomness are statistically identical to the ground truth but not pixel-perfect since AS3D uses non-public pseudo-random number generators.

We use Diffmat v2 to translate artist-created production-grade procedural materials across

Table 4.1: Runtime comparison of generator nodes in Diffmat v2 and AS3D at different texture resolutions. Our benchmark includes forward and backward evaluation times for several frequently used generator nodes, excluding backward evaluations for nodes that do not require gradient backpropagation (i.e., those defined without differentiable continuous parameters). In all cases, Diffmat v2 demonstrates comparable or even faster runtimes than AS3D, enabling seamless integration into commercial-grade software without sacrificing performance. For the Make It Tile Patch node, Diffmat v2’s FX-Map supersedes Diffmat’s hard-coded implementation, resulting in a 300× speedup. Due to the substantial computational cost associated with the StyleGAN2 (> 25M parameters) network, a neural proxy takes 11 ms to evaluate at a 256×256 resolution, with the runtime scaling linearly (i.e., 44 ms at 512×512). In contrast, Diffmat v2 has a fixed kernel launch cost from PyTorch and Taichi, with the runtime scaling sub-linearly.

Node Name	Requires Gradient	256×256		512×512		1024×1024	
		Diffmat v2 (F/B)	AS3D	Diffmat v2 (F/B)	AS3D	Diffmat v2 (F/B)	AS3D
Make It Tile Patch	Y	8 ms / 5 ms	11 ms	10 ms / 6 ms	13 ms	22 ms / 19 ms	13 ms
Tile Generator	Y	8 ms / 6 ms	2 ms	8 ms / 9 ms	2 ms	16 ms / 16 ms	4 ms
Brick Generator	Y	5 ms / 4 ms	16 ms	5 ms / 4 ms	16 ms	6 ms / 5 ms	18 ms
Stripes	Y	3 ms / 1 ms	6 ms	3 ms / 2 ms	6 ms	4 ms / 5 ms	7 ms
Arc Pavement	Y	6 ms / 3 ms	2 ms	6 ms / 4 ms	2 ms	8 ms / 5 ms	2 ms
Splatter	Y	7 ms / 4 ms	6 ms	7 ms / 4 ms	6 ms	12 ms / 8 ms	5 ms
Perlin Noise	N	5 ms / –	8 ms	6 ms / –	9 ms	7 ms / –	13 ms
Crystal 2	N	6 ms / –	3 ms	6 ms / –	5 ms	8 ms / –	15 ms
Clouds 2	N	11 ms / –	16 ms	13 ms / –	16 ms	15 ms / –	20 ms
BnW Spots 3	N	15 ms / –	24 ms	17 ms / –	25 ms	20 ms / –	26 ms

various categories without using AS3D’s proprietary automation tools. Figure 4.1 (right) displays a selection of 16 examples (see Figure 4.6 for the full collection). Most of them contain highly structured or complex stochastic geometries enabled by Diffmat v2’s functional additions. The removal of external dependency saves a substantial amount of disk storage since input noises and patterns are generated on the fly rather than pre-cached into images.

Furthermore, efficient forward and backward evaluations are necessary for interactive editing and rapid inverse capture. Through extensive performance engineering (see Section 4.1.1), we ultimately achieve runtimes comparable to AS3D for every supported generator node in Diffmat v2 (see Table 4.1 for a comparison using selected generator nodes). Forward and backward evaluation speeds at the graph level are reported in Table 4.2.

4.2.2 Material Capture from Synthetic and Real Images

We demonstrate the effectiveness and superiority of our method by comparing it with state-of-the-art solutions on both synthetic and real-world targets.

Comparison with MATch [5]. To ensure a fair comparison, we preprocess material graphs that cannot be fully translated by Diffmat using the following steps: 1) we replace unsupported generator nodes with pre-cached bitmaps using AS3D; 2) we also pre-cache unsupported filter nodes (including FX-Map s and Pixel Processor s) and freeze their predecessor subgraphs during optimization. The results are shown in Figure 4.1 (middle) and Figure 4.7.

Table 4.2: Optimization runtimes of production-grade material graphs. We show the time consumption per forward and backward iteration at various image resolutions. As a reference, Diffmat takes 1,659 ms to run forward evaluation of the Office Carpet material (the last row in the table) at a 512×512 resolution, and 1,853 ms for backward evaluation. These runtimes are 17× slower than Diffmat v2.

Graph Name (Position)	256×256		512×512		1024×1024	
	Forward	Backward	Forward	Backward	Forward	Backward
Brick Wall (Figure 4.1, Row 1 Col 1)	105 ms	51 ms	108 ms	54 ms	117 ms	89 ms
Concrete Tiles (Figure 4.1, Row 2 Col 1)	21 ms	13 ms	18 ms	20 ms	24 ms	60 ms
Aluminium Cut (Figure 4.1, Row 3 Col 1)	28 ms	16 ms	28 ms	25 ms	41 ms	50 ms
Metal Grill (Figure 4.1, Row 4 Col 1)	22 ms	14 ms	20 ms	25 ms	40 ms	168 ms
Amethyst (Figure 4.1, Row 1 Col 2)	52 ms	40 ms	60 ms	52 ms	102 ms	129 ms
Marble (Figure 4.1, Row 3 Col 2)	44 ms	15 ms	46 ms	18 ms	53 ms	68 ms
Bitten Cookie (Figure 4.1, Row 4 Col 2)	150 ms	84 ms	150 ms	87 ms	188 ms	214 ms
Weave and Knit (Figure 4.7, Row 1 Col 1)	160 ms	114 ms	199 ms	144 ms	412 ms	609 ms
Nature Ground (Figure 4.7, Row 3 Col 2)	264 ms	186 ms	299 ms	235 ms	479 ms	758 ms
Office Carpet (Figure 4.7, Row 4 Col 2)	68 ms	51 ms	95 ms	106 ms	227 ms	434 ms

In Figure 4.1 (middle), the top three results demonstrate our ability to jointly optimize continuous and discrete parameters. Although the initial integer parameters result in mismatched texture scaling, our method manages to recover the correct scales. The last example contains a binary switch that controls a cookie material’s “bites” feature. When retargeting the material at a rusty metal image, MATch can only lighten the bitten area using filter nodes since the hole is encoded in a pre-cached bitmap. Instead, our method directly turns off the “bites” switch during integer parameter optimization, yielding a solid rusty metal texture. A similar example is the removal of a forest ground material’s “skull” feature at Figure 4.7 Row 3 Column 3 (abbreviated as R3C3 hereafter). Additional examples of scale correction via integer parameter optimization can be found in Figure 4.7 R5/R6/R7C1/R8C2. Figure 4.7 R1 shows that our approach can match two distinct fabric photos with the same source material. MATch merely adjusts the color and slightly tweaks the yarn substrate and embroidery threads. In contrast, our method completely alters the fabric geometry by 1) R1C1: changing the substrate yarns into reflective brown-green wool swirls and completely removing the horizontal threads; 2) R1C2: transforming sector-shaped thread patterns into square shapes and tinting them with blue and gold colors. Meanwhile, our method flattens the rounded-shaped wood plank at R2C1 to resemble the target, and reconstructs the normal map of the porous rocks at R2C2 more accurately. Other relevant cases include R4C2 (the intricate microfiber detail of the carpet) and R8C1 (removed scattered small dents on the crystal surface). Besides geometries, our method also more closely aligns with the target image in other material properties, such as roughness (see Figure 4.7 R3C1/R6C2/R7C2). In summary, our method’s improvements are thoroughly assessed among a wide range of material categories, including fabric, leather, concrete, ground, wood, stone, porcelain, plastic, organic, and translucent materials.

Comparison with Neural Proxies [84] Our method captures the target more faithfully without being restricted to a predetermined resolution or parameter range (see Figure 4.1 (left) and Figure 4.8). Before delving into the comparison, it is important to note that most material graphs used in the MATch comparison contain over ten generator nodes not provided by Hu et al. [84]. Consequently,

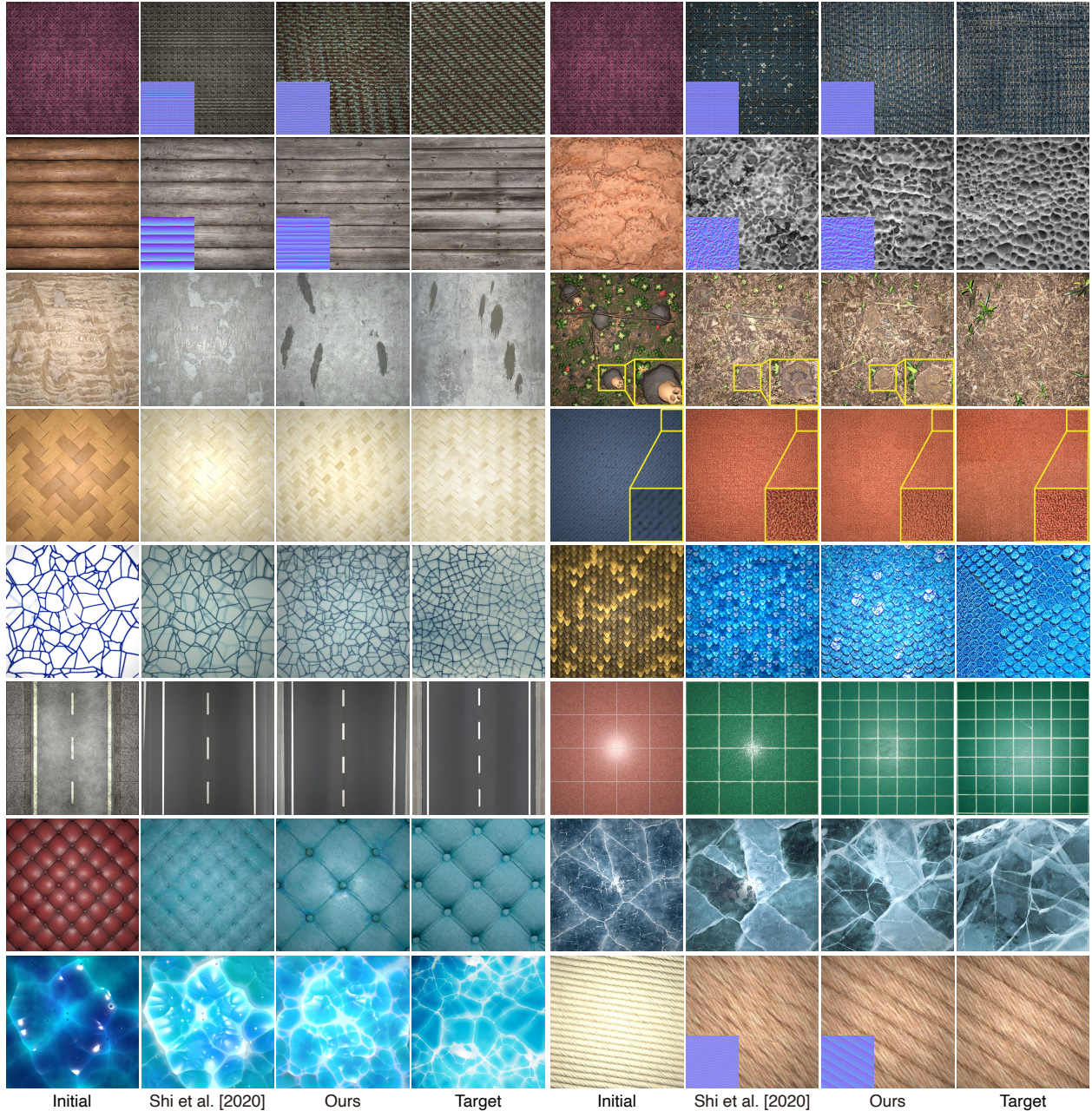


Figure 4.7: Additional comparisons with MATch [5]. For each example, we optionally display the optimized normal map at the bottom left or provide a zoomed-in view of a specific region of interest. Please refer to Section 4.2.2 Paragraph 1 for a detailed discussion.

their approach is currently inapplicable to these examples. Furthermore, testing these examples poses out-of-memory risks, as each neural proxy already occupies $\sim 1500\text{MB}$ GPU memory during forward evaluation at a 256×256 resolution. As such, we reuse the original graphs and results in Hu et al.[84] as a baseline. In Figure 4.1 (left) R1, the target brick wall has a greater number of rows and columns than the training data of the brick generator network. Neural Proxies thus fails to converge to the correct numbers. Our analytical brick generator node implementation accommodates

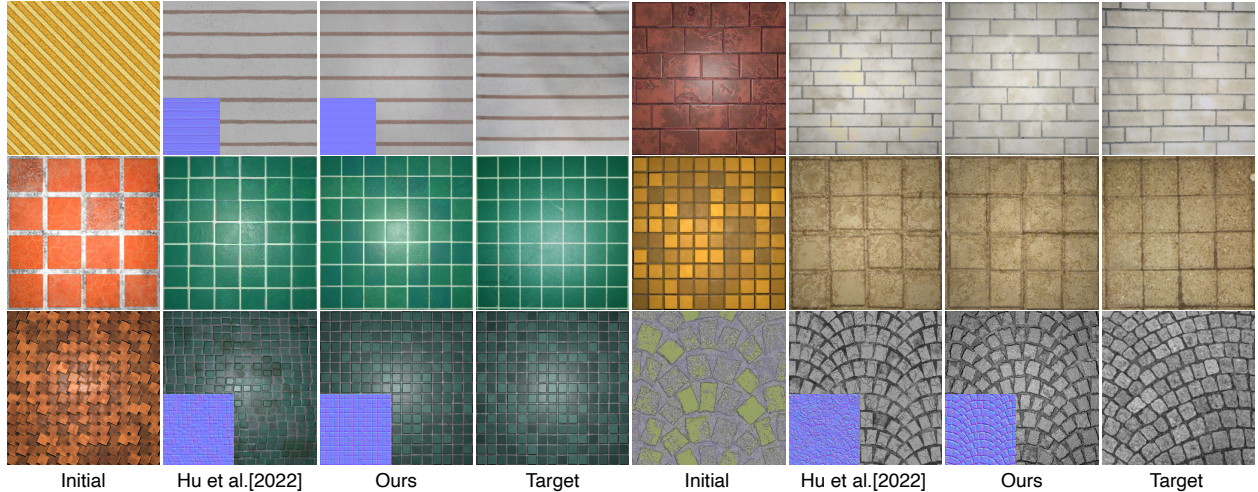


Figure 4.8: Additional comparisons with Neural Proxies [84]. The optimized normal maps are visualized at the bottom left for some examples. Section 4.2.2 Paragraph 2 provides a detailed discussion.

arbitrary parameter combinations, allowing the grid search to find a satisfactory match in two epochs. Neural Proxies is also susceptible to local minima as the deep neural network approximation creates a complicated loss landscape. The reproduction quality is often compromised when the initial guess is visually distant from the target (Figure 4.1 (left) R2 and R3). Our method, on the contrary, conducts gradient-free optimization to escape such local minima and find better solutions. Additionally, the approximative nature of Neural Proxies introduces further errors when exporting optimized parameters back to AS3D. These errors typically materialize as structural artifacts (e.g., incorrect fluctuations in the normal at Figure 4.1 (left) R4 bottom left) or inconsistent SVBRDF maps (e.g., misaligned normal and albedo maps at Figure 4.8 R3C2). Of equal importance, optimizing procedural materials at a high image resolution is often required to capture fine details in the target photo. Nevertheless, Neural Proxies can only operate in predetermined resolutions once trained. Diffmat v2’s resolution-agnostic implementation ensures the accurate reproduction of the finest details (as illustrated by the dirt details at Figure 4.8 R2C2). Throughout the comparison experiment, we observed that the limitations of Hu et al. [84] frequently result in suboptimal solutions (e.g., incorrect integer parameters in Figure 4.8 R1C2/R2C1/R3C1), while our method exhibits superior robustness.

4.2.3 Exposed Parameter Translation

Figure 4.9 illustrates the translation of various exposed parameters from a reference material graph to a target material graph. These exposed parameters alter details in albedo (R1), normal (R2), color tone (R3), and material structure (R4). We note that the example in R4 is feasible entirely due to Diffmat v2’s differentiable generator nodes, which modify the geometry of the steel material. Our pixel-matching implementation effectively transfers these geometric changes to the target graph.

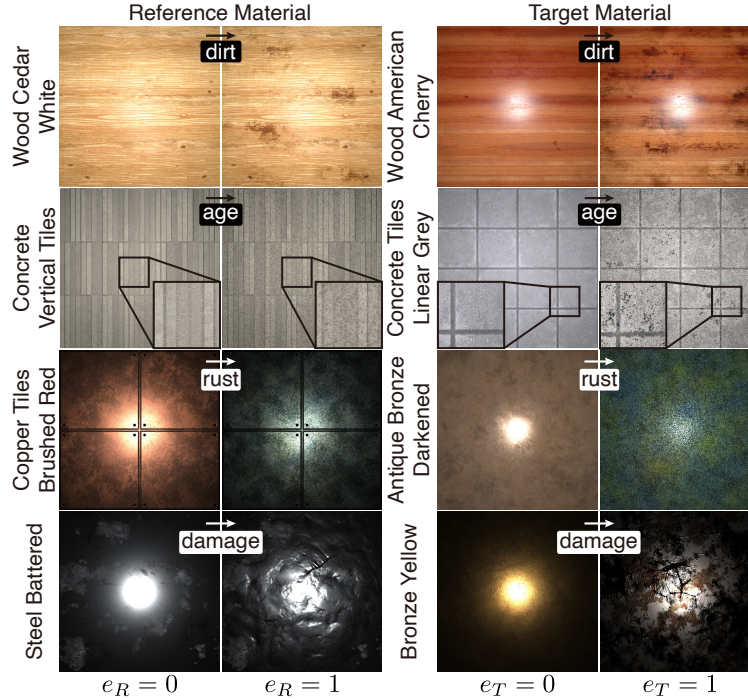


Figure 4.9: Results of exposed parameter transfer between material graphs. The labeled arrow between each image pair indicates the name of the exposed parameter. We direct readers to the Supplementary Videos for transition animations when varying exposed parameters from 0 to 1. A detailed analysis of the results can be found in Section 4.2.3.

4.2.4 Failure Cases

Figure 4.10 showcases two instances where Diffmat v2 fails to match procedural materials against synthetic or real photographs successfully. In the first example, we attempt to optimize a complex, overparameterized mixed weave material whose node graph contains a large number of integer parameters. Our method is unaware that many integer parameters are implicitly correlated to preserve the weave structure. This material graph does not define any exposed parameter as explicit constraints, so it is incorrect to assume that their effects are orthogonal and optimize each parameter independently. As a result, our coordinate descent method gets trapped in local minima and disrupts the weave structure. A workaround for similar overparameterized cases is to define exposed parameters that specify underlying constraints, thereby reducing the search space. However, this requires an artist’s effort as Diffmat v2 can only automatically transfer continuous exposed parameters from similar graphs. The second failure case, also reported by Hu et al. [84], involves the Gatys texture descriptor only capturing global image features and overlooking subtle details (i.e., wood knots). To address this issue, the image feature extraction network could be enhanced to focus more on modeling textures than generic images. Alternatively, our method could be extended to capture material appearances from multiple angles and distances, providing stronger supervision of surface properties.

Like previous work in differentiable procedural material capture, our method does not infer the procedural material graph structure from the input image. Optimization will fail if the prespecified graph lacks crucial elements of the target appearance. Guerrero et al. [90] tried to overcome this

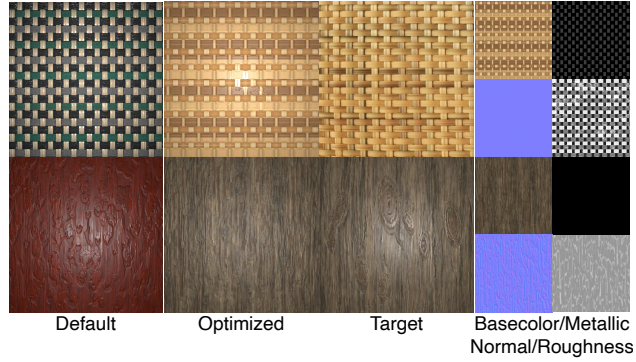


Figure 4.10: Demonstration of failure cases when optimizing procedural materials against real-world (top) and synthetic (bottom) images.

limitation by synthesizing novel graph structures using a deep generative model. However, it remains a challenge to accurately reconstruct production-grade procedural material graphs from target photos while ensuring that the output graph is easily comprehensible to artists.

4.3 Conclusion

This chapter presents an end-to-end pipeline for inverse procedure material capture based on Diffmat v2, a significantly enhanced computational infrastructure derived from the original Diffmat. Diffmat v2 fully translates and optimizes a vast array of artist-created, production-grade material graphs, which was previously infeasible due to the lack of differentiable noise and pattern generators. Our approach alternates between gradient-based and gradient-free optimization techniques to capture continuous and discrete parameters, resulting in higher-fidelity appearance reproductions than prior work. In addition, we introduce a novel method for example-based exposed parameter transfer, which simplifies the post-editing of materials without high-level meta-parameters. This allows for easy reuse and retargeting of materials. Although this work does not directly address graph structure synthesis, we provide advanced tools to alleviate the burden on artists by automating low-level parameter tuning and high-level parameter translation.

The applicability of Diffmat v2 can be further expanded with some minor improvements. Firstly, it is currently unable to parse procedural graphs with hierarchical dependencies, i.e., custom macro-nodes. Our parser requires users to manually unpack them in the main graph for end-to-end translation. Secondly, Diffmat v2 does not support non-square resolutions. Artists frequently create non-square textures to accommodate specific object shapes or reduce computational expenses for intermediate maps (e.g., when producing 1D gradient maps). Although Diffmat v2 automatically extends the canvas to a square in these scenarios, supporting non-square textures will improve optimization efficiency, especially at very high resolutions. Thirdly, Diffmat v2 does not support SVG or Text nodes, as they are less commonly used in natural materials. Moving forward, we plan to provide regular updates and maintain active technical support for Diffmat v2 to refine its features and encourage its use in various downstream tasks.

Chapter 5

Node Parameter Prediction with RL

The problem of procedural material generation is a special case of program synthesis [92] as procedural materials are effectively functional programs. Given the extreme complexity of such programs, different approaches were proposed in recent years to either regress or optimize for the parameters of a given graph [5, 83, 84] or generate the graph alongside the parameters [90, 92]. Nonetheless, regardless of how the graph structure is obtained, the estimated parameters play a key role in controlling the procedural graph’s output texture, such as structure, color, etc. Existing parameter estimation methods, however, either rely on CNNs that need to be re-trained for each graph, or on a lengthy optimization requiring a few minutes for parameter fitting. On the other hand, graph generation approaches leverage transformer models [91] and a linearized representation of material graphs to generate sets of nodes, edges, and parameters, both in a conditional and unconditional setting [90, 92]. However, to reach a given target appearance, these methods still rely on extensive random sampling and a costly post-optimization step as their parameter estimations are not accurate or robust enough. This amounts to several minutes of inference time, precluding their practical usage in responsive generative AI applications. We find the limitations of parameter generation in these methods partly due to the architecture used for conditioning, but mostly due to their supervised training approach. Supervised training is limited by the scarce availability of procedural material data and biases presented in the synthetic data. Only small sets of procedural materials are available for training, and there is a noticeable domain gap between these procedural material datasets and real materials. Additionally, supervised methods are trained with a parameter-space loss that does not necessarily correspond well to visual differences in material appearance.

To address these problems, we propose to leverage RL, presenting the first RL-based method to generate parameters for procedural materials that match the appearance given in a target image. Our use of RL is key to enabling two desirable properties: we can (1) directly maximize a reward function representing the visual similarity of procedural materials rather than their parameters, and (2) train on real material images, avoiding the scarcity and biases of procedural material datasets. Specifically, we follow a strategy inspired by RLHF: first pre-train a parameter generator using supervised training, and then fine-tune it with RL. During RL training, we use the appearance similarity between the target image and the procedural material as the reward, allowing us to train on images of real materials directly.

Compared to the supervised-learning method [92] which relies on picking the best of 150 samples and a time-consuming post-optimization [5] to achieve a reasonable appearance match, we show that our RL-based training framework significantly improves the prediction accuracy of the parameter

generator. Our method reaches a close appearance to the target image with much fewer samples and can even avoid the expensive post-optimization. We conduct several comparative evaluations to motivate the design choices in our two-stage (supervised and RL-fine-tuning) training pipeline. In summary, we enable faster, more accurate procedural material graph parameter generation through the following contributions:

- We propose the first RL training approach for procedural generative models, bridging the gap between generated tokens and image-space errors.
- Our setup is the first to introduce real material photographs as training data for procedural material generation.
- We show significantly better matching quality with substantially reduced inference time, unlocking the practical use-case of generative AI in procedural material modeling.

To facilitate reproducibility, we release the training and inference code on [GitHub](#) together with a parameter generator model pre-trained and fine-tuned on publicly available procedural materials.

5.1 Learning-Based Material Graph Generation

To generate complex procedural material graphs in AS3D, prior work [90, 92] leverages Transformers [91] to generate a linearized version of the graph. Specifically, the graph is produced in three sequences S_n , S_e , and S_p representing nodes, edges, and parameters, respectively. These sequences are generated using three separate Transformers, p_θ , p_ϕ , and p_ψ , either unconditionally from a distribution $p(g)$ [90] or conditioned on a text or image y , $p(g|y)$ [92]. For conditional generation, each transformer is conditioned as follows:

$$p(g|y) := p_\theta(S_n|y) p_\phi(S_e|S_n, y) p_\psi(S_p|S_e, S_n, y), \quad (5.1)$$

where θ , ϕ , and ψ are the trainable parameters of each transformer.

In this work, we focus on the parameter generator p_ψ . As a Transformer network, p_ψ models the joint probability of the parameter sequence S_p as product of conditional probability for each token s_i of the sequence, conditioned on all previously generated tokens $s_{<i}$:

$$p_\psi(S_p|S_e, S_n, y) := \prod_i p_\psi(s_i|s_{<i}, S_e, S_n, y). \quad (5.2)$$

In each step, the parameter estimator p_ψ outputs the probability distribution for the token s_i given the previous tokens, the graph structure, and the prompt. This probability distribution can be sampled to obtain the next token s_i .

To train the parameter generator, previous work relies on supervised training, where the ground truth token of each parameter is used to compute a parameter error. This approach has two main shortcomings: high-quality ground-truth data is scarce and expensive to create, and the parameter error does not always correlate well with the perceptual error of the resulting material. This frequently results in color or structural mismatch when matching a given target appearance. To mitigate the impact of this limitation, [92] rely on an additional post-processing optimization step [5], which further optimizes the continuous function parameters of the material for a better

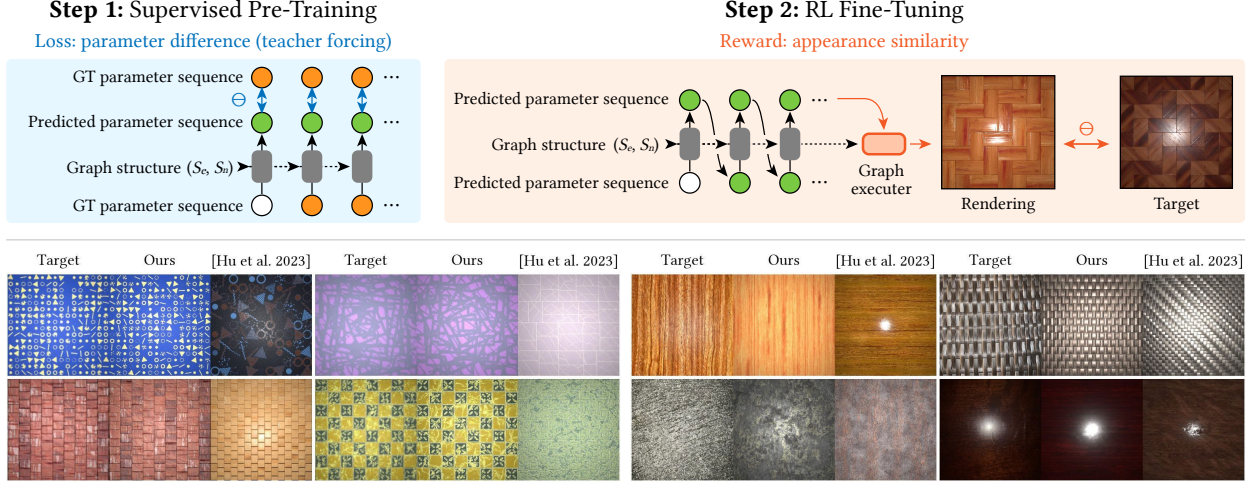


Figure 5.1: Top: We present a reinforcement learning (RL) framework to improve the prediction accuracy of a parameter generator for procedural material graphs. We pre-train the parameter generator with synthetic data and fine-tune it on both synthetic and real images using a reward function based on appearance similarity. Bottom: We show rendered material appearances using parameters predicted by our RL-fine-tuned model and the supervised-learning-only model [92] without post-optimization (left: synthetic data; right: real data). Our model yields significantly better matches with the target images.

perceptual match. The optimization step is applied to the top 5 samples among 150 random graph samples to select the best result. However, each optimization process typically takes up to several minutes per graph sample, making it impractically expensive compared with the cost of transformer predictions. In our work, we aim to lift this fundamental efficiency bottleneck by improving the quality of the generated material graphs. Specifically, we focus on improving parameter generation $p_\psi(S_p|S_e, S_n, y)$ assuming node sequences S_n and edge sequences S_e are known.

5.2 Method

Our goal is to train a parameter generator p_ψ to predict a parameter sequence S_p that better matches a target appearance, given as a target image I , than previous approaches. As we identify the parameter-space loss as a main bottleneck in current methods, we use an appearance-space loss (reward) instead to measure the perceptual match of the resulting image to a target. Producing tokens representing a program using a transformer-based model, then executing the program and producing an image, is a fundamentally non-differentiable sequence of operations. Thus, our key motivation for using Reinforcement Learning (RL) is that it can handle any non-differentiable reward function.

Our training approach therefore breaks into two steps as shown in Figure 5.1: (1) We first pre-train the parameter generator with supervised learning (Section 5.2.2) using augmented synthetic data (Section 5.3.1) for which ground-truth is available. (2) We then leverage RL to further fine-tune the parameter generator using appearance similarity as the reward function (Section 5.2.3). The implementation details are provided in Section 5.2.4.

5.2.1 Parameter Generator

For the parameter generator p_ψ , we use the same transformer architecture as in previous work [92], but change the conditioning mechanism and reduce the network capacity. More specifically, p_ψ uses the CLIP embedding y of the target image I for conditioning. In previous work, the embedding y is added to the latent state vector. Instead, we leverage cross-attention from the latent state vector to the CLIP embedding as the conditioning mechanism. This allows the parameter generator to obtain more detailed information about the target image. Additionally, we reduce the number of parameters by setting the hidden state dimension, the number of layers, and the number of attention heads to 48/4/4 for the node encoder and 96/4/4 for the parameter decoder as we found that it helps prevent overfitting.

5.2.2 Supervised Pre-Training

While RL does not necessarily require supervised pre-training, our parameter prediction setting is complex with potentially over 1,000 steps and a large pool of possible parameter values to choose from in each step. Exploring this space from scratch with only the appearance as guidance would be challenging. Thus, we found it beneficial to pre-train our parameter generator using augmented synthetic material graph data (Section 5.3.1). We pre-train the parameter generator through teacher forcing using an Adam optimizer [157] and a $1e-4$ learning rate. We apply gradient norm clipping (1.0) and a cosine learning rate schedule with linear warm-up and train on 8 NVIDIA A10g GPUs for 10 epochs. Combining the architecture and conditioning modifications, we observe a small improvement in the validation loss for supervised training compared to previous work [92], where the average cross-entropy loss per token declines from 0.803 to 0.770. For a fair comparison, we evaluate Hu et al. [92] using the updated architecture and training method.

5.2.3 Reinforcement Learning by Program Execution

After pre-training, we fine-tune the parameter estimator p_ψ using RL based on an appearance-space reward. Inspired by the recent success of LLM fine-tuning from human feedback [111], we use Proximal Policy Optimization (PPO) [158], where a *policy network* is trained to perform *actions* that maximize an accumulated per-action *reward*. In our setting the policy network is the parameter generator $p_\psi(s_i | s_{<i}, S_e, S_n, y)$ and an action is an evaluation of the generator, which predicts the probability distribution for the next token in the parameter sequence given all previous tokens. To simplify notations, we use “.” in short for $(s_{<i}, S_e, S_n, y)$ in subsequent equations.

The goal of PPO is to find the network parameters ψ that maximize the *expected accumulated reward* $J(p_\psi)$:

$$J(p_\psi) := \mathbb{E}_{S_p \sim p_\psi} \left[\sum_{i=1}^N \gamma^i R(s_i, \cdot) \right], \quad (5.3)$$

where $S_p = (s_1, \dots, s_N)$ is the parameter sequence and $\gamma \in (0, 1)$ is a *discount factor* that emphasizes earlier parameter tokens. The step-wise reward $R(s_i, \cdot)$ is computed for each token prediction s_i and is based on the appearance similarity between the target image I and the material resulting from the predicted parameters. We define the reward in detail further below. It can be shown [158] that the

gradient of $J(p_\psi)$ w.r.t the generator parameters ψ (also known as the *policy gradient*) is:

$$\nabla_\psi J(p_\psi) = \mathbb{E}_{S_p \sim p_\psi} \left[\sum_{i=1}^N \gamma^i R(s_i, \cdot) \nabla_\psi \log p_\psi(s_i | \cdot) \right]. \quad (5.4)$$

The expected value is approximated by averaging over parameter sequences randomly sampled from our generator and the resulting gradient is used to update our parameter generator.

Reward function. Our reward encourages similar appearances between the target image I and the material resulting from a parameter sequence S_p :

$$R_{\text{App}}(S_p, S_e, S_n, I) := 1 - d(h(S_p, S_e, S_n), I), \quad (5.5)$$

where h is a non-differentiable executor of the graph defined by $h(S_p, S_e, S_n)$, and I is the target image. The function d defines an appearance difference between two images:

$$\begin{aligned} d(I_x, I_y) := & \lambda_0 \|GM(I_x) - GM(I_y)\|_1 + \lambda_1 \|I_x^L - I_y^L\|_1 \\ & + \lambda_2 \|I_x^{\text{ab}} - I_y^{\text{ab}}\|_1, \end{aligned} \quad (5.6)$$

where I_x and I_y are images and GM is an operator that computes the Gram matrices of extracted VGG features [154]. I^L and I^{ab} are the L* channel and a*b* channels of the image in the L*a*b* color space. λ_i ($i = 0, 1, 2$) are adjustable weights to balance the contribution from color and texture statistics.

The appearance reward R_{App} is sparse in the sense that it is only available at the last step once the entire parameter sequence is generated. For all the steps, we follow Wu et al. [159] and impose a negative regularization reward $R_{\text{KL}}(s_i, \cdot)$ so that our generator p_ψ doesn't deviate excessively from the pre-trained generator p_ψ^* :

$$R_{\text{KL}}(s_i, \cdot) := -\beta \text{KL} \left(p_\psi(s_i | \cdot) \parallel p_\psi^*(s_i | \cdot) \right), \quad (5.7)$$

where KL denotes the KL-divergence between parameter token probability distributions. The final step-wise reward is given by:

$$R(s_i, \cdot) = \begin{cases} R_{\text{App}}(s_i, \cdot) + R_{\text{KL}}(s_i, \cdot), & i = N; \\ R_{\text{KL}}(s_i, \cdot), & i < N. \end{cases} \quad (5.8)$$

Note that this reward only uses the appearance reward for the last token. Next, we describe how we propagate the reward of the last token to previous tokens to update p_ψ at earlier steps.

Advantage and value network. Previous work has shown that using the *advantage* of each step, i.e. how much better a given token is compared to alternatives that could have been taken in the same step, is preferable to the reward [160]. In our case, the advantage also effectively propagates the appearance-based reward of the last token to all other tokens in the sequence. The advantage is derived as

$$A_i^{p_\psi}(s_i, \cdot) = R(s_i, \cdot) + \gamma V_{i+1}^{p_\psi} - V_i^{p_\psi}, \quad (5.9)$$

where $V_i^{p_\psi}$ is the *value* of the partial parameter sequence $s_{<i}$, defined as the expected accumulated reward (or the expected *return*) when completing the partial sequence $s_{<i}$ with our generator:

$$V_i^{p_\psi} := \mathbb{E}_{s_{\geq i} \sim p_\psi} \left[\sum_{t=i}^N \gamma^{(t-i)} R(s_t, \cdot) \right], \quad (5.10)$$

where $s_{\geq i}$ denotes the sequence $(s_i, \dots, s_N)$. The advantage $A_i^{p_\psi}(s_i, \cdot)$ from Equation (5.9) then replaces the reward $R(s_i, \cdot)$ in policy gradient calculation (Equation (5.4)). Note how the advantage at a token s_i is based on estimated rewards for all future tokens, including the last token s_N which has our appearance-based reward.

Approximating the expectation in Equation (5.10) with multiple samples is expensive since it is used in the inner loop of Equation (5.4). Instead, the value is typically approximated with a *value network* v_ρ trained simultaneously with p_ψ . We follow standard practice and employ generalized advantage estimation (GAE) [160] to approximate advantage scores, which provides a more effective trade-off between bias and variance than directly using Equation (5.9).

PPO objective. The PPO algorithm maximizes an objective function to jointly train the policy network p_ψ and the value network v_ρ . The core intuition is to perform iterative policy updates in smaller steps so that the training process more likely converges to an optimal solution. We follow the canonical PPO objective function in Schulman et al. [158]:

$$L(\psi, \rho) = \hat{\mathbb{E}}_i \left[L_i^{\text{CLIP}}(\psi) - c_1 L_i^{\text{VF}}(\rho) + c_2 S_i(\psi) \right]. \quad (5.11)$$

The first component is the clipped surrogate objective function,

$$L_i^{\text{CLIP}}(\psi) = \min \left(r_i(\psi) \hat{A}_i, \text{clip}(r_i(\psi), 1 - \epsilon, 1 + \epsilon) \hat{A}_i \right), \quad (5.12)$$

$$r_i(\psi) = \frac{p_\psi(s_i | \cdot)}{p_{\psi_{\text{old}}}(s_i | \cdot)}, \quad (5.13)$$

which constrains policy changes by clipping the probability ratio $r_i(\psi)$ between the current policy p_ψ and the previous policy $p_{\psi_{\text{old}}}$. $\hat{A}_i$ is short for the empirical estimate of $A_i^{p_{\psi_{\text{old}}}}(s_i, \cdot)$ from GAE. Then, the second loss component supervises value predictions using expected returns.

$$L_i^{\text{VF}}(\rho) = \left(v_\rho(\cdot) - V_i^{\text{target}} \right)^2. \quad (5.14)$$

With GAE, the target value combines advantage and value estimates from the previous policy and value networks.

$$V_i^{\text{target}} = \hat{A}_i + v_{\rho_{\text{old}}}(\cdot). \quad (5.15)$$

Finally, the third loss component $S_i(\psi)$ is the entropy of the token prediction at s_i , which serves as a bonus to encourage sufficient exploration.

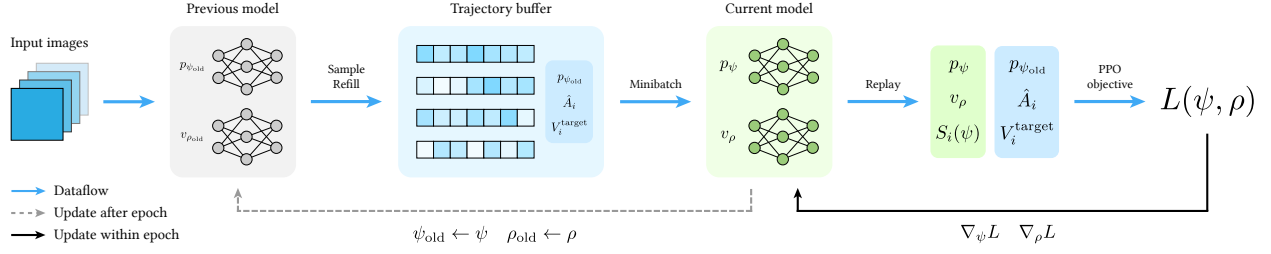


Figure 5.2: Workflow of our RL implementation. Each epoch begins with the previous model randomly sampling parameter token sequences (trajectories) and refilling a trajectory buffer. We then extract mini-batches of trajectories from the buffer and replay them to obtain the latest next-token probability and value predictions. The predictions are used to compute the PPO objective and update the current model through gradient backpropagation. The current model replaces the previous model at the end of each epoch.

5.2.4 RL Implementation

We base our RL framework on the Reinforcement Learning for Language Models (RL4LMs) library [130] and adjust it to our application. We inherit the value network architecture from the parameter generator p_{ψ} and initialize it with the pre-trained parameters p_{ψ}^* except for the classification head, which is replaced by a randomly initialized linear regression layer. Each training epoch comprises the following steps:

- Sample batches of entire parameter token sequences, i.e., *trajectories*, using the current policy p_{ψ} and fill a replay buffer collecting all state-action transitions in the trajectories.
- Train the policy network p_{ψ} and the value network v_{ρ} by sampling mini-batches of trajectories from the replay buffer, computing PPO’s objective function, and updating network weights via gradient backpropagation.
- Clear the replay buffer and continue.

This workflow is illustrated in Figure 5.2.

We fine-tune the parameter generator with RL using 8 GPUs for policy sampling and 96 CPU cores to evaluate the reward model. On an NVIDIA A100 40GB node, RL fine-tuning lasts 800 epochs and finishes in one day. Our RL fine-tuning process employs a replay buffer of 96 trajectories, resulting in an effective replay buffer of 768 trajectories in total. We train the networks for 10 epochs after refilling the replay buffer (sub-epochs relative to an RL training epoch) using a mini-batch size of 48 trajectories, an Adam optimizer, and a $1e-5$ learning rate.

In addition, RL4LMs [130] introduces a learnable masking network that filters the probability distribution of the next token and drops tokens below a certain cumulative probability threshold. We implement the same mechanism in our framework and directly use the policy network p_{ψ} to generate the action mask for each token prediction. The action mask leaves out all tokens in the last 10% of the probability distribution. As p_{ψ} evolves throughout the training process, it automatically learns to mask out tokens that may lead to degenerate material appearances. This helps stabilize RL fine-tuning by reducing the variance of the policy gradient.

Table 5.1 lists the hyperparameter values in our RL implementation such as reward weights, discount factors, and loss coefficients.

Table 5.1: Hyperparamter values for RL fine-tuning.

Description	Symbol	Value
VGG loss weight	λ_0	2.0
L channel loss weight	λ_1	0.01
a*b* channel loss weight	λ_2	0.05
KL divergence loss weight	β	0.001
Reward discount factor	γ	0.99
GAE discount factor	λ	0.98
Value loss coefficient	c_1	0.1
Entropy bonus coefficient	c_2	0.05

5.3 Dataset

We train our model with two datasets, a synthetic dataset (Section 5.3.1) used entirely for pre-training and partially for the RL fine-tuning step, and a real photograph dataset (Section 5.3.2) used only for the RL fine-tuning step.

5.3.1 Synthetic Dataset

We generate our synthetic dataset using the same 4,667 ground-truth procedural material graphs as previous work [90, 92], but augment it to guarantee that our initial parameter generator has completely converged and cannot benefit from additional synthetic data in a supervised setting. In particular, we extensively sample parameter combinations for each graph, encouraging graphs with rare nodes to be sampled more to balance the dataset. With exhaustive sampling, we create 10 million image-to-graph pairs, maximizing the usage of the original procedural materials. We use 90% of this dataset for training and a 5%/5% split for validation and testing. The split is done at the graph structure level and ensures graphs seen during training cannot arise in the validation/test set. This dataset is principally used in the supervised pre-training of our policy network. We combine this synthetic dataset with a dataset of real photographs described below for RL fine-tuning. We found training with this mixed dataset performs well on both real and synthetic test images.

5.3.2 Real Photograph Dataset

As our RL approach uses an image-space reward function, which is estimated from the sampled trajectories yielded by the parameter generator, it does not require paired image-to-graph ground-truth data for training. We can therefore train our model using real photographs as image prompts. The real photograph dataset, PhotoMat, contains over 10,000 fronto-parallel flash-lit material surface images captured by smartphones [52]. We divide this dataset into 90%/5%/5% splits for training, evaluation, and testing. Since we assume known node and edge sequences in our experiments, given a photograph, we retrieve the K-nearest-neighbor graphs from the rendered images in the synthetic dataset [86, 92] using CLIP [161] embeddings. During training, we retrieve $K = 5$ graphs and randomly use one as S_n and S_e to sample the parameter sequence S_p . We create a combined dataset in which we draw samples with a chance of 50% from the synthetic dataset or the real photograph dataset.

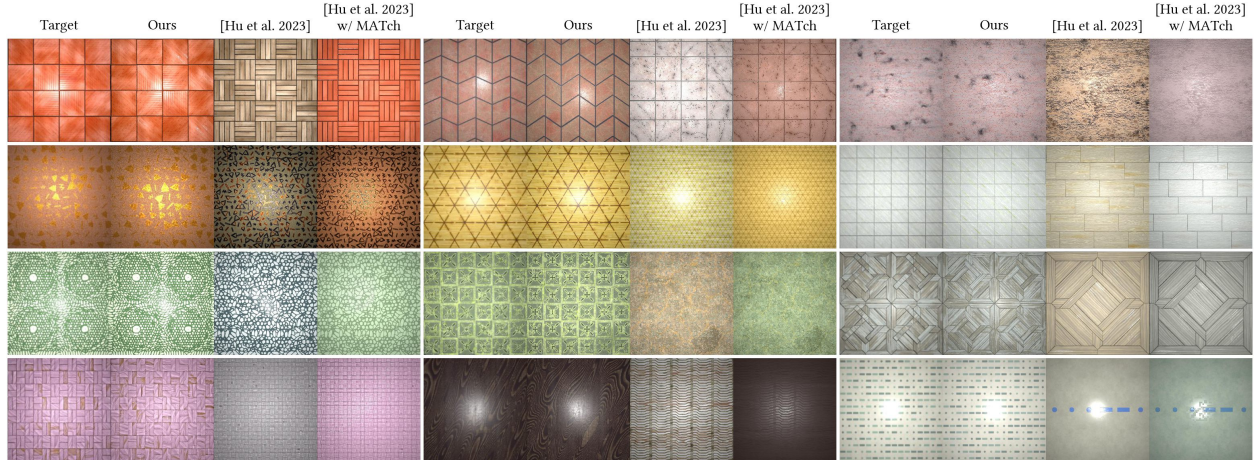


Figure 5.3: Qualitative evaluation on the synthetic dataset. We evaluate the appearance similarity between the target images and the generated graphs. Our RL-fine-tuned model predicts more accurate parameters, resulting in better structure and color/roughness matching compared to Hu et al. [92] Even with MATch post-optimization, the structure mismatching from Hu et al.’s predictions cannot be fixed due to the non-differentiability of relevant parameters.

5.4 Evaluation

We evaluate the RL-trained parameter transformer model p_ψ in terms of appearance similarity to a given target image after parameter prediction and program execution. We demonstrate performance improvement on both synthetic and real images (photographs) over the state-of-the-art method [92] on direct parameter prediction (Section 5.4.1) with or without MATch (Chapter 3 and 4). The graph structures (i.e., S_n and S_e) are assumed to be known for synthetic data and retrieved from our graph dataset for real data. To ensure a fair comparison, we retrain Hu et al.’s model using our augmented synthetic dataset, then evaluate both methods on the same ground-truth or retrieved graph structures. We further test RL fine-tuning using generated graph structures from Hu et al. [92] to evaluate the impact of graph structure retrieval (Section 5.4.3).

For the remaining subsections, we validate a few design choices of RL through ablation studies (Section 5.4.4), where we show evaluation results using only synthetic or real data for fine-tuning, without pre-training, or without the L^*a*b^* loss term. We also compare our method against alternative RL settings that use rewards instead of advantages or involve different reward combinations (Section 5.4.5). Finally, we analyze the trade-off between matching quality and time consumption for our method and discuss the implications for its deployment in real-time GenAI applications (Section 5.4.6).

5.4.1 Qualitative Results

We qualitatively compare our method with Hu et al. [92] on both synthetic and real images, demonstrating a significant improvement in appearance matching. Each result figure comprises input images alongside material renderings using the predicted parameters from different methods.

Figure 5.3 shows the performance of our model on the synthetic test set (Section 5.3.1). Given a

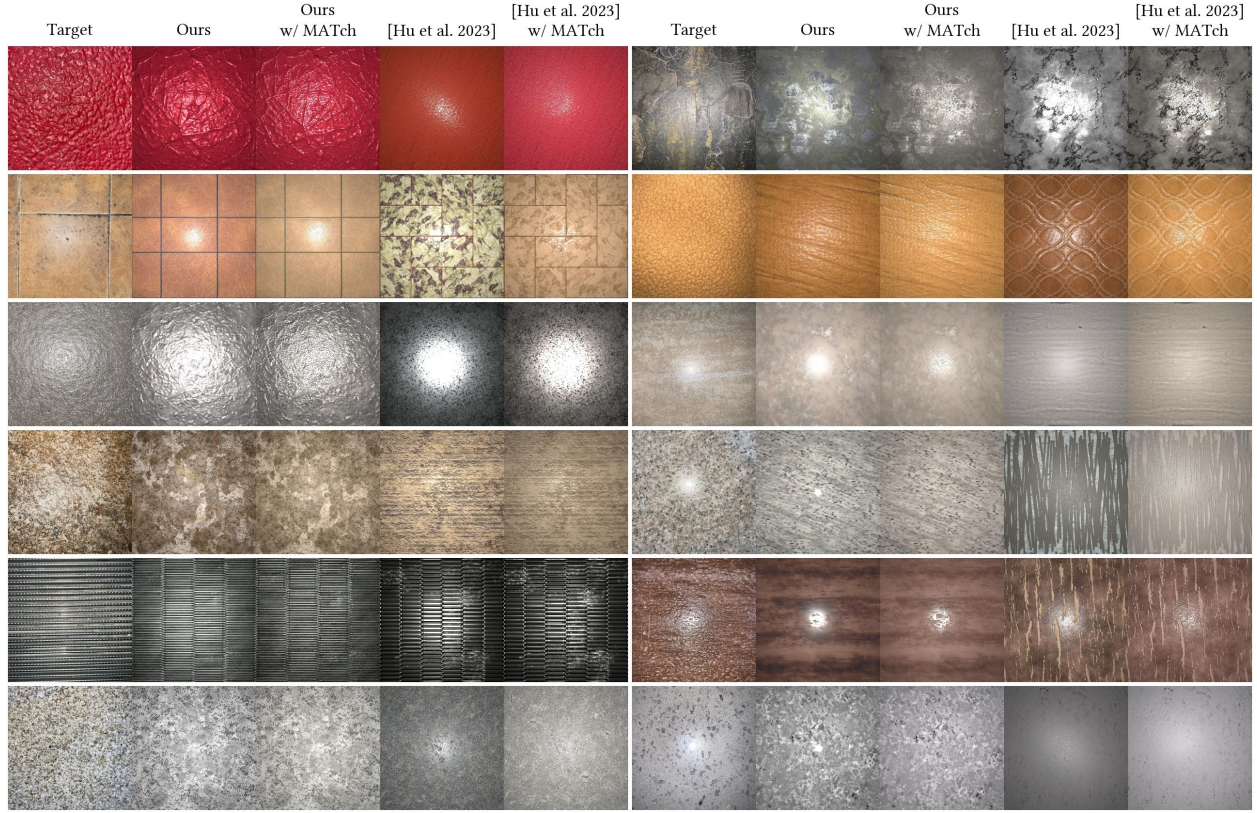


Figure 5.4: Qualitative evaluation on the real photograph dataset (top-1 results). RL fine-tuning with real photographs reduces the domain gap between synthetic and real images. Thus, our predictions are much closer to the targets than Hu et al. [92] which is only pre-trained on synthetic data. Our model achieves decent appearance similarity even without MATch post-optimization, while Hu et al. depends on MATch to improve matching quality.

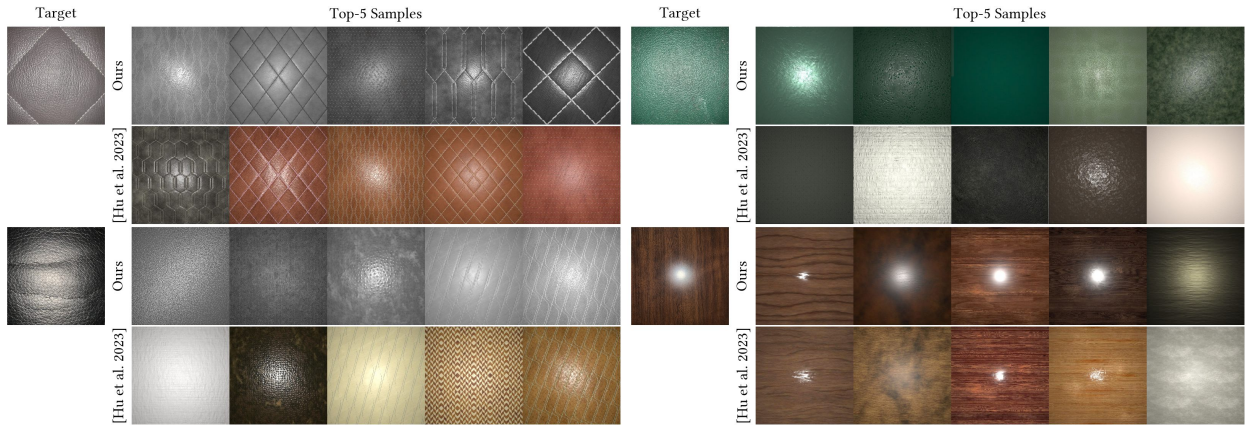


Figure 5.5: Qualitative evaluation on the real photograph dataset (top-5 results). We present the top results generated by our RL model. All the samples match reasonably with the target photographs and are noticeably better than Hu et al. [92] on average. For the supervised-only model, the match quality decreases dramatically for lower-ranking samples.

Table 5.2: Quantitative evaluation on the synthetic test set. For Reward, higher is better; for LPIPS, lower is better. Notice that our method, even without post-optimization, already outperforms previous work with MATch optimization, which verifies the high accuracy of our generated parameters.

Method	w/o MATch		w/ MATch	
	Reward $\uparrow$	LPIPS $\downarrow$	Reward $\uparrow$	LPIPS $\downarrow$
Hu et al. [92]	0.649	0.503	0.895	0.490
Ours	0.904	0.242	0.928	0.347

synthetic image and its ground-truth graph structure, we sample 5 parameter sequences and take the best result in terms of appearance similarity. The rendered procedural graphs with sampled parameters from our RL-fine-tuned model match the target image better than Hu et al. even without post-optimization or extensive sampling.

We further evaluate our approach to match real material photographs. This is particularly challenging for procedural material modeling as (1) the appearance distribution of real photos is more diverse than synthetic material renderings and (2) we do not have the ground-truth graph structure. Our RL training using real data partially resolves the first challenge and we find reasonably matching graph structures via retrieval from our material graph database. More specifically, we retrieve 5 graphs from the database based on the cosine similarity of CLIP image embeddings and sample 5 parameter sequences for each graph. We apply the same sampling strategy for Hu et al. [92] against which we compare. For qualitative comparison, we report the top-1 generated results in Figure 5.4 and provide the top-5 generated results in Figure 5.5. The results from our RL-trained model (both top-1 and top-5) show a substantial improvement in appearance matching quality thanks to the reduced domain gap between training and testing images. Without post-optimization, previous work performs poorly on a limited sample budget (5 samples only) and non-top-ranked graph structures.

Despite predicting parameters for the same set of graph structures, our RL model predicts more accurate parameters for the essential noise/pattern generator nodes in these graphs, resulting in better structure matching for both synthetic and real images. While post-optimization can improve color or roughness matching for Hu et al. [92], mismatches in the material structure are much harder to remediate as most noise/pattern generator nodes are non-differentiable.

More visual comparisons can be found in Figure A.1 and A.2.

5.4.2 Quantitative Evaluation

To showcase that the RL model constantly performs well on a wide range of image inputs, we report two metrics that measure the appearance similarity: 1) Reward as defined in Equation (5.5), representing both texture statistics and color match; (2) LPIPS [162], typically used as a perceptual distance metric for generic images.

Synthetic Data. We report the comparison on synthetic data in Table 5.2. The metrics are computed over 2,330 images from our synthetic test set, where we choose 10 random appearance variations for each graph structure. All results are generated from top-1 samples out of 5 sampled parameter sequences (same as Section 5.4.1). We note that the LPIPS score of our method becomes

Table 5.3: Quantitative evaluation on the test set of real photographs. Our results are constantly superior to previous work. For the LPIPS metric, our results without optimization yield better performance than previous work with optimization.

Method	w/o MATch		w/ MATch	
	Reward $\uparrow$	LPIPS $\downarrow$	Reward $\uparrow$	LPIPS $\downarrow$
Hu et al. [92]	0.827	0.620	0.901	0.574
Ours	0.847	0.571	0.912	0.555

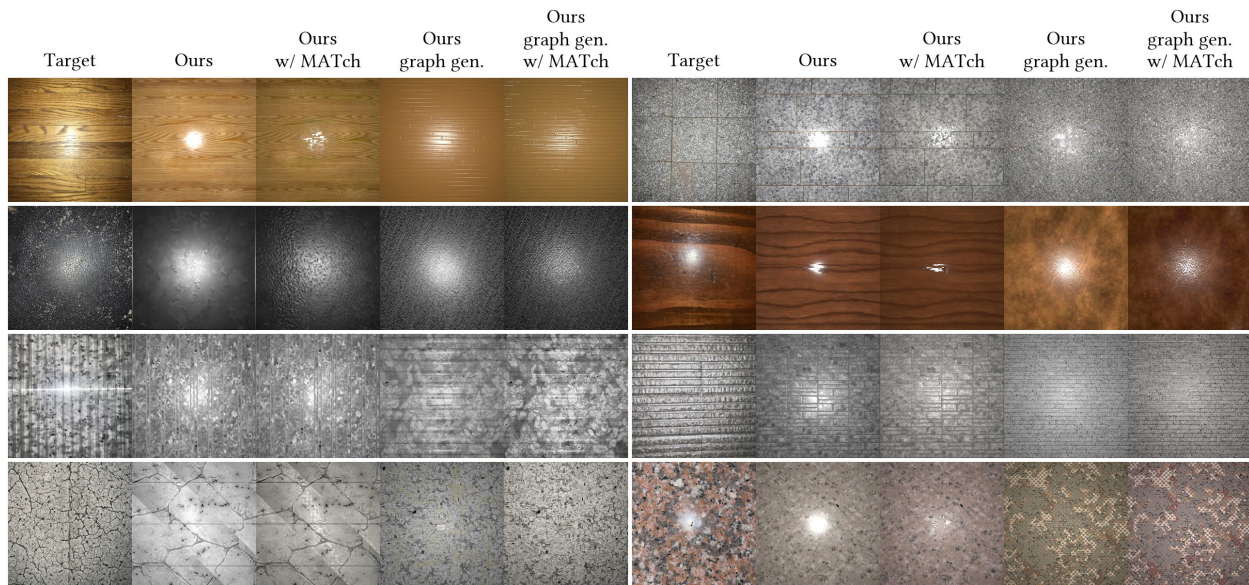


Figure 5.6: Parameter prediction for the real photograph test set after fine-tuning the parameter generator using retrieved (ours) or predicted graph structures (ours graph gen.), with and without MATch post-optimization. In many cases (Row 1, Row 2 Column 2, and Row 4), the results from generated graphs show worse matching in material structure, especially after MATch optimization.

worse after MATch optimization as the reward function has a different landscape from LPIPS despite both being derived from VGG features. Still, our scores are constantly better than Hu et al. [92]

Real Data. We report the comparison on real data in Table 5.3. In this case, we use the entire test set of PhotoMat including 582 real images. We apply the same sampling strategy described in Section 5.4.1, i.e., 5 random parameter samples per graph, to 5 nearest neighbor graphs. The quality is measured using top-1 samples as in Figure 5.4.

For both synthetic and real data, our RL-trained model outperforms Hu et al. [92] in both reward and LPIPS scores, although the advantage is more evident on synthetic data. Indeed, real material photos are significantly harder to match as they cover a wider appearance distribution than our synthetic material dataset. To a lesser extent, they exhibit rich variation in lighting and camera effects whereas our procedural material renderer adopts a fixed setup. Nonetheless, our results *without optimization* show similar or even better quality than Hu et al. *with optimization*. This allows our approach to rely less on time-consuming post-optimization as the parameter predictions are more accurate.

Table 5.4: Quantitative comparison between RL fine-tuning using graph structures retrieved from the synthetic dataset (ours) or predicted by the node and edge generators in Hu et al. [92] (ours w/ graph gen.). The metrics are computed for the real photograph test set.

Method	w/o MATch		w/ MATch	
	Reward $\uparrow$	LPIPS $\downarrow$	Reward $\uparrow$	LPIPS $\downarrow$
Ours w/ graph gen.	0.852	0.601	0.893	0.578
Ours	0.847	0.571	0.912	0.555

5.4.3 Graph Retrieval vs. Generation

We investigate whether incorporating more diverse graph structures during fine-tuning improves generalization to unseen real images. To test this idea, we pre-train the node and edge generators (p_θ, p_ϕ) in Hu et al. [92] with our augmented synthetic dataset and randomly sample S_n and S_e for real images during RL fine-tuning. p_θ and p_ϕ remain fixed in the process and only serve as sources of domain randomization for the parameter generator p_ψ . Thus, they do not contribute to the discounted reward in Equation (5.3).

We compare our model with its variant trained on generated graph structures qualitatively and quantitatively. For each real image, we apply our model using the sampling method described in Section 5.4.1 and perform MATch optimization on the highest-ranked generated result. In contrast, we randomly generate both graph structures and parameters for the variant model under an identical budget of 25 samples. We report reward and LPIPS scores over the real photograph test set with and without MATch optimization in Table 5.4. Visual comparison results are included in Figure 5.6.

We notice that domain randomization with pre-trained node and edge generators does not consistently improve the matching quality. The variant model shows worse metrics than our model except for a marginal lead in reward without MATch optimization. As also illustrated in Figure 5.6, the generated graph structures are occasionally preferable to retrieved ones (e.g., more accurate beam patterns in Row 3) but inferior in other cases (e.g., Row 1 and Row 4). The suboptimal graph structure quality becomes more prominent after MATch optimization as it is non-trivial to correct structural mismatches with differentiable parameter optimization (see the discussion in Section 5.4.1). This contributes to a lower reward score with MATch than our model. Therefore, empirically, our graph retrieval strategy demonstrates comparable expressiveness to the state-of-the-art graph generation model without imposing additional restrictions.

5.4.4 Ablation Study

We conduct ablation studies to verify our choice of hybrid dataset composition, two-step training strategy, and $L^*a^*b^*$ color regularization in the reward function.

Choice of dataset. To validate the choice of a 50-50 mixed training dataset, we show the prediction accuracy on synthetic and real photograph test sets using RL-fine-tuned models trained on different dataset compositions (synthetic data only, real data only, mixed dataset) in Table 5.5. A visual comparison is provided in Figure 5.7. We found that training with only synthetic data does not generalize well to real data (bottom table), highlighting the importance of introducing real data in the training through RL. Further, incorporating real data in training also helps improve parameter

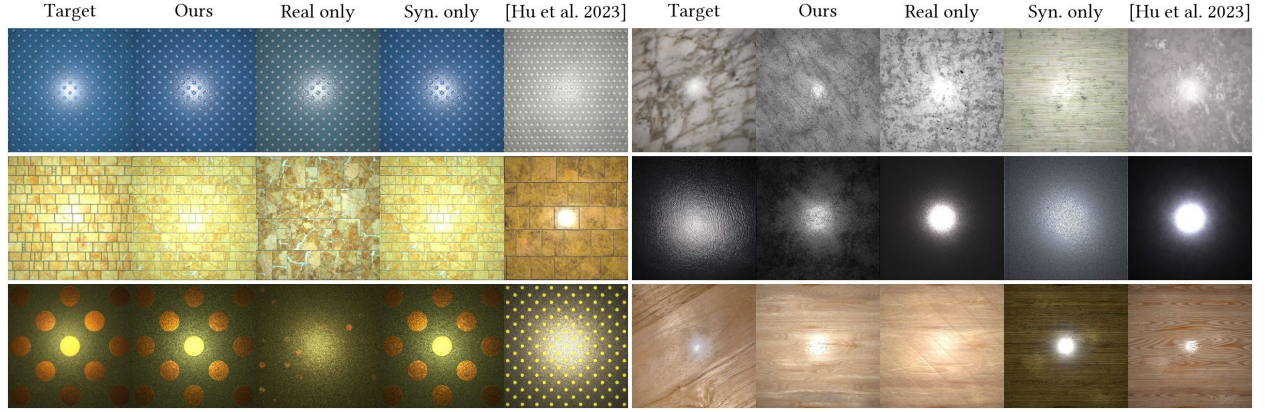


Figure 5.7: Ablation study: choice of dataset. We experimented with RL fine-tuning on different dataset compositions including a mixed dataset (Ours), real photographs only (Real only), and synthetic data only (Syn. only). The examples in the left column are from the synthetic test set, while the ones in the right column are from the real photograph test set. The results from Hu et al. [92] (without post-optimization) are provided for reference. We notice that training only with synthetic data fails to produce good matching for real photographs, while interestingly, the model trained only on real data shows reasonable generalization to synthetic data. Our choice of hybrid dataset composition performs best on both test sets.

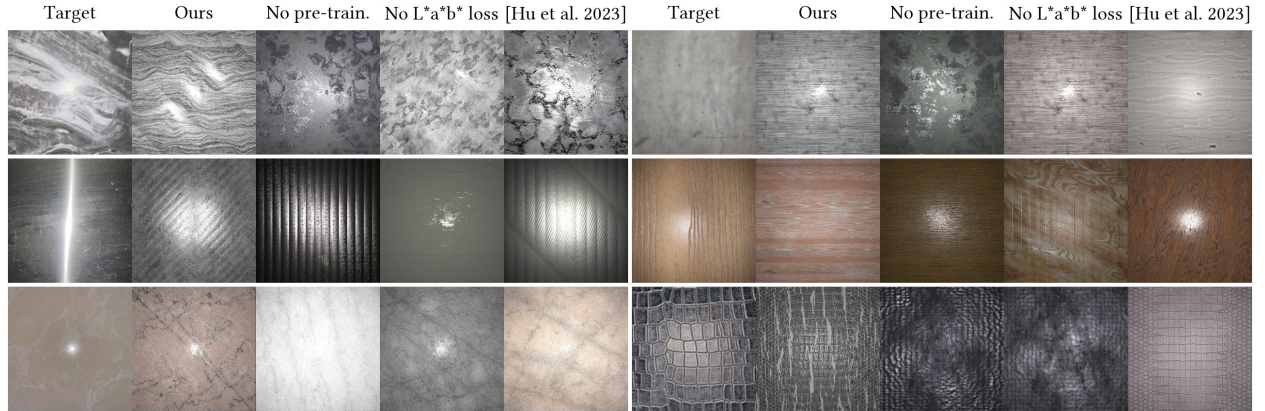


Figure 5.8: Ablation study: pre-training and loss function. The appearance matching quality degrades without pre-training or the L*a*b* loss term, causing color or pattern mismatching. All input images are real photographs. The results from Hu et al. [92] (without post-optimization) are provided as a reference.

estimation for synthetic data (top table). We select the mixed dataset as our default option as it demonstrates better generalization across both synthetic and real test datasets.

Pre-training and loss function. We evaluate how the pre-training step and the L*a*b* color term in the loss function affect RL performance. Table 5.6 and Figure 5.8 show that removing these two components results in worse appearance matches with input images.

Table 5.5: Ablation study: choice of dataset. We test the prediction quality on the synthetic test set (top) and the real photograph test set (bottom) using RL models trained on different dataset compositions (synthetic data only, real data only, and our mixed dataset).

Synthetic test set	Reward↑	LPIPS↓
Hu et al. [92]	0.649	0.503
Ours w/ synthetic data only	0.897	0.249
Ours w/ real data only	0.854	0.311
Ours	0.904	0.242
Real photograph test set	Reward↑	LPIPS↓
Hu et al. [92]	0.827	0.620
Ours w/ synthetic data only	0.819	0.581
Ours w/ real data only	0.850	0.574
Ours	0.847	0.571

Table 5.6: Ablation study: pre-training and loss function. The metrics are collected from the real photograph test dataset. The matching quality worsens without pre-training or the $L^*a^*b^*$ component in the reward function.

Method	Reward↑	LPIPS↓
Hu et al. [92]	0.827	0.620
Ours w/o pre-training	0.828	0.580
Ours w/o $L^*a^*b^*$ loss	0.837	0.581
Ours	0.847	0.571

5.4.5 Alternative RL Settings

We inspect several technical details of the RL algorithm by validating our model against the following alternative RL settings:

- *Replacing advantages with rewards.* Estimate the expected return for each prediction step only using discounted rewards from the sampled trajectory instead of the advantage score (Equation (5.9)). The observed rewards also provide direct supervision for the value network.
- *Replacing VGG loss.* Measure differences in texture statistics using sliced Wasserstein distance (SWD) [163] instead of the Gram matrices of VGG features.
- *Dominant VGG loss,* where we set $\lambda_0 = 20$, $\lambda_1 = 0.002$, and $\lambda_2 = 0.01$ in the reward function to enforce better matches of VGG features.
- *Dominant a^*b^* color loss,* which uses $\lambda_0 = 0.5$, $\lambda_1 = 0.006$, and $\lambda_2 = 0.06$ to strengthen color regularization.

Qualitative comparison results are shown in Figure 5.9. Our method demonstrates a more robust matching quality on unseen real images than the alternative RL settings. Specifically, we find that overemphasizing the VGG reward term may lead to severe color mismatches, while excessively weighting the a^*b^* reward term may compromise the generated material structure.

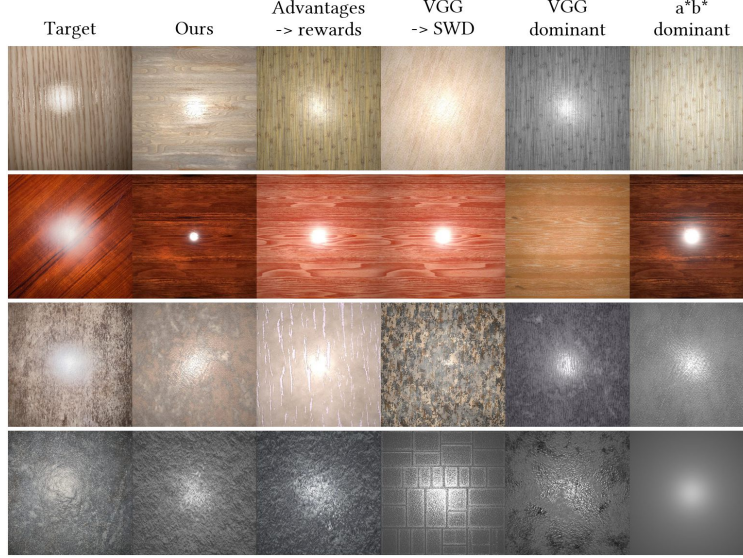


Figure 5.9: We compare our method with several alternative RL settings including estimating returns using rewards instead of advantages (advantages $\rightarrow$ rewards), replacing the VGG loss term with sliced Wasserstein distance (VGG $\rightarrow$ SWD), or making the VGG or a^*b^* color term dominant in the reward function. All evaluation uses real test images. The other approaches are more prone to color and structure mismatches.

5.4.6 Matching Quality and Time Cost

Our RL fine-tuning approach enables fast and accurate parameter prediction, promoting the practical use of generative AI for procedural material modeling. Hu et al. [92] achieves high-quality procedural material generation with 150 samples plus several minutes of post-optimization. This is typically too long for the deployment of state-of-the-art generative AI models [164, 165]. In contrast, we show in Figure 5.10 that our method can generate materials of similar or superior quality to the previous work using significantly fewer samples. We also demonstrate superior generation quality under identical sampling budgets thanks to the high accuracy and low variance of the RL fine-tuned model. The sampling budget refers to the number of parameter sequences we sample. We provide additional examples in Figure A.3 and A.4.

We analyze the trade-off between generation quality (reward) and time consumption over synthetic and real photograph datasets in Figure 5.11. In both test cases, our model achieves a considerably higher reward within 1 second. The user may then opt to run post-optimization to further improve the quality. On the other hand, the supervised method requires more than 20 $\times$ longer time or minute-long MATch optimization to reach the same reward level, making it less practical for interactive generative AI applications where faster response and higher throughput are generally preferred.

Key to our model’s faster inference speed is the omission of redundant or inconsequential parameters in the generated sequence. Our definition of S_p allows parameters to be skipped during generation and remain at their default values. Combined with the discounted reward in RL, this encourages the model to improve image similarity and predict shorter sequences whenever possible. As demonstrated in Figure 5.12, on synthetic data, our RL-fine-tuned model (Epoch 256) reduces

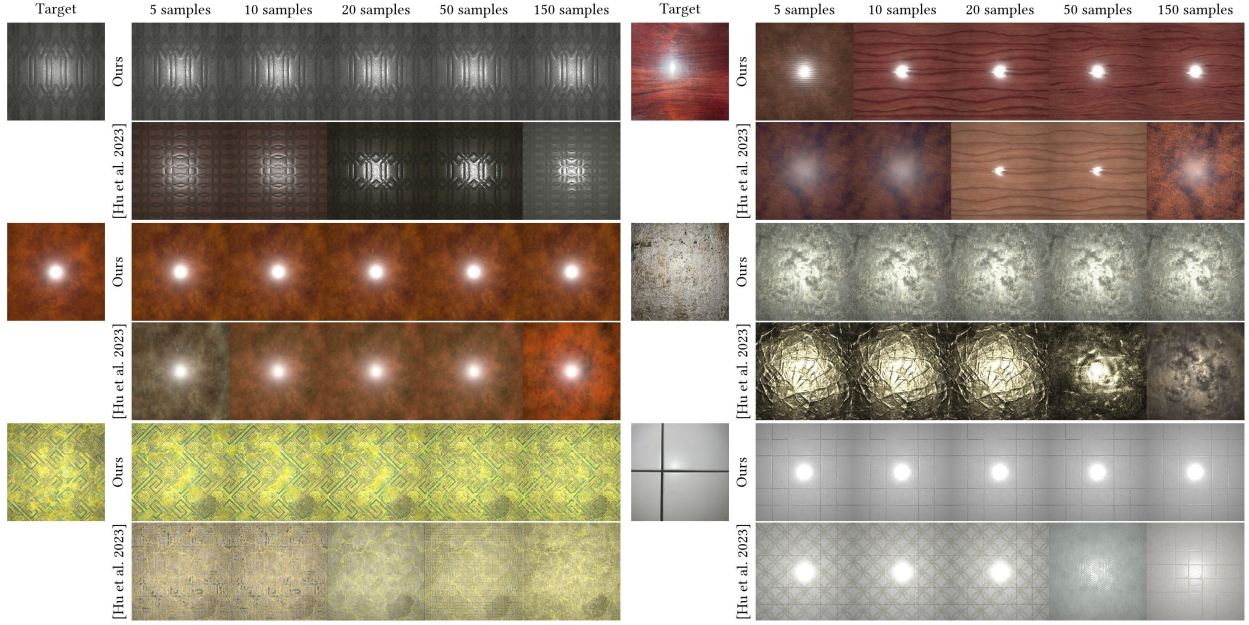


Figure 5.10: Qualitative evaluation on synthetic (left column) and real photograph datasets (right column) with increasing random sampling budgets. Since RL fine-tuning effectively reduces the variance during inference, our model does not require extensive sampling to produce high-quality materials. Previous work usually needs to sample more graphs (150 samples in Hu et al. [92]) to refine the quality.

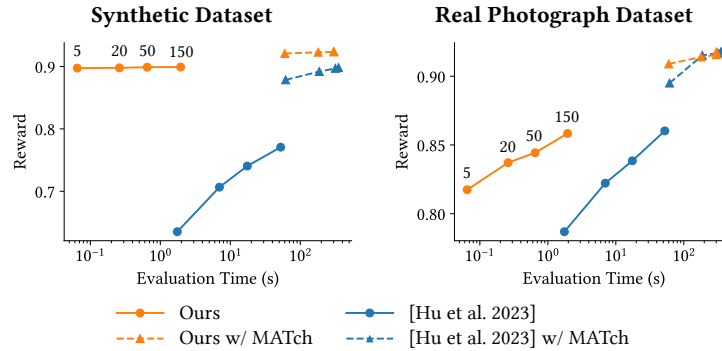


Figure 5.11: We analyze the trade-off between matching quality (reward) and time consumption for both RL-trained and supervised models. We collect the total evaluation time, including parameter generation and post-optimization, at four sampling budgets: 5, 20, 50, and 150 samples (black text above line markers). The top-ranked samples used for post-optimization at these budgets are 1, 3, 5, and 5, respectively. Time consumption is measured on an NVIDIA RTX 3090 GPU at peak throughput. A longer evaluation time typically leads to better matching quality. Our model (solid orange lines) achieves higher reward at a much lower time cost. Conversely, the supervised model (solid blue lines) has to draw more samples or rely on expensive post-optimization to reach comparable quality, which is detrimental to evaluation time. The results illustrate that our model is fast and accurate, unlocking opportunities for practical deployment in generative AI applications.

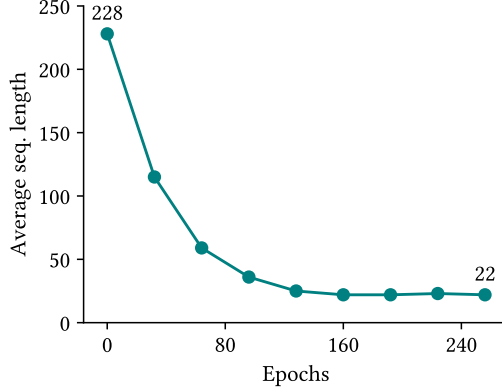


Figure 5.12: We visualize the average length of predicted parameter sequences w.r.t. training epochs. The policy model effectively learns to set parameters with minor contributions to material appearances at default values to shorten the sequence length. The average sequence lengths are computed among 200 randomly selected images in the training dataset.

the average length of predicted parameter sequences to less than 10% of the supervised model [92] (Epoch 0) while achieving superior matching quality. This shows that our model can distill the most important parameters from each graph structure and avoid others with little impact on the final appearance. In contrast, most materials in our 10-million synthetic dataset have long ground-truth parameter sequences due to random sampling. This prompts the supervised model to generate longer sequences than necessary, resulting in a higher variance in parameter prediction.

5.5 Discussion and Conclusion

In Section 5.4.3, we conclude that our assumption of known or retrieved graph structures yields comparable expressiveness to those predicted using the state-of-the-art generative model [92]. However, mismatches may still occur if a material graph can not sufficiently represent the input texture. Figure 5.13 shows instances where neither our approach nor Hu et al. [92] could match the target appearance with retrieved or generated graphs. Addressing these limitations requires improvements in material graph structure prediction techniques as the foundation for more effective parameter generator fine-tuning.

In summary, we have introduced RL to improve the generation of high-quality procedural material graphs conditioned on images. Our approach leverages an image-space reward function to bridge the domain gap between parameter tokens and appearance similarity. This allows us to integrate real photographs into fine-tuning, enhancing generalization to unseen real images. We demonstrate substantial improvements in image matching quality compared to conventional supervised training methods, without relying on expensive post-optimization. Our RL fine-tuning workflow shows promise for a wide range of generative models in procedural graphics, particularly in scenarios where (1) the parameter space has a weak correlation with the image space, or (2) the availability of ground-truth data is limited.

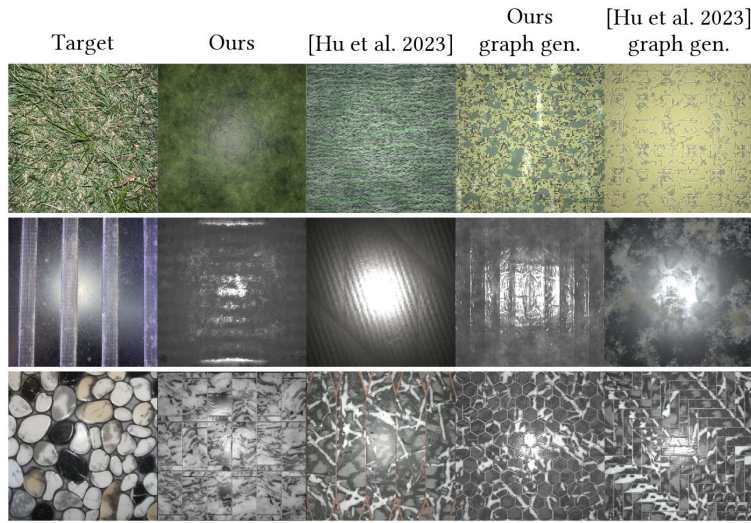


Figure 5.13: In a few cases, both our RL-fine-tuned model and previous work struggle to match the input target closely due to imperfect retrieved graph structures (2nd and 3rd column). As in Section 5.4.3, we also experimented with generated graph structures (graph gen.) using node/edge generators trained from supervised learning [92] (4th and 5th column). In some cases, they may produce slightly better results but still fail to model important structural features in the input appearance.

Chapter 6

Material Graph Generation using VLMs

In this chapter, we explore the fact that procedural material graphs in Blender [2] can be transpiled into standard Python programs and fine-tune a pre-trained large vision-language model (VLM) to generate such programs directly. A straightforward approach is to directly prompt pre-trained VLMs to generate procedural material programs from images. However, we find that existing commercial VLMs (e.g., GPT-4o-mini) struggle on this task (see Table 6.1) as procedural material programs are domain-specific and underrepresented in the training data of VLMs. Therefore, we fine-tune a pre-trained VLM on pairs of images and procedural material programs. The scarcity of publicly available procedural materials, however, poses a fundamental challenge. Inspired by FunSearch [133], we leverage pre-trained VLMs to perform program-level data augmentation followed by carefully designed node parameter augmentation. This data augmentation strategy extends our dataset from $\sim 1.6\text{K}$ examples to $\sim 550\text{K}$ examples, significantly improving the matching quality of predicted materials.

Through extensive evaluation, we show that our method outperforms previous single-image procedural material generation methods on both synthetic and real-world examples. As shown in Figure 6.1, our generated procedural materials can be directly applied to a 3D scene with full editability. Furthermore, previous works are developed for proprietary software, e.g., AS3D [1], thus precluded from releasing training data. We contribute an open-source procedural material dataset in Blender [2] to promote future research in this area.

6.1 Method

Figure 6.2 illustrates the workflow of our method. Given an input image capturing the material appearance on a flat surface, we use a fine-tuned large VLM to predict a Blender procedural material in Python code. Executing the code in Blender yields the shader node graph of the material, which is rendered into the output image under a point light source. The procedural materials predicted by our model preserve full editability albeit under the constraint of pre-defined node and parameter types from Blender and users. Following previous works [5, 92], we aim to achieve a close perceptual match with the input image instead of a pixel-perfect reproduction.

Fine-tuning a large VLM requires a comprehensive procedural material dataset with diverse material appearances and graph structures. Prior work [92] has developed such a dataset for AS3D materials but their dataset can not be released due to proprietary intellectual property. To address

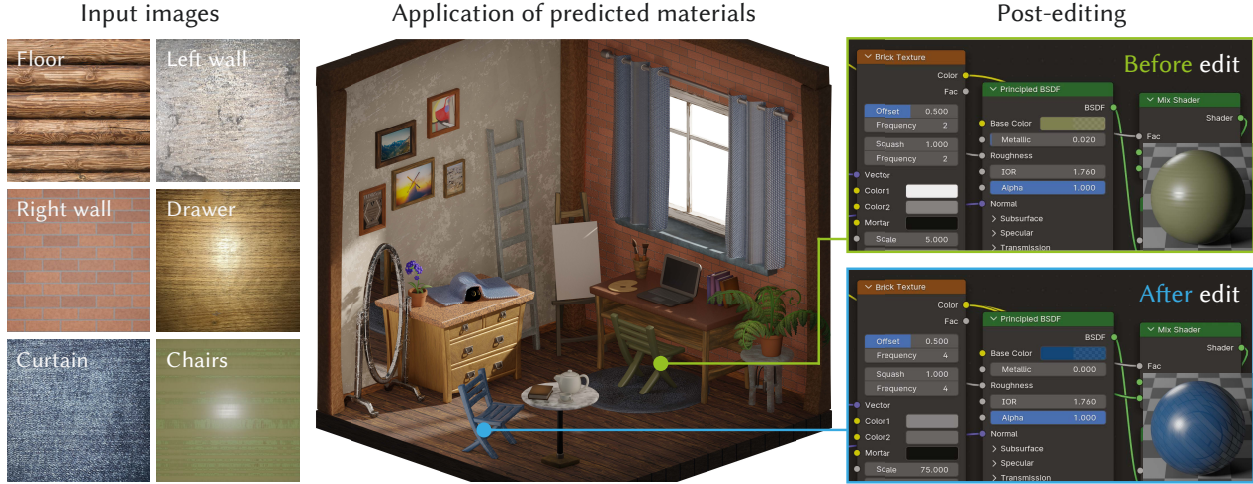


Figure 6.1: Given single input images (left), our model generates corresponding procedural materials (top right) that can be directly applied to a 3D scene (middle). The generated materials can be easily edited, e.g., by changing some node parameters, to achieve desired visual appearance (bottom right).

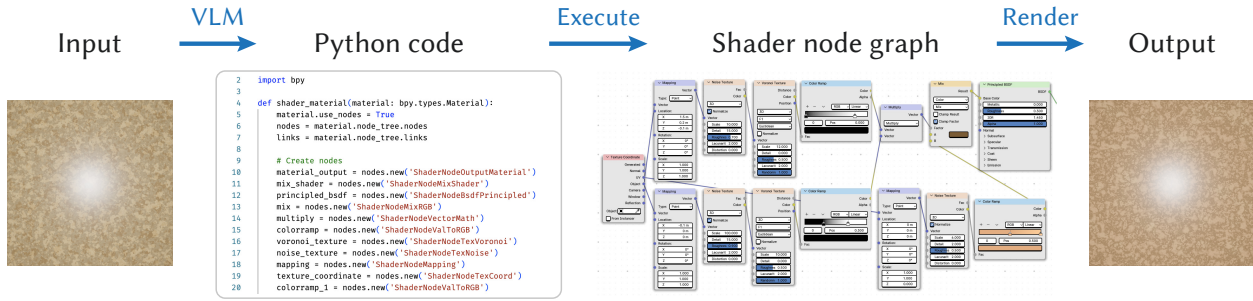


Figure 6.2: Workflow overview. Given an input image, we use our fine-tuned VLM to predict a procedural material in the form of Python code. Executing the code in Blender yields a user-editable shader node graph, whose rendered appearance closely matches the input image.

the lack of an open-source counterpart, we specially curate a dataset containing over 550K Blender procedural materials (Section 6.1.1 and 6.1.2). The remaining subsections describe our fine-tuning process (Section 6.1.3) and post-optimization algorithm (Section 6.1.4).

6.1.1 Data Collection

We first collected 3,663 free Blender materials from three online sources: 1) 2,411 materials in BlenderKit¹, an online repository of Blender 3D assets; 2) the 60 base materials of Infinigen [166], a procedural 3D scene generation framework; 3) 1,192 materials from individually published Blender procedural material packs². These materials are manually created by experienced technical artists and cover a wide variety of material categories. Then, we conduct the following curation process before adding each material to training data:

¹<https://www.blenderkit.com>

²<https://heypictures.gumroad.com/l/zwhai>; <https://maroc777772.gumroad.com/l/oaehcd>

Node graph clean-up. We run a depth-first search (DFS) from the material output node to identify nodes that participate in output appearance calculation, removing other redundant nodes and edges. The material is dropped if a valid output node does not exist or the node graph is too large (with more than 30 nodes). In addition, each node group in the material is treated as a custom node type distinguished by its functional signature. To prevent the VLM from struggling to memorize too many custom node types, we iteratively expand the smallest node groups in these materials into corresponding subgraphs until reaching the graph size limit.

Program validation. We subsequently transpile the remaining materials into Python code and render the resulting appearances. This step filters out materials with excessively long code sequences (more than 2,048 tokens after tokenization) or degenerate renderings, including empty images, uniform colors, or textures lacking meaningful spatial features. We heuristically measure the complexity of a material using the compressed JPEG file size of its rendered image. By sorting the materials with descending JPEG file sizes, we observe that most materials below 12KB exhibit undesirable texture quality. We thus apply this heuristic threshold to discard unwanted, meaningless materials from the dataset.

6.1.2 Data Augmentation

The scarcity of high-quality artist-created procedural materials has been a fundamental challenge for training generative models [92]. Blender materials are no exception—the curation process above leaves 1,640 materials in our dataset, which we found severely inadequate for VLM fine-tuning. Hu et al. [92] proposes to create appearance variations of existing materials by randomly perturbing node parameters. While this effectively introduces color and pattern changes, the sampled variants retain the same node graph structure, which hardly prevents the model from overfitting to a limited set of graph structures.

Our data augmentation strategy combines the augmentation of graph structures and parameters.

Graph structure augmentation. We augment the 1.6K artist-created materials with diverse program structures using an approach similar to the evolutionary workflow in FunSearch [133]. Specifically, we manually select a subset of 870 complex materials with rich texture detail from the artist-created materials into a sample pool. The evolution process involves selecting random program pairs from the sample pool and performing “genetic crossovers” using a commercial pre-trained LLM (GPT-4o-mini), which is prompted to generate novel programs using elements from the selected programs. The prompt template is as follows.

You are familiar with creating procedural materials using Blender's Python API. You will be given a Python code block delimited by triple backticks, which contain two functions that define the node trees of two procedural materials in Blender. The functions are named ``shader_material_1`` and ``shader_material_2``, respectively. Your task is to write a Python function ``shader_material`` that creates a new material based on the provided functions. Write your code following the guidelines below.

Code template:

```

```python
import bpy

def shader_material(material: bpy.types.Material):
 material.use_nodes = True
 nodes = material.node_tree.nodes
 links = material.node_tree.links

 # Create nodes
 # YOUR CODE HERE

 # Create links to connect nodes
 # YOUR CODE HERE

 # Set parameters for each node
 # YOUR CODE HERE
```

```

Rules:

1. Create no more than 30 nodes. Only use node types and parameter fields referenced in the provided functions. Change parameter values as needed.
2. Make sure your code can be correctly executed in Blender 3.3. Refer to the Blender Python API documentation for valid node types and parameters.
3. Try to generate materials with complex and semantically meaningful appearances. Avoid generating materials that lack structure or look too similar to the provided examples.
4. Follow the format and coding style of the example functions. Do not add new code blocks or comments.
5. Simply reply with code. Exclude any additional text or explanations.

The LLM can restructure the example programs and modify their node parameters as needed. We then validate the LLM-generated programs and add executable programs to the sample pool, where each program pair contributes up to four valid programs out of 20 trials. Our evolutionary workflow produces 50.4K programs with distinct structures, more than 30× the number of artist-created materials.

Parameter augmentation. After graph structure augmentation, we conduct parameter-space augmentation following [92]. Given a procedural material with a prescribed graph structure, we randomly sample its node parameters around their initial values to generate appearance variations. The collection of sampled parameters includes: 1) unoccupied input and output connectors of each node with well-defined default values, e.g., the color input to a principled BSDF shader node; 2) node-specific attributes, such as the operation type of a math node and the interpolation option of a color ramp node. Although our sampling strategy is similar to [92], we develop a native implementation for Blender’s procedural material system, which applies different sampling algorithms contingent on whether a node parameter is inherently continuous or discrete.

- For a continuous parameter, i.e., a float, vector, color, or color ramp, we refer to its definition in the associated node and its value statistics across the procedural material dataset to determine a feasible range. The parameter is then uniformly sampled within the intersection of the feasible range and a $\pm 25\%$ interval around the initial value. Vector or array elements are sampled independently from one another. Considering that the relative interval may be insufficient for small initial values, we enforce ± 0.05 as an absolute, minimal sampling interval. Specifically, color parameters are sampled in the hue-saturation-value (HSV) color space instead of the conventional RGB space. We uniformly sample hue channels from the entire $[0, 1]$ range and sample the other channels within a ± 0.5 interval. This encourages diverse color tones in sampled materials.
- We divide discrete parameters into two separate scenarios. An integer parameter is treated as continuous and rounded to the nearest integer after uniform sampling. Categorical and Boolean parameters, in contrast, do not have the notion of absolute values or intervals. We thus uniformly sample each parameter from its value range at a probability of 0.25.

An important caveat of sampling all node parameters at once is the additional program length. In Blender Python programs, explicit assignments are unnecessary for node parameters remaining at default values. Sampling all node parameters away from their default values will accumulate dozens of lines of code, creating excessively long code sequences. We prevent this issue by sampling node parameters holding default values with a probability of 0.2. If the resulting program exceeds the token length limit (i.e., 2,048 LLaMA 3 tokens), we randomly reset some parameters back to default values to reduce the code length. The resulting program must pass the validation (Section 6.1.1) before being incorporated into training data.

Parameter augmentation creates another $10\times$ more program variations from random parameter perturbations. The augmented dataset eventually comprises over 550K materials, more than $330\times$ the initial size.

6.1.3 VLM Fine-Tuning

Our augmented procedural material dataset enables the adaptation of pre-trained VLMs to image-conditioned procedural material generation tasks. We employ the latest version of LLaVA-NeXT [167] from the LLaVA model family [168–170], a series of open-source multi-modal LLMs trained on public-domain visual question answering (VQA) data. The model architecture consists of a CLIP ViT/L-14 vision encoder [161], a LLaMA 3 8B language decoder [171], and an MLP projector that bridges the embedding spaces of the encoder and decoder networks.

For supervised fine-tuning, we pair the input image with the following user prompt to the VLM and tokenize the ground-truth Python program using LLaMA 3’s tokenizer.

```
<image>
Write a Python function with Blender API to create a material node graph
for this image.
```

The trainable modules include the MLP projector and an array of LoRA adapters [172] applied to all attention-related linear layers in the LLaMA 3 model (with $r = 8$, $\alpha = 32$, and a 0.05 dropout probability), amounting to 40M trainable parameters in total.

Algorithm 1 MCMC-based local parameter search.

```
1:  $i \leftarrow 0$ ,  $M \leftarrow \tilde{M}$ ,  $l \leftarrow L(I, \tilde{I})$ 
2:  $M^* \leftarrow M$ ,  $l^* \leftarrow l$  ▷ Initialize the result
3: while  $i < N_{\text{iters}}$  do
4:   Randomly sample  $M$ 's node parameters to obtain  $M'$ 
5:    $l' \leftarrow L(I, R(M'))$ 
6:   if  $l' < l$  then
7:      $M \leftarrow M'$ ,  $l \leftarrow l'$  ▷ Accept a better solution
8:   else
9:     Set  $M \leftarrow M'$ ,  $l \leftarrow l'$  at a small probability  $p_{\text{acc}}$  ▷ Otherwise, accept stochastically
10:  end if
11:  if  $l < l^*$  then
12:     $M^* \leftarrow M$ ,  $l^* \leftarrow l$  ▷ Update the best solution
13:  end if
14:   $i \leftarrow i + 1$ 
15: end while
```

We use an AdamW optimizer [173] with a 1e-4 learning rate and a cosine annealing schedule. The initial linear warmup period accounts for 3% of training steps. To effectively utilize GPU memory, we apply FlashAttention-2 [174] and train the model in BF16 precision on 8× NVIDIA H100 80GB GPUs using DeepSpeed ZeRO-3 [175, 176]. Each GPU accommodates a batch size of 4 training samples, resulting in an overall batch size of 32. The fine-tuning process lasts 5 epochs over three days.

6.1.4 Post-Optimization

After predicting a plausible procedural material with the VLM, we can further improve the perceptual match against the input image by refining its node parameters. This post-optimization step is commonly used in previous works concerning Adobe Substance 3D materials [5, 84, 92]. In particular, they leverage a differentiable implementation of the node graph to backpropagate analytical gradients from a perceptual loss based on the Gram matrices of VGG features [148]. However, Blender procedural materials currently do not have differentiable counterparts, nor is it feasible to train individual differentiable proxies [84] for hundreds of built-in and custom node types.

Instead, we opt for a gradient-free, stochastic local parameter search based on the Markov Chain Monte Carlo (MCMC) algorithm [177, 178], which has been applied to inductive program synthesis [179]. Let $\tilde{M}(\Phi)$ be a predicted procedural material with node parameters Φ that yields an output rendering image $\tilde{I} = R(\tilde{M})$, $L(I, \tilde{I})$ be the perceptual loss function between the input image I and the prediction $\tilde{I}$. Our MCMC algorithm translates into the pseudocode in Algorithm 1. We run MCMC sampling for each predicted material for $N_{\text{iters}} = 200$ iterations, accepting a worse sample in each iteration at a small probability $p_{\text{acc}} = 0.05$.

Implementation-wise, we largely follow the same node parameter sampling method as data augmentation (Section 6.1.2) for MCMC sampling. However, we only randomly sample 10% of available node parameters in each iteration. Furthermore, we constrain the relative uniform sampling interval for ranged parameters to $\pm 20\%$ (including colors) and the sampling probability

of categorical parameters to 0.2. These practices prompt the MCMC algorithm to make smaller steps in each iteration, improving the likelihood of converging to a better solution. We also ignore the program length limit during post-optimization. Therefore, the node parameters are no longer differentiated based on default values.

6.2 Experiment

6.2.1 Evaluation Setup

Datasets. We evaluate our method on in-distribution and out-of-distribution test images collected from three different sources: 1) 44 synthetic materials from *Blender* [2], randomly selected and separated from our training dataset (before augmentation); 2) 64 synthetic materials from *Substance* [1], which are randomly sampled from the evaluation set of Hu et al. [92]; 3) 64 real photographs gathered from [5] and [52], captured using smart phone cameras. We note that the node graph system of AS3D is more expressive than Blender and capable of creating more intricate material appearances. Therefore, although the *Substance* test set comprises synthetic images, we consider them an out-of-distribution test set harder than *Blender*.

Baselines. We compare our method to four baselines: GPT-4o-mini, nearest neighbor retrieval, Conditional MatFormer [92] and BlenderAlchemy [136]. For GPT-4o-mini, we directly query the model with the input image and a carefully designed prompt.

You are familiar with creating procedural materials using Blender's Python API. You will be given an image that describes a material appearance. Your task is to write a Python function ``shader_material`` that creates a Blender procedural material to match the appearance of the image when rendered on a flat surface. Write your code following the guidelines below.

Code template:

```
```python
import bpy

def shader_material(material: bpy.types.Material):
 material.use_nodes = True
 nodes = material.node_tree.nodes
 links = material.node_tree.links

 # Create nodes
 # YOUR CODE HERE

 # Create links to connect nodes
 # YOUR CODE HERE

 # Set parameters for each node
```

```
YOUR CODE HERE
...
```

Rules:

1. Create no more than 30 nodes.
2. Make sure your code can be correctly executed in Blender 3.3. Refer to the Blender Python API documentation for valid node types and parameters.
3. Simply reply with code. Exclude any additional text or explanations.

For nearest neighbor retrieval, we retrieve the nearest material from our original training dataset based on the style loss [148] between its rendered image and the input image. Conditional MatFormer [92] trains a custom transformer on 446K procedural material programs (after parameter augmentation) from the AS3D asset repository. Since its code is not publicly available, we asked the authors to run their trained model on our test sets. BlenderAlchemy [136] starts with an initial procedural material program and iteratively queries a VLM to modify the program to better match the input image. We run their method starting from the nearest neighbor retrieval result and process VLM queries with GPT-4o-mini. We compare all methods without performing post-optimization.

**Evaluation protocol.** Similar to [92], we generate at most  $N$  programs using our model and stop once obtaining  $K < N$  valid programs. We then pick the program with the lowest style loss [148] between the input and rendered images as the final result. Following Section 6.1.1, a generated program is considered valid if it successfully executes and produces an output rendering above the 12KB JPEG file size threshold. For a fair comparison, we evaluate GPT-4o-mini and Conditional MatFormer using the same protocol. We use  $N = 50$  and  $K = 20$  for the following quantitative and qualitative results. Additional comparison results under a tighter sample budget ( $N = 20, K = 4$ ) are included in Table B.1.

**Evaluation metric.** To measure the perceptual similarity between the rendered image of a generated program and the input image, we use three metrics—style loss [148], sliced Wasserstein distance (SWD) [180], and CLIP cosine similarity [161]. The style loss is a widely adopted metric for texture distance. Following prior work [92], we compute the style loss using the L1 difference of Gram matrices of VGG features [154] plus the L1 difference of  $16 \times 16$  downsampled images (weighted by 0.1). SWD calculates the distance between two images based on the optimal transport theory [181], which has been demonstrated for neural texture synthesis [163]. In addition, the CLIP metric measures the semantic difference of textures as a complement.

Aside from appearance matching, we also evaluate the model’s ability to produce valid procedural material programs. Inheriting the evaluation protocol above, assuming we run the model  $n \leq N$  times and get  $k \leq K$  valid programs for a test image, we define a *correctness* score as  $k/n$ , i.e., the fraction of valid program predictions. This metric only applies to our method and GPT-4o-mini, since Conditional MatFormer represents procedural materials as customized token sequences and implements an inference-time grammar checker to ensure the generated program is executable.



Table 6.1: A quantitative comparison of our method with several baselines on in-distribution (Blender) and out-of-distribution (Substance and real images) datasets. For a fair comparison, the numbers for our approach here are before post optimization. Red: best score, Orange: second best score.

Dataset	Method	Style loss ↓	SWD ↓	CLIP ↑	Program correctness ↑
Blender	GPT-4o-mini	0.041	3.478	0.728	0.294
	Cond. MatFormer	0.034	2.828	0.686	—
	BlenderAlchemy	0.027	2.381	0.777	—
	Nearest Neighbor	0.026	2.123	0.778	—
	Ours	0.019	1.760	0.856	0.911
Substance	GPT-4o-mini	0.039	3.722	0.680	0.227
	Cond. MatFormer	0.033	2.366	0.736	—
	BlenderAlchemy	0.029	2.727	0.690	—
	Nearest Neighbor	0.027	2.546	0.720	—
	Ours	0.026	2.283	0.762	0.890
Real images	GPT-4o-mini	0.035	4.252	0.642	0.248
	Cond. MatFormer	0.028	2.872	0.684	—
	BlenderAlchemy	0.025	2.682	0.700	—
	Nearest Neighbor	0.021	2.487	0.700	—
	Ours	0.025	2.417	0.722	0.870

## 6.2.2 Results

**Material generation quality.** We report quantitative comparison results in Table 6.1, and show qualitative examples in Figure 6.3 and Figure B.1. GPT-4o-mini struggles to produce valid material programs. Conditional MatFormer [92] performs reasonably well on the Substance dataset (on which it was trained) but shows limited generalization to the other test sets. Nearest neighbor retrieval obtains competitive quantitative scores, which agrees with the findings in previous work [92]. However, most retrieved materials are visually sub-optimal against the input. BlenderAlchemy [136] often produces results that do not deviate much from the starting programs. Moreover, it incurs a considerable running cost since each program generation requires dozens of multi-modal queries to the GPT model.

Our method outperforms the baselines on all three test datasets. The generated appearances are structurally and semantically close to the input images. To demonstrate the practical usability of our generated materials, we apply them to 3D meshes and show the rendering results in Figure 6.1.

**Inference time consumption.** We report the average inference time of our method and other baseline methods in Table 6.2. Specifically, for our method and Conditional MatFormer, the time consumption is measured on an NVIDIA H100 80GB GPU at the best throughput. The remaining baselines are timed on an AMD Ryzen 7 5800X 8-core CPU with parallel requests to the OpenAI API. Conditional MatFormer has a high inference throughput due to a small network size. However,

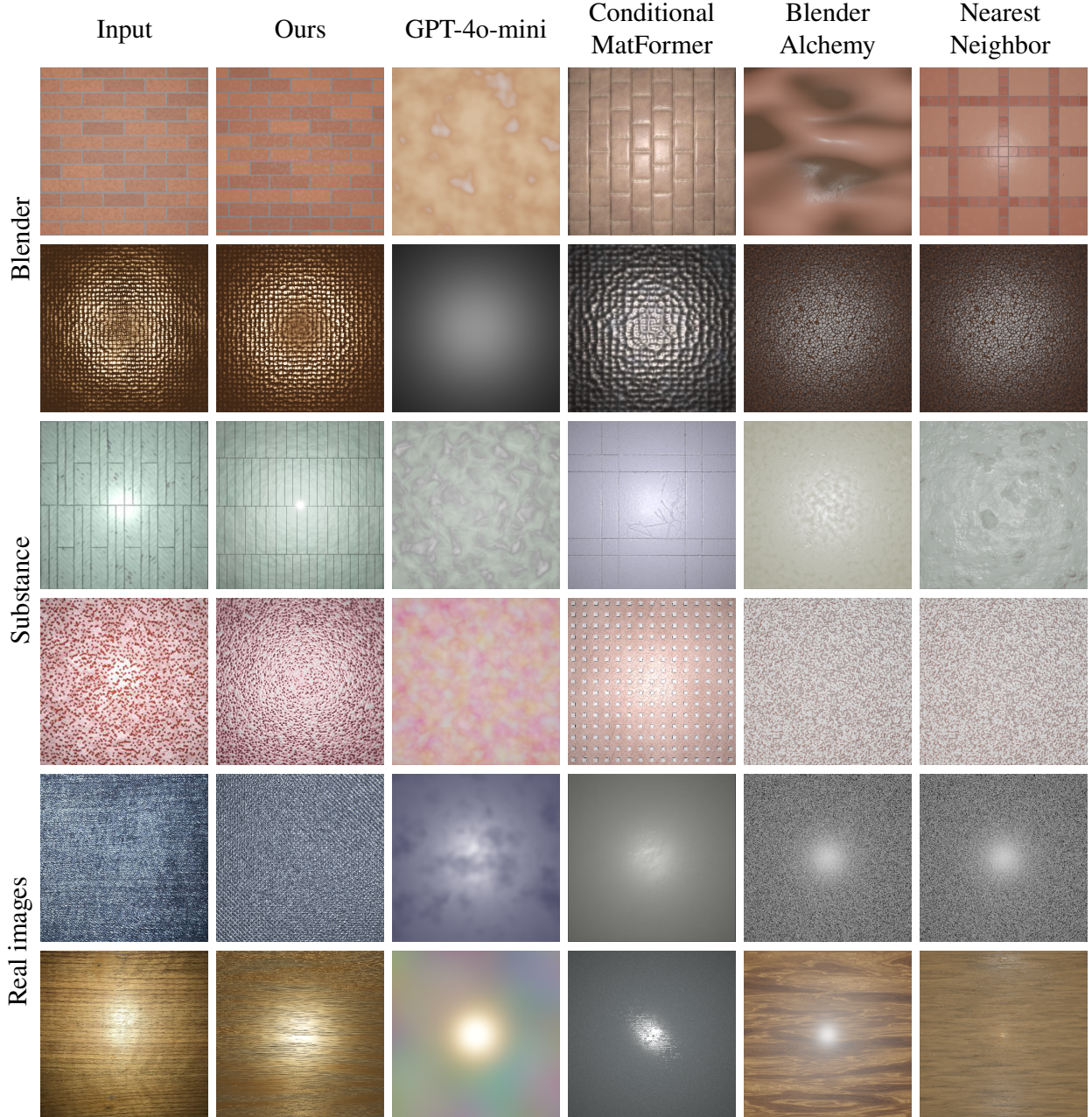


Figure 6.3: A qualitative comparison of our method with several baselines. The input images are shown in the leftmost column. Re-rendered images from our generated procedural material programs best match the input.

even after selecting the top-5 graphs from 150 random samples, their predictions still require a costly post-optimization to match the input [92].

**Performance impact of material complexity.** We plot the average style loss and program correctness for VLM-generated materials with varied complexity in Figure 6.4. The materials are divided into separate bins based on the transpiled program length and the number of nodes. Most

Table 6.2: The inference time consumption of our method and other baselines.

Method	Avg. time (s)	#Samples	Avg. time per sample (s)
GPT-4o-mini	7.5	20	0.38
Cond. MatFormer	4.1	20	0.21
BlenderAlchemy	252.4	32	7.89
Ours	47.9	20	2.40

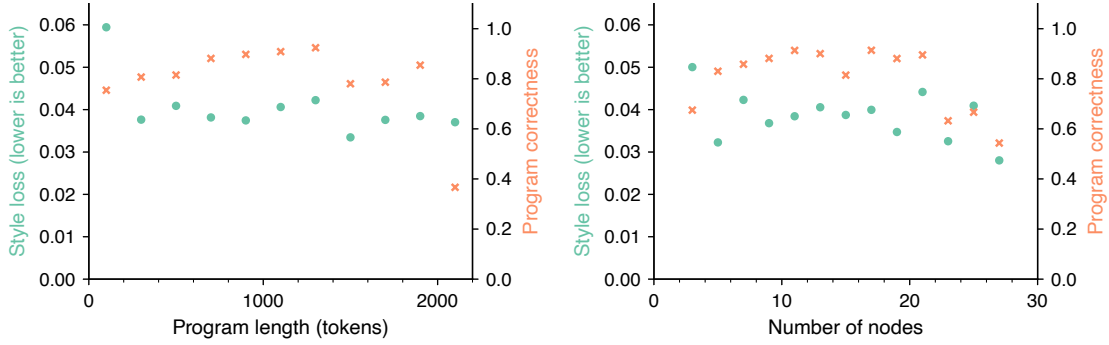


Figure 6.4: The average style loss and program correctness of VLM-generated materials with varied material complexity. The complexity is measured using program lengths in tokens and node graph sizes, respectively.

materials with moderate complexity show comparable style losses and correctness. Interestingly, both metrics are worse for simple materials with fewer than 4 nodes, largely due to invalid or suboptimal choices of custom node groups. Program correctness also declines for materials with more than 22 nodes or 2,000 tokens, implying a higher chance of errors in more intricate materials. As mentioned in Section 6.3, implementing inference-time syntax checking to mask out invalid tokens can guarantee the correctness of generated programs.

### 6.2.3 Ablation Study

In Table 6.3 and Figure 6.5, we compare our method against ablated versions without data augmentation and validate the post-optimization algorithm. Both graph structure augmentation and parameter augmentation are crucial to improving the matching quality on in-distribution and out-of-distribution test sets. Specifically, structure augmentation guides the model to predict more accurate texture patterns, while parameter augmentation enhances the matching of color and other appearance details. As for program correctness, our model has a significantly higher probability of generating valid programs after fine-tuning using the fully augmented dataset. In addition, our results are further improved by the post-optimization step using MCMC local parameter search. For generated programs with sufficient capacity to represent the input image, the post-optimization algorithm refines the overall color and spatial structures, achieving noticeably better alignment with the input.



Table 6.3: Ablation study. - *parameter aug.* removes the graph parameter augmentation, and - *structure aug.* further removes the graph structure augmentation. + *post opt.* optimizes graph parameters in our generated programs using MCMC.

Dataset	Method	Style loss ↓	SWD ↓	CLIP ↑	Program correctness ↑
Blender	- structure aug.	0.032	2.757	0.750	0.603
	- parameter aug.	0.022	1.965	0.839	0.897
	Ours	0.019	1.760	0.856	0.911
	+ post opt.	0.015	1.701	0.851	—
Substance	- structure aug.	0.033	3.084	0.727	0.487
	- parameter aug.	0.030	2.482	0.744	0.767
	Ours	0.026	2.283	0.762	0.890
	+ post opt.	0.020	2.184	0.748	—
Real images	- structure aug.	0.030	3.308	0.688	0.342
	- parameter aug.	0.028	2.682	0.707	0.713
	Ours	0.025	2.417	0.722	0.870
	+ post opt.	0.018	2.309	0.732	—

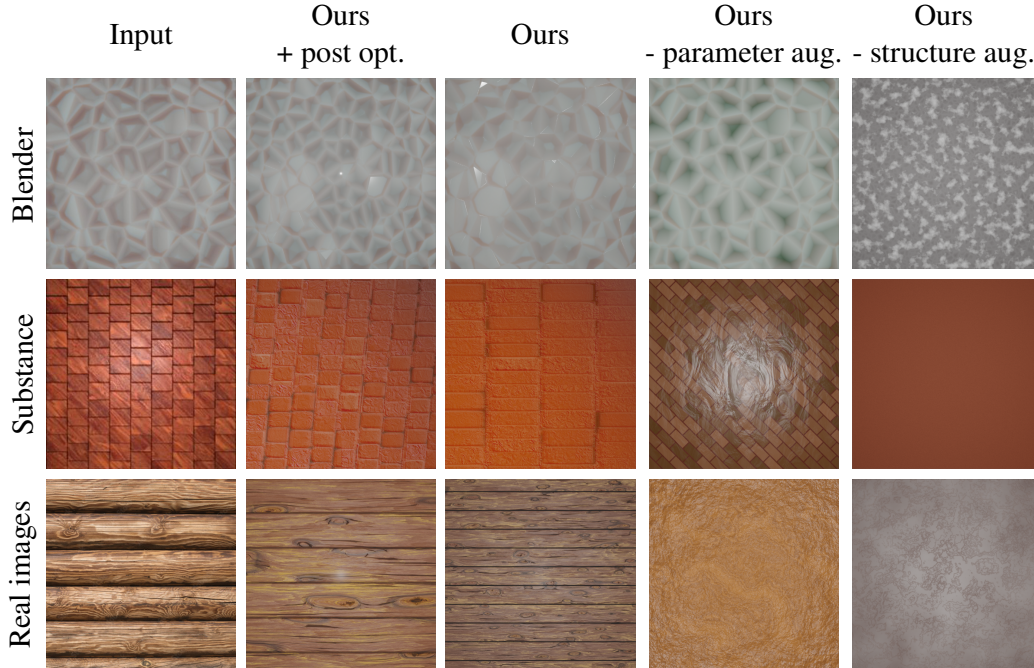


Figure 6.5: Visual examples for ablation study.

## 6.2.4 User study

To investigate the practical value of VLM-generated procedural materials to an artist’s creation process, we hosted a user study among professional artists who specialize in Blender material graphs

Table 6.4: Guidelines for interpreting the usability ratings of algorithm-generated procedural materials in our user study. Intermediate ratings indicate preferences between adjacent descriptions.

Rating	Description
1	Barely useful. It is better to create the material manually.
4	Partially reusable as a template. Many sections need to be reworked.
7	A good starting point, requiring careful but not too cumbersome post-editing.
10	An ideal solution. Very few edits are necessary.

Table 6.5: User study on the practical usability of procedural materials generated with our fine-tuned VLM and BlenderAlchemy [136]. We report the average 2AFC preference score and usability rating over different test sets. Numbers after the " $\pm$ " symbol are standard deviations.

<b>Preference</b> (Ours over BlenderAlchemy)	Blender	Substance	Real images	<b>Average</b>
Visual quality	92%	95%	86%	91%
Node graph quality	92%	91%	98%	94%
<b>Usability Rating</b> (1-10)				
BlenderAlchemy	4.1 $\pm$ 1.7	3.9 $\pm$ 1.9	4.5 $\pm$ 1.5	4.2 $\pm$ 1.7
Ours	7.9 $\pm$ 1.2	5.6 $\pm$ 1.7	6.9 $\pm$ 1.0	6.8 $\pm$ 1.3

and researchers with sufficient domain expertise. The participants are tasked with comparing the materials generated from our method and BlenderAlchemy [136] for 12 randomly selected test images (4 images from each test set). Each comparison involves the following items:

- **Preference in visual quality.** The participant chooses which method ("A" or "B") produces a better visual match with the input image. We adopt the two-alternative forced choice (2AFC) approach and randomly shuffle the results to conceal the source algorithms.
- **Preference in node graph quality.** In the same 2AFC setting, we show the participant a snapshot of the generated material node graph from each method and ask them to determine which graph requires simpler user post-editing to match the input image.
- **Usability rating.** The participant rates the usability of each generated material on a scale from 1 to 10 based on how much it substitutes an artist’s creation process. The interpretations of usability ratings are provided in Table 6.4.

We collected 16 responses from 8 professional artists and 8 researchers, respectively. Then, we calculated the average preference percentage for our method over BlenderAlchemy and the average usability rating for both methods in Table 6.5. In more than 90% of the test cases, participants prefer our generated materials over BlenderAlchemy in visual and node graph quality. Our method achieves an average usability rating of 6.8, indicating that artists generally consider VLM-generated materials efficient starting points for post-editing. In addition, some artists praised our method for the interesting results and noted how our method alleviates the core issues with complex shader



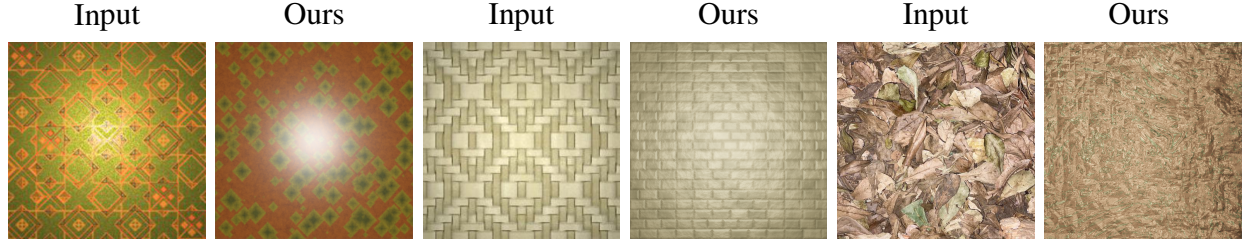


Figure 6.6: Failure cases. The first two examples (Column 1-4) come from the *Substance* test set. The last example (Column 5-6) is from the real image test set.

graphs compared with image-based textures. They also appreciated the better reusability of simpler graph structures in production environments.

### 6.3 Discussion and Conclusion

Figure 6.6 demonstrates a few failure cases of our method where the intricate texture pattern of the input image exceeds the expressiveness of existing Blender node types (including artist-created node groups) within a reasonable node graph size (e.g., we use a limit of 30 nodes). In such cases, our model may struggle to predict the best material graph structure to approximate the input texture patterns. Extending the node graph system with custom node groups from a broader range of artist-created materials will alleviate the limitation in expressiveness. An alternative is to increase the graph size limit and the VLM’s context window, although this requires additional engineering efforts in data augmentation to generate more feature-rich materials using an LLM.

In conclusion, we present a single-image procedural material generation method that fine-tunes a pre-trained large VLM to predict Blender procedural materials in Python code format. Our approach harnesses the embedded knowledge of the VLM through domain specialization, enabled by the first open-source procedural material dataset with more than 550K image-program pairs. We report superior perceptual matching quality on out-of-distribution synthetic and real input images, outperforming existing models trained on proprietary materials or leveraging zero-shot VLM inference. We hope the release of our dataset and model will benefit future research on visual program synthesis and inverse procedural graphics.

# Chapter 7

## Conclusion and Future Work

This thesis presents my latest research on inverse procedural material modeling for appearance capture from single cellphone flash photos. The incorporated works substantially enhance the perceptual match between the generated materials and the input images through innovative dataset generation, training, and post-optimization approaches. First, I introduced differentiable implementations of generic noise/pattern generator nodes and pixel-wise shading nodes to a differentiable material graph framework [5]. This enabled the co-optimization of continuous and discrete parameters in a material graph without neural network approximations for the generator nodes. Then, I leveraged RL fine-tuning to train a node parameter generation model with visual feedback, significantly reducing the reliance on costly post-optimization to remediate the color and pattern mismatches due to inaccurate parameter predictions. Finally, I demonstrated material graph generation using a fine-tuned large VLM as a superior alternative to custom, pre-trained node and edge generation models [92]. The procedural material dataset used for VLM fine-tuning was augmented using a commercial LLM to produce diverse material graph structures.

In addition to establishing a new state of the art for single-image procedural material generation, my thesis offers inspiring directions for future works.

**RL fine-tuning for material graph generation.** An attractive, albeit more challenging, future direction is to improve node and edge generation with RL [120]. Indeed, the sample space of plausible graph structures is considerably broader, resulting in many possible local minima. A strong regularization metric is thus necessary to guide the generation of graph structures and ensure that the generated graph structures represent semantically meaningful material appearances. Additionally, our RL-based approach enables advanced test-time search techniques guided by the learned value network [182], which is impossible for supervised training approaches. This may yield better parameter sequences than greedy search or stochastic sampling. A similar methodology can be applied to text prompts in addition to image conditioning, e.g., using the text-to-image prior approach from Hu et al. [92]

**Generating complex materials with VLMs.** Future works may address the challenge of generating complex visual programs using VLMs from several orthogonal directions. Including more artist-created materials and performing data augmentation using next-generation LLMs with advanced chain-of-thought (CoT) reasoning capabilities will further improve the complexity and diversity of material graphs. Incorporating visual feedback through RL fine-tuning (Chapter 5) is also an

effective approach to improving visual quality and model generalization. Additionally, it is possible to represent Blender procedural materials with a compact domain-specific language (DSL) based on handcrafted rules or library learning. This can produce much shorter transpiled programs than Python and simplify VLM fine-tuning. Moreover, an inference-time syntax checker can be implemented for the DSL to guarantee the correctness of generated programs.

**Extension to other inverse procedural graphics problems.** The methodologies presented in my thesis can potentially benefit a wide range of generative models in procedural graphics, particularly in scenarios where (1) the parameter space of the procedural representation has a weak correlation with the metric space for quality evaluation, or (2) the availability of ground-truth object-program pairs is limited. Two popular examples are procedural 3D shapes [135] and scenes [166]. Future efforts in these areas will foster the development of powerful GenAI tools, which will expedite the creative process of professional artists and democratize modern CG production for casual users.

Last but not least, the following thought process would be helpful when approaching a generic inverse procedural modeling problem:

1. Define a proper metric to evaluate the quality of a procedurally generated output relative to the input data.
2. Assess the complexity and expressiveness of the procedural representation. Conducting heuristic search or training a generative model from scratch using RL [118] might work if a typical procedure contains very few operations (e.g., less than 10).
3. Investigate the availability of existing data in this procedural representation. If the aggregate expressiveness covered by the existing data is sufficient for inverse modeling, running local parameter or program structure search (e.g., using MCMC sampling) from a retrieved program might be worth considering.
4. The aforementioned optimization can utilize gradient information if the procedural representation is differentiable with respect to the control parameters.
5. Depending on training data availability, prepare a comprehensive procedural dataset with various program structures and outputs. Incorporate data augmentation in the program structure space and the parameter space if necessary.
6. Use the training dataset to pre-train a small custom model or fine-tune a pre-trained multimodal LLM. Fine-tuning is preferable especially in data-limited scenarios (e.g., <1M training examples).
7. Further fine-tune the generative model with RL using the predefined metric to improve generation quality.
8. Finally, apply gradient-based or gradient-free post-optimization after model inference to refine the generation results.

# Appendix A

## Additional Results for RL Fine-Tuning

We show additional qualitative results on synthetic and real images in Figure [A.1](#) and Figure [A.2](#) respectively. More visual comparisons with increased sample budgets on the synthetic set dataset and the real photograph test dataset are available in Figure [A.3](#) and Figure [A.4](#).



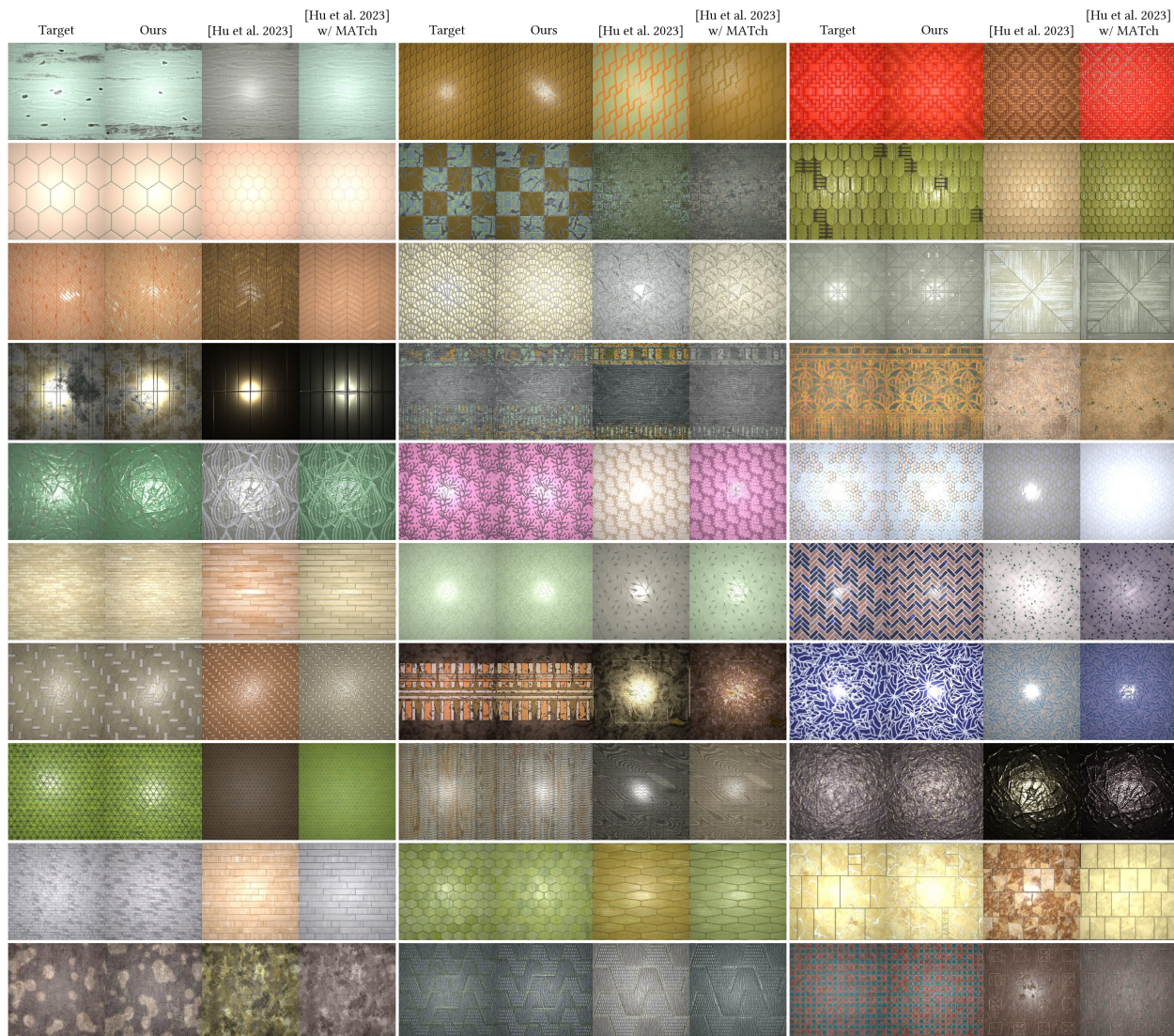


Figure A.1: Additional qualitative results on the synthetic dataset.



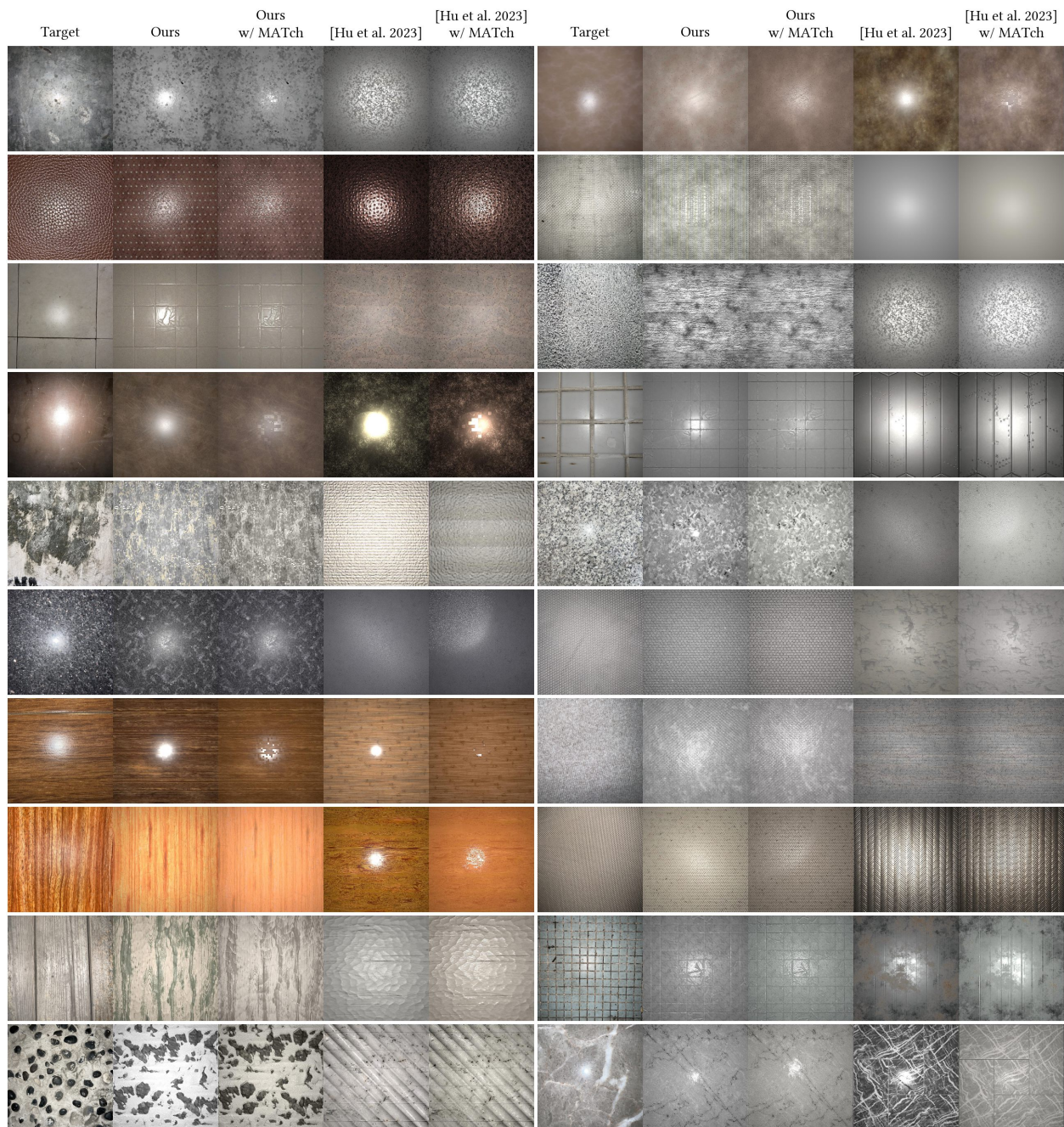


Figure A.2: Additional qualitative results on the real photograph dataset.



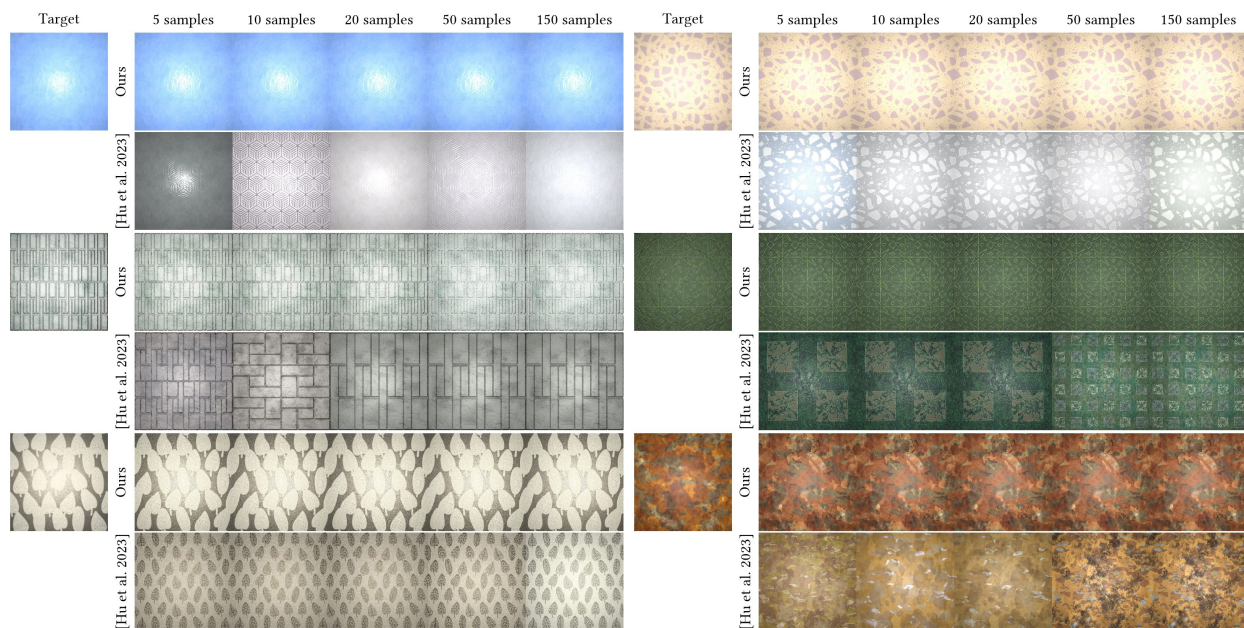


Figure A.3: Additional qualitative results on the synthetic dataset with increasing sample budgets.

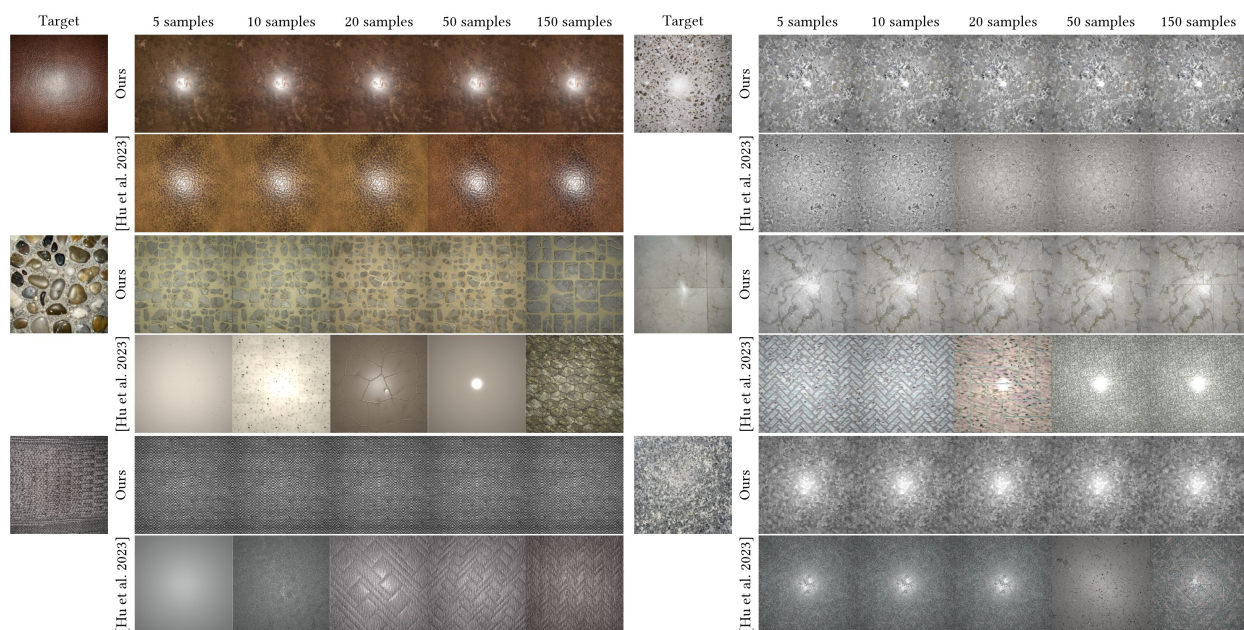


Figure A.4: Additional qualitative results on the real photograph dataset with increasing sample budgets.

# Appendix B

## Additional Results for VLMaterial

**Qualitative comparisons.** We provide additional qualitative examples and ablation study results in Figure B.1 and Figure B.2, respectively.

**Quantative comparison in a limited budget.** We also compare our method and baselines quantitatively using a more limited sample budget during inference in Table B.1, where the prediction result is selected from up to  $K = 4$  valid samples out of  $N = 20$  trials. Our method is affected by the variance in token predictions, which encourages exploration during inference. Fine-tuning the VLM with RL using an image similarity reward will likely reduce the entropy of token predictions and improve generalization in budget-limited scenarios. Nevertheless, RL fine-tuning should carefully balance exploration and exploitation to prevent local minima.

Table B.1: Comparison between our method and baselines under a more constrained sample budget ( $N = 20, K = 4$ ). The results of BlenderAlchemy and Nearest Neighbor are copied from Table 6.1.

Dataset	Method	Style loss ↓	SWD ↓	CLIP ↑	Program correctness ↑
Blender	GPT-4o-mini	0.044	3.618	0.716	0.343
	Cond. MatFormer	0.042	3.019	0.682	—
	BlenderAlchemy	0.027	2.381	0.777	—
	Nearest Neighbor	0.026	2.123	0.778	—
	Ours	0.025	1.978	0.833	0.926
Substance	GPT-4o-mini	0.042	3.799	0.678	0.234
	Cond. MatFormer	0.040	2.281	0.742	—
	BlenderAlchemy	0.029	2.727	0.690	—
	Nearest Neighbor	0.027	2.546	0.720	—
	Ours	0.031	2.431	0.750	0.926
Real images	GPT-4o-mini	0.040	4.369	0.636	0.241
	Cond. MatFormer	0.037	2.869	0.690	—
	BlenderAlchemy	0.025	2.682	0.700	—
	Nearest Neighbor	0.021	2.487	0.700	—
	Ours	0.030	2.566	0.716	0.880



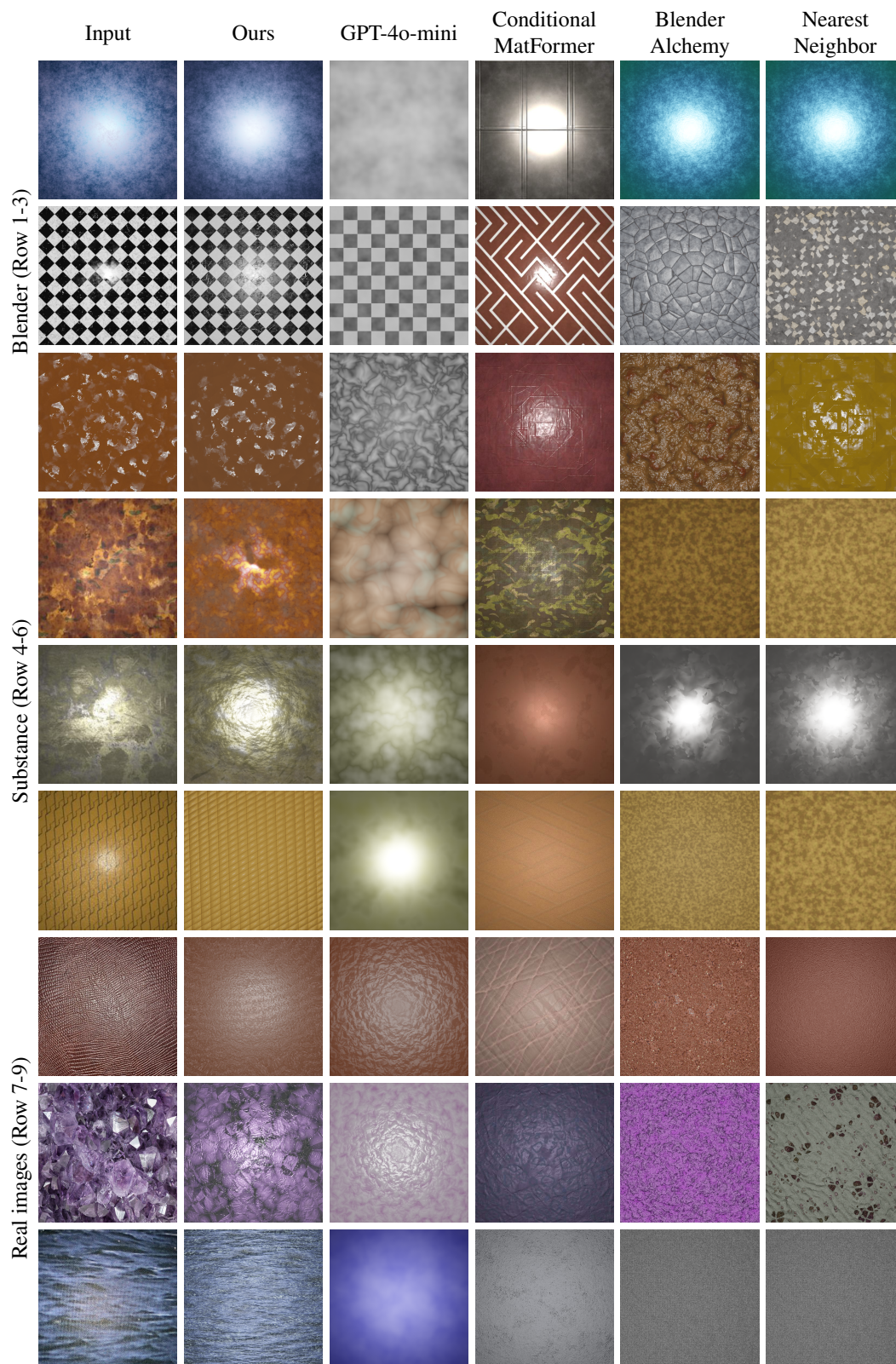


Figure B.1: More qualitative comparisons with baselines.



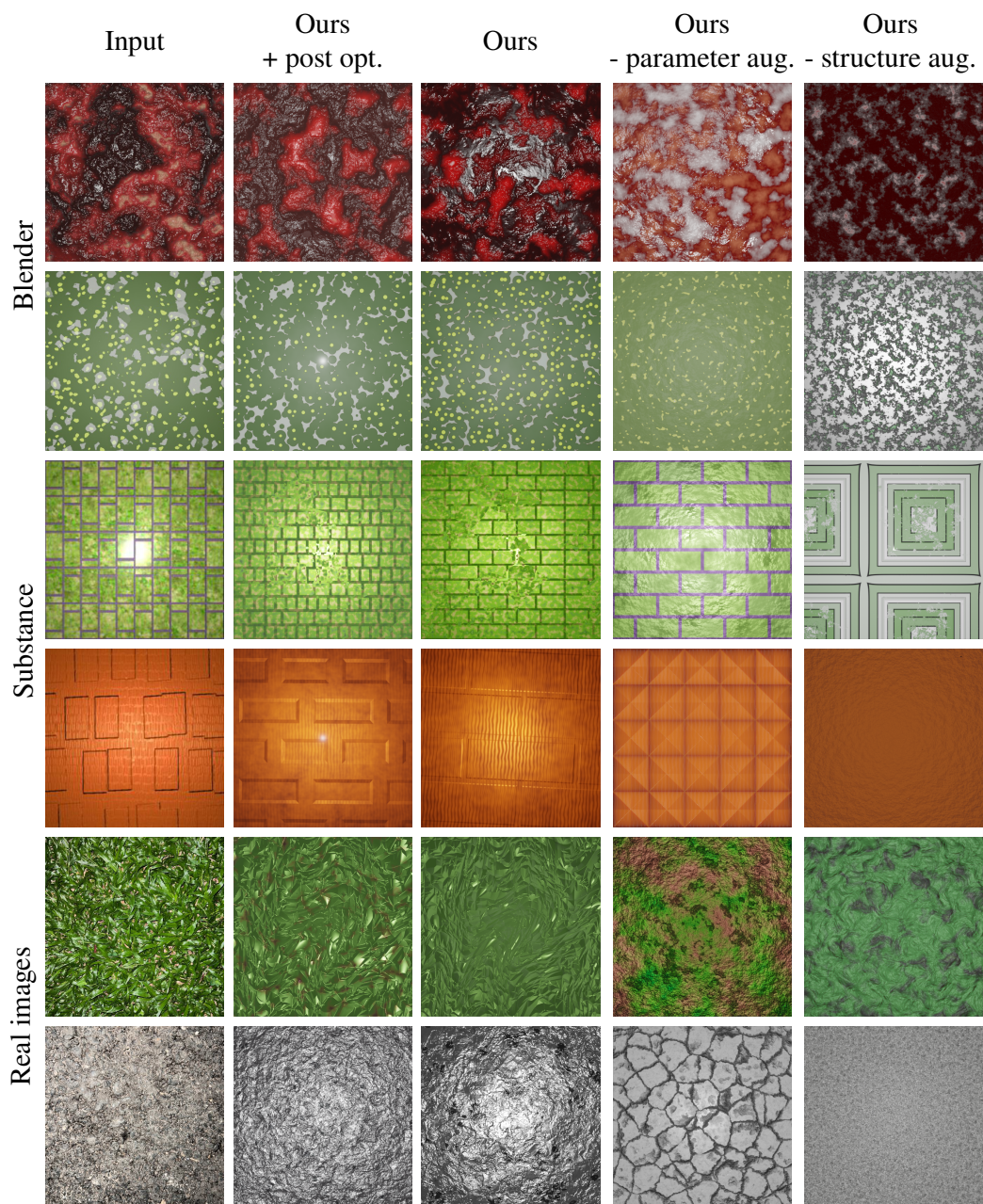


Figure B.2: More examples for ablation study.

# References

- [1] Adobe. *Adobe Substance 3D Designer*. 2024. URL: <https://www.adobe.com/products/substance3d>.
- [2] Blender. *Blender - a 3D modelling and rendering package*. Blender Foundation. 2024. URL: <http://www.blender.org>.
- [3] Autodesk. *Autodesk Maya*. 2024. URL: <https://www.autodesk.com/products/maya/overview>.
- [4] Epic. *Unreal Engine*. Epic Games. 2024. URL: <https://www.unrealengine.com>.
- [5] L. Shi, B. Li, M. Hašan, K. Sunkavalli, T. Boubekur, R. Mech, and W. Matusik. “Match: Differentiable material graphs for procedural material capture”. In: *ACM Transactions on Graphics (TOG)* 39.6 (2020), pp. 1–15.
- [6] D. Guarnera, G. C. Guarnera, A. Ghosh, C. Denk, and M. Glencross. “BRDF representation and acquisition”. In: *Computer Graphics Forum*. Vol. 35. 2. Wiley Online Library. 2016, pp. 625–650.
- [7] K. J. Dana, B. Van Ginneken, S. K. Nayar, and J. J. Koenderink. “Reflectance and texture of real-world surfaces”. In: *ACM Transactions On Graphics (TOG)* 18.1 (1999), pp. 1–34.
- [8] M. Aittala, T. Weyrich, and J. Lehtinen. “Practical SVBRDF capture in the frequency domain.” In: *ACM Trans. Graph.* 32.4 (2013), pp. 110–1.
- [9] J. Lawrence, A. Ben-Artzi, C. DeCoro, W. Matusik, H. Pfister, R. Ramamoorthi, and S. Rusinkiewicz. “Inverse shade trees for non-parametric material representation and editing”. In: *ACM Transactions on Graphics (ToG)* 25.3 (2006), pp. 735–745.
- [10] J. Yu, Z. Xu, M. Mannino, H. W. Jensen, and R. Ramamoorthi. “Sparse sampling for image-based SVBRDF acquisition”. In: *Image* 1 (2016), p. 1.
- [11] M. Holroyd, J. Lawrence, and T. Zickler. “A coaxial optical scanner for synchronous acquisition of 3D geometry and surface reflectance”. In: *ACM Transactions on Graphics (TOG)* 29.4 (2010), pp. 1–12.
- [12] A. Ghosh, T. Hawkins, P. Peers, S. Frederiksen, and P. Debevec. “Practical modeling and acquisition of layered facial reflectance”. In: *ACM SIGGRAPH Asia 2008 papers*. 2008, pp. 1–10.
- [13] A. Ghosh, T. Chen, P. Peers, C. A. Wilson, and P. Debevec. “Circularly polarized spherical illumination reflectometry”. In: *ACM SIGGRAPH Asia 2010 papers*. 2010, pp. 1–12.
- [14] G. Nam, J. H. Lee, H. Wu, D. Gutierrez, and M. H. Kim. “Simultaneous acquisition of microscale reflectance and normals.” In: *ACM Trans. Graph.* 35.6 (2016), pp. 185–1.

- [15] K. Kang, C. Xie, C. He, M. Yi, M. Gu, Z. Chen, K. Zhou, and H. Wu. “Learning efficient illumination multiplexing for joint capture of reflectance and shape.” In: *ACM Trans. Graph.* 38.6 (2019), pp. 165–1.
- [16] B. Tunwattanapong, G. Fyffe, P. Graham, J. Busch, X. Yu, A. Ghosh, and P. Debevec. “Acquiring reflectance and shape from continuous spherical harmonic illumination”. In: *ACM Transactions on graphics (TOG)* 32.4 (2013), pp. 1–12.
- [17] A. Ghosh, T. Chen, P. Peers, C. A. Wilson, and P. Debevec. “Estimating specular roughness and anisotropy from second order spherical gradient illumination”. In: *Computer Graphics Forum*. Vol. 28. 4. Wiley Online Library. 2009, pp. 1161–1170.
- [18] J. Wang, S. Zhao, X. Tong, J. Snyder, and B. Guo. “Modeling anisotropic surface reflectance with example-based microfacet synthesis”. In: *ACM SIGGRAPH 2008 papers*. 2008, pp. 1–9.
- [19] D. B. Goldman, B. Curless, A. Hertzmann, and S. M. Seitz. “Shape and spatially-varying brdfs from photometric stereo”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 32.6 (2009), pp. 1060–1071.
- [20] L.-P. Asselin, D. Laurendeau, and J.-F. Lalonde. “Deep SVBRDF estimation on real materials”. In: *2020 International Conference on 3D Vision (3DV)*. IEEE. 2020, pp. 1157–1166.
- [21] P. Ren, J. Wang, J. Snyder, X. Tong, and B. Guo. “Pocket reflectometry”. In: *ACM Transactions on Graphics (TOG)* 30.4 (2011), pp. 1–10.
- [22] A. Gardner, C. Tchou, T. Hawkins, and P. Debevec. “Linear light source reflectometry”. In: *ACM Transactions on Graphics (TOG)* 22.3 (2003), pp. 749–758.
- [23] M. Aittala, T. Weyrich, J. Lehtinen, et al. “Two-shot SVBRDF capture for stationary materials.” In: *ACM Trans. Graph.* 34.4 (2015), pp. 110–1.
- [24] M. Aittala, T. Aila, and J. Lehtinen. “Reflectance modeling by neural texture synthesis”. In: *ACM Transactions on Graphics (ToG)* 35.4 (2016), pp. 1–13.
- [25] J. Riviere, P. Peers, and A. Ghosh. “Mobile surface reflectometry”. In: *ACM SIGGRAPH 2014 Posters*. 2014, pp. 1–1.
- [26] Z. Zhou, G. Chen, Y. Dong, D. Wipf, Y. Yu, J. Snyder, and X. Tong. “Sparse-as-possible SVBRDF acquisition”. In: *ACM Transactions on Graphics (TOG)* 35.6 (2016), pp. 1–12.
- [27] H. Wu, Z. Wang, and K. Zhou. “Simultaneous localization and appearance estimation with a consumer RGB-D camera”. In: *IEEE transactions on visualization and computer graphics* 22.8 (2015), pp. 2012–2023.
- [28] Z. Hui, K. Sunkavalli, J.-Y. Lee, S. Hadap, J. Wang, and A. C. Sankaranarayanan. “Reflectance capture using univariate sampling of brdfs”. In: *Proceedings of the IEEE International Conference on Computer Vision*. 2017, pp. 5362–5370.
- [29] Z. Xu, J. B. Nielsen, J. Yu, H. W. Jensen, and R. Ramamoorthi. “Minimal BRDF sampling for two-shot near-field reflectance acquisition”. In: *ACM Transactions on Graphics (TOG)* 35.6 (2016), pp. 1–12.



- [30] G. Nam, J. H. Lee, D. Gutierrez, and M. H. Kim. “Practical svbrdf acquisition of 3d objects with unstructured flash photography”. In: *ACM Transactions on Graphics (ToG)* 37.6 (2018), pp. 1–12.
- [31] Y. Dong, G. Chen, P. Peers, J. Zhang, and X. Tong. “Appearance-from-motion: Recovering spatially varying surface reflectance under unknown lighting”. In: *ACM Transactions on Graphics (TOG)* 33.6 (2014), pp. 1–12.
- [32] Y. Dong, J. Wang, X. Tong, J. Snyder, Y. Lan, M. Ben-Ezra, and B. Guo. “Manifold bootstrapping for SVBRDF capture”. In: *ACM Transactions on Graphics (TOG)* 29.4 (2010), pp. 1–10.
- [33] X. Li, Y. Dong, P. Peers, and X. Tong. “Modeling surface appearance from a single photograph using self-augmented convolutional neural networks”. In: *ACM Transactions on Graphics (ToG)* 36.4 (2017), pp. 1–11.
- [34] W. Ye, X. Li, Y. Dong, P. Peers, and X. Tong. “Single image surface appearance modeling with self-augmented cnns and inexact supervision”. In: *Computer Graphics Forum*. Vol. 37. 7. Wiley Online Library. 2018, pp. 201–211.
- [35] V. Deschaintre, M. Aittala, F. Durand, G. Drettakis, and A. Bousseau. “Single-image svbrdf capture with a rendering-aware deep network”. In: *ACM Transactions on Graphics (ToG)* 37.4 (2018), pp. 1–15.
- [36] V. Deschaintre, M. Aittala, F. Durand, G. Drettakis, and A. Bousseau. “Flexible svbrdf capture with a multi-image deep network”. In: *Computer graphics forum*. Vol. 38. 4. Wiley Online Library. 2019, pp. 1–13.
- [37] Z. Li, K. Sunkavalli, and M. Chandraker. “Materials for masses: SVBRDF acquisition with a single mobile phone image”. In: *Proceedings of the European conference on computer vision (ECCV)*. 2018, pp. 72–87.
- [38] Z. Li, Z. Xu, R. Ramamoorthi, K. Sunkavalli, and M. Chandraker. “Learning to reconstruct shape and spatially-varying reflectance from a single image”. In: *ACM Transactions on Graphics (TOG)* 37.6 (2018), pp. 1–11.
- [39] D. Gao, X. Li, Y. Dong, P. Peers, K. Xu, and X. Tong. “Deep inverse rendering for high-resolution SVBRDF estimation from an arbitrary number of images.” In: *ACM Trans. Graph.* 38.4 (2019), pp. 134–1.
- [40] Y. Guo, C. Smith, M. Hašan, K. Sunkavalli, and S. Zhao. “MaterialGAN: Reflectance capture using a generative SVBRDF model”. In: *arXiv preprint arXiv:2010.00114* (2020).
- [41] P. Henzler, V. Deschaintre, N. J. Mitra, and T. Ritschel. “Generative modelling of BRDF textures from flash images”. In: *arXiv preprint arXiv:2102.11861* (2021).
- [42] V. Deschaintre, G. Drettakis, and A. Bousseau. “Guided fine-tuning for large-scale material transfer”. In: *Computer Graphics Forum*. Vol. 39. 4. Wiley Online Library. 2020, pp. 91–105.
- [43] X. Zhou and N. K. Kalantari. “Adversarial Single-Image SVBRDF Estimation with Hybrid Training”. In: *Computer Graphics Forum*. Vol. 40. 2. Wiley Online Library. 2021, pp. 315–325.

- [44] G. Vecchio, S. Palazzo, and C. Spampinato. “Surfacenet: Adversarial svbrdf estimation from a single image”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2021, pp. 12840–12848.
- [45] J. Guo, S. Lai, C. Tao, Y. Cai, L. Wang, Y. Guo, and L.-Q. Yan. “Highlight-aware two-stream network for single-image SVBRDF acquisition”. In: (2021).
- [46] X. Zhou and N. K. Kalantari. “Look-ahead training with learned reflectance loss for single-image svbrdf estimation”. In: *ACM Transactions on Graphics (TOG)* 41.6 (2022), pp. 1–12.
- [47] M. Fischer and T. Ritschel. “Metappearance: Meta-learning for visual appearance reproduction”. In: *ACM Transactions on Graphics (TOG)* 41.6 (2022), pp. 1–13.
- [48] R. Martin, A. Roullier, R. Rouffet, A. Kaiser, and T. Boubekeur. “MaterIA: Single Image High-Resolution Material Capture in the Wild”. In: *Computer Graphics Forum*. Vol. 41. 2. Wiley Online Library. 2022, pp. 163–177.
- [49] X. Zhou, M. Hasan, V. Deschaintre, P. Guerrero, K. Sunkavalli, and N. K. Kalantari. “Tilegen: Tileable, controllable material generation and capture”. In: *SIGGRAPH Asia 2022 conference papers*. 2022, pp. 1–9.
- [50] G. Vecchio, R. Martin, A. Roullier, A. Kaiser, R. Rouffet, V. Deschaintre, and T. Boubekeur. “Controlmat: A controlled generative approach to material capture”. In: *ACM Transactions on Graphics* 43.5 (2024), pp. 1–17.
- [51] G. Vecchio, R. Sortino, S. Palazzo, and C. Spampinato. “Matfuse: controllable material generation with diffusion models”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2024, pp. 4429–4438.
- [52] X. Zhou, M. Hasan, V. Deschaintre, P. Guerrero, Y. Hold-Geoffroy, K. Sunkavalli, and N. K. Kalantari. “Photomat: A material generator learned from single flash photos”. In: *ACM SIGGRAPH 2023 Conference Proceedings*. 2023, pp. 1–11.
- [53] D. Ebert. *Texturing & Modeling, A procedural Approach*. Morgan Kaufman, 2002.
- [54] A. Lagae, S. Lefebvre, R. Cook, T. DeRose, G. Drettakis, D. S. Ebert, J. P. Lewis, K. Perlin, and M. Zwicker. “A survey of procedural noise functions”. In: *Computer Graphics Forum*. Vol. 29. 8. Wiley Online Library. 2010, pp. 2579–2600.
- [55] K. Perlin. “An image synthesizer”. In: *ACM Siggraph Computer Graphics* 19.3 (1985), pp. 287–296.
- [56] K. Perlin. “Improving noise”. In: *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*. 2002, pp. 681–682.
- [57] G. Wyvill and K. Novins. “Filtered noise and the fourth dimension”. In: *ACM SIGGRAPH 99 Conference abstracts and applications*. 1999, p. 242.
- [58] M. Olano, J. C. Hart, W. Heidrich, B. Mark, and K. Perlin. “Real-time shading languages”. In: *SIGGRAPH*. 2002.
- [59] A. Fournier, D. Fussell, and L. Carpenter. “Computer rendering of stochastic models”. In: *Communications of the ACM* 25.6 (1982), pp. 371–384.

- [60] J. Lewis. “Methods for stochastic spectral synthesis”. In: *Proceedings on Graphics Interface’86/Vision Interface’86*. 1986, pp. 173–179.
- [61] J. P. Lewis. “Generalized stochastic subdivision”. In: *ACM Transactions on Graphics (TOG)* 6.3 (1987), pp. 167–190.
- [62] R. L. Cook and T. DeRose. “Wavelet noise”. In: *ACM Transactions on Graphics (TOG)* 24.3 (2005), pp. 803–811.
- [63] K.-i. Anjyo. “A Simple Spectral Approach to Stochastic Modelling for Natural Objects”. In: (1988).
- [64] M. F. Barnsley, R. L. Devaney, B. B. Mandelbrot, H.-O. Peitgen, D. Saupe, R. F. Voss, and D. Saupe. “Algorithms for random fractals”. In: *The science of fractal images* (1988), pp. 71–136.
- [65] M. F. Barnsley, R. L. Devaney, B. B. Mandelbrot, H.-O. Peitgen, D. Saupe, R. F. Voss, and R. F. Voss. *Fractals in nature: from characterization to simulation*. Springer, 1988.
- [66] G. Y. Gardner. “Simulation of natural scenes using textured quadric surfaces”. In: *Acm Siggraph Computer Graphics* 18.3 (1984), pp. 11–20.
- [67] J. J. Van Wijk. “Spot noise texture synthesis for data visualization”. In: *Proceedings of the 18th annual conference on Computer graphics and interactive techniques*. 1991, pp. 309–318.
- [68] A. Goldberg, M. Zwicker, and F. Durand. “Anisotropic noise”. In: *ACM Transactions on Graphics (TOG)* 27.3 (2008), pp. 1–8.
- [69] J.-P. Lewis. “Texture synthesis for digital painting”. In: *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*. 1984, pp. 245–252.
- [70] J.-P. Lewis. “Algorithms for solid noise synthesis”. In: *Proceedings of the 16th annual conference on Computer graphics and interactive techniques*. 1989, pp. 263–270.
- [71] E. Praun, A. Finkelstein, and H. Hoppe. “Lapped textures”. In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. 2000, pp. 465–470.
- [72] A. Lagae, S. Lefebvre, G. Drettakis, and P. Dutré. “Procedural noise using sparse Gabor convolution”. In: *ACM Transactions on Graphics (TOG)* 28.3 (2009), pp. 1–10.
- [73] L. Lefebvre and P. Poulin. “Analysis and synthesis of structural textures”. In: *Graphics Interface*. Vol. 2000. 2000, pp. 77–86.
- [74] J.-M. Dischler, K. Maritaud, B. Lévy, and D. Ghazanfarpour. “Texture particles”. In: *Computer graphics forum*. Vol. 21. 3. Wiley Online Library. 2002, pp. 401–410.
- [75] S. Lefebvre, S. Hornus, and A. Lasram. “By-example synthesis of architectural textures”. In: *ACM SIGGRAPH 2010 papers*. 2010, pp. 1–8.
- [76] C. Eisenacher, C. Tappan, B. Burley, D. Teece, and A. Shek. “Example-based texture synthesis on disney’s tangled”. In: *ACM SIGGRAPH 2010 Talks*. 2010, pp. 1–1.
- [77] B. Galerne, A. Lagae, S. Lefebvre, and G. Drettakis. “Gabor noise by example”. In: *ACM Transactions on Graphics (ToG)* 31.4 (2012), pp. 1–9.
- [78] A. Lagae, P. Vangorp, T. Lenaerts, and P. Dutré. “Procedural isotropic stochastic textures by example”. In: *Computers & Graphics* 34.4 (2010), pp. 312–321.

- [79] Y. Guo, M. Hašan, L. Yan, and S. Zhao. “A bayesian inference framework for procedural material parameter estimation”. In: *Computer Graphics Forum*. Vol. 39. 7. Wiley Online Library. 2020, pp. 255–266.
- [80] G. Gilet, B. Sauvage, K. Vanhoey, J.-M. Dischler, and D. Ghazanfarpour. “Local random-phase noise for procedural texturing”. In: *ACM Transactions on Graphics (ToG)* 33.6 (2014), pp. 1–11.
- [81] B. Galerne, A. Leclaire, and L. Moisan. “Texton noise”. In: *Computer Graphics Forum*. Vol. 36. 8. Wiley Online Library. 2017, pp. 205–218.
- [82] E. Heitz and F. Neyret. “High-performance by-example noise using a histogram-preserving blending operator”. In: *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 1.2 (2018), pp. 1–25.
- [83] Y. Hu, J. Dorsey, and H. Rushmeier. “A novel framework for inverse procedural texture modeling”. In: *ACM Transactions on Graphics (ToG)* 38.6 (2019), pp. 1–14.
- [84] Y. Hu, P. Guerrero, M. Hasan, H. Rushmeier, and V. Deschaintre. “Node graph optimization using differentiable proxies”. In: *ACM SIGGRAPH 2022 conference proceedings*. 2022, pp. 1–9.
- [85] Y.-Y. Yeh, Z. Li, Y. Hold-Geoffroy, R. Zhu, Z. Xu, M. Hašan, K. Sunkavalli, and M. Chandraker. “Photoscene: Photorealistic material and lighting transfer for indoor scenes”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022, pp. 18562–18571.
- [86] K. Yan, F. Luan, M. Hašan, T. Groueix, V. Deschaintre, and S. Zhao. “Psdr-room: Single photo to scene using differentiable rendering”. In: *SIGGRAPH Asia 2023 Conference Papers*. 2023, pp. 1–11.
- [87] Y. Hu, C. He, V. Deschaintre, J. Dorsey, and H. Rushmeier. “An inverse procedural modeling pipeline for svbrdf maps”. In: *ACM Transactions on Graphics (TOG)* 41.2 (2022), pp. 1–17.
- [88] P. Guehl, R. Allegre, J.-M. Dischler, B. Benes, and E. Galin. “Semi-Procedural Textures Using Point Process Texture Basis Functions”. In: *Computer Graphics Forum*. Vol. 39. 4. Wiley Online Library. 2020, pp. 159–171.
- [89] G. Baldi, R. Allègre, and J.-M. Dischler. “Differentiable point process texture basis functions for inverse procedural modeling of cellular stochastic structures”. In: *Computers & Graphics* 112 (2023), pp. 116–131.
- [90] P. Guerrero, M. Hašan, K. Sunkavalli, R. Měch, T. Boubekur, and N. J. Mitra. “Matformer: A generative model for procedural materials”. In: *arXiv preprint arXiv:2207.01044* (2022).
- [91] A. Vaswani. “Attention is all you need”. In: *Advances in Neural Information Processing Systems* (2017).
- [92] Y. Hu, P. Guerrero, M. Hasan, H. Rushmeier, and V. Deschaintre. “Generating Procedural Materials from Text or Image Prompts”. In: *ACM SIGGRAPH 2023 Conference Proceedings*. 2023, pp. 1–11.
- [93] S. Zhang, S. Peng, T. Xu, Y. Yang, T. Chen, N. Xue, Y. Shen, H. Bao, R. Hu, and X. Zhou. “Mapa: Text-driven photorealistic material painting for 3d shapes”. In: *ACM SIGGRAPH 2024 Conference Papers*. 2024, pp. 1–12.



- [94] J. Johnson, B. Hariharan, L. Van Der Maaten, J. Hoffman, L. Fei-Fei, C. Lawrence Zitnick, and R. Girshick. “Inferring and executing programs for visual reasoning”. In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 2989–2998.
- [95] K. Ellis, D. Ritchie, A. Solar-Lezama, and J. Tenenbaum. “Learning to infer graphics programs from hand-drawn images”. In: *Advances in neural information processing systems* 31 (2018).
- [96] K. Ellis, M. Nye, Y. Pu, F. Sosa, J. Tenenbaum, and A. Solar-Lezama. “Write, execute, assess: Program synthesis with a repl”. In: *Advances in Neural Information Processing Systems* 32 (2019).
- [97] Y. Tian, A. Luo, X. Sun, K. Ellis, W. T. Freeman, J. B. Tenenbaum, and J. Wu. “Learning to infer and execute 3d shape programs”. In: *arXiv preprint arXiv:1901.02875* (2019).
- [98] K. Ellis, L. Wong, M. Nye, M. Sable-Meyer, L. Cary, L. Anaya Pozo, L. Hewitt, A. Solar-Lezama, and J. B. Tenenbaum. “DreamCoder: growing generalizable, interpretable knowledge with wake–sleep Bayesian program learning”. In: *Philosophical Transactions of the Royal Society A* 381.2251 (2023), p. 20220050.
- [99] P. Reddy, M. Gharbi, M. Lukac, and N. J. Mitra. “Im2vec: Synthesizing vector graphics without vector supervision”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021, pp. 7342–7351.
- [100] A. Carlier, M. Danelljan, A. Alahi, and R. Timofte. “Deepsvg: A hierarchical generative network for vector graphics animation”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 16351–16361.
- [101] J. A. Rodriguez, S. Agarwal, I. H. Laradji, P. Rodriguez, D. Vazquez, C. Pal, and M. Pedersoli. “Starvector: Generating scalable vector graphics code from images”. In: *arXiv preprint arXiv:2312.11556* (2023).
- [102] G. Sharma, R. Goyal, D. Liu, E. Kalogerakis, and S. Maji. “Csgnet: Neural shape parser for constructive solid geometry”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 5515–5523.
- [103] T. Du, J. P. Inala, Y. Pu, A. Spielberg, A. Schulz, D. Rus, A. Solar-Lezama, and W. Matusik. “Inversecsg: Automatic conversion of 3d models to csg trees”. In: *ACM Transactions on Graphics (TOG)* 37.6 (2018), pp. 1–16.
- [104] K. Kania, M. Zieba, and T. Kajdanowicz. “UCSG-NET-unsupervised discovering of constructive solid geometry tree”. In: *Advances in neural information processing systems* 33 (2020), pp. 8776–8786.
- [105] R. K. Jones, T. Barton, X. Xu, K. Wang, E. Jiang, P. Guerrero, N. J. Mitra, and D. Ritchie. “Shapeassembly: Learning to generate programs for 3d shape structure synthesis”. In: *ACM Transactions on Graphics (TOG)* 39.6 (2020), pp. 1–20.
- [106] M. A. Uy, Y.-Y. Chang, M. Sung, P. Goel, J. G. Lambourne, T. Birdal, and L. J. Guibas. “Point2cyl: Reverse engineering 3d objects from point clouds to extrusion cylinders”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022, pp. 11850–11860.

- [107] R. Wu, C. Xiao, and C. Zheng. “Deepcad: A deep generative network for computer-aided design models”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2021, pp. 6772–6782.
- [108] X. Xu, W. Peng, C.-Y. Cheng, K. D. Willis, and D. Ritchie. “Inferring cad modeling sequences using zone graphs”. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2021, pp. 6062–6070.
- [109] C. Li, H. Pan, A. Bousseau, and N. J. Mitra. “Free2cad: Parsing freehand drawings into cad commands”. In: *ACM Transactions on Graphics (TOG)* 41.4 (2022), pp. 1–16.
- [110] X. Xu, P. K. Jayaraman, J. G. Lambourne, K. D. Willis, and Y. Furukawa. “Hierarchical neural coding for controllable cad model generation”. In: *arXiv preprint arXiv:2307.00149* (2023).
- [111] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, et al. “Training language models to follow instructions with human feedback”. In: *Advances in neural information processing systems* 35 (2022), pp. 27730–27744.
- [112] OpenAI. *ChatGPT: Optimizing Language Models for Dialogue*. 2022.
- [113] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, et al. “Llama 2: Open foundation and fine-tuned chat models”. In: *arXiv preprint arXiv:2307.09288* (2023).
- [114] H. Lee, S. Phatale, H. Mansoor, K. Lu, T. Mesnard, C. Bishop, V. Carbune, and A. Rastogi. “Rlaif: Scaling reinforcement learning from human feedback with ai feedback”. In: *arXiv preprint arXiv:2309.00267* (2023).
- [115] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. H. Hoi. “Coderl: Mastering code generation through pretrained models and deep reinforcement learning”. In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 21314–21328.
- [116] S. Duan, N. Kanakaris, X. Xiao, H. Ping, C. Zhou, N. K. Ahmed, G. Ma, M. Capota, T. L. Willke, S. Nazarian, et al. “Leveraging Reinforcement Learning and Large Language Models for Code Optimization”. In: *arXiv preprint arXiv:2312.05657* (2023).
- [117] P. Shojaei, A. Jain, S. Tipirneni, and C. K. Reddy. “Execution-based code generation using deep reinforcement learning”. In: *arXiv preprint arXiv:2301.13816* (2023).
- [118] Y. Hu, H. He, C. Xu, B. Wang, and S. Lin. “Exposure: A white-box photo post-processing framework”. In: *ACM Transactions on Graphics (TOG)* 37.2 (2018), pp. 1–17.
- [119] A. S. Pinto, A. Kolesnikov, Y. Shi, L. Beyer, and X. Zhai. “Tuning computer vision models with task rewards”. In: *International Conference on Machine Learning*. PMLR. 2023, pp. 33229–33239.
- [120] K. Black, M. Janner, Y. Du, I. Kostrikov, and S. Levine. “Training diffusion models with reinforcement learning”. In: *arXiv preprint arXiv:2305.13301* (2023).
- [121] K. Lee, H. Liu, M. Ryu, O. Watkins, Y. Du, C. Boutilier, P. Abbeel, M. Ghavamzadeh, and S. S. Gu. “Aligning text-to-image models using human feedback”. In: *arXiv preprint arXiv:2302.12192* (2023).

- [122] Y. Fan, O. Watkins, Y. Du, H. Liu, M. Ryu, C. Boutilier, P. Abbeel, M. Ghavamzadeh, K. Lee, and K. Lee. “Reinforcement Learning for Fine-tuning Text-to-Image Diffusion Models”. In: *Thirty-seventh Conference on Neural Information Processing Systems (NeurIPS) 2023*. Neural Information Processing Systems Foundation. 2023.
- [123] D. Xie, J. Li, H. Tan, X. Sun, Z. Shu, Y. Zhou, S. Bi, S. Pirk, and A. E. Kaufman. “Carve3d: Improving multi-view reconstruction consistency for diffusion models with rl finetuning”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2024, pp. 6369–6379.
- [124] A. Khalifa, P. Bontrager, S. Earle, and J. Togelius. “Pcgrl: Procedural content generation via reinforcement learning”. In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. Vol. 16. 1. 2020, pp. 95–101.
- [125] C. E. López, J. Cunningham, O. Ashour, and C. S. Tucker. “Deep reinforcement learning for procedural content generation of 3d virtual environments”. In: *Journal of Computing and Information Science in Engineering* 20.5 (2020), p. 051005.
- [126] S. Mohaghegh, M. A. R. Dehnavi, G. Abdollahinejad, and M. Hashemi. “PCGPT: Procedural Content Generation via Transformers”. In: *arXiv preprint arXiv:2310.02405* (2023).
- [127] L. Gisslén, A. Eakins, C. Gordillo, J. Bergdahl, and K. Tollmar. “Adversarial reinforcement learning for procedural content generation”. In: *2021 IEEE Conference on Games (CoG)*. IEEE. 2021, pp. 1–8.
- [128] D. H. Laidlaw, W. B. Trumbore, and J. F. Hughes. “Constructive solid geometry for polyhedral objects”. In: *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*. 1986, pp. 161–170.
- [129] R. K. Jones, H. Walke, and D. Ritchie. “Plad: Learning to infer shape programs with pseudo-labels and approximate distributions”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022, pp. 9871–9880.
- [130] R. Ramamurthy, P. Ammanabrolu, K. Brantley, J. Hessel, R. Sifa, C. Bauckhage, H. Hajishirzi, and Y. Choi. “Is reinforcement learning (not) for natural language processing: Benchmarks, baselines, and building blocks for natural language policy optimization”. In: *arXiv preprint arXiv:2210.01241* (2022).
- [131] OpenAI. *GPT-4o*. 2024. URL: <https://openai.com/o1/>.
- [132] G. Team, R. Anil, S. Borgeaud, Y. Wu, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth, et al. “Gemini: a family of highly capable multimodal models”. In: *arXiv preprint arXiv:2312.11805* (2023).
- [133] B. Romera-Paredes, M. Barekatin, A. Novikov, M. Balog, M. P. Kumar, E. Dupont, F. J. Ruiz, J. S. Ellenberg, P. Wang, O. Fawzi, et al. “Mathematical discoveries from program search with large language models”. In: *Nature* 625.7995 (2024), pp. 468–475.
- [134] T. X. Olausson, J. P. Inala, C. Wang, J. Gao, and A. Solar-Lezama. “Is Self-Repair a Silver Bullet for Code Generation?” In: *The Twelfth International Conference on Learning Representations*. 2023.
- [135] C. Sun, J. Han, W. Deng, X. Wang, Z. Qin, and S. Gould. “3d-gpt: Procedural 3d modeling with large language models”. In: *arXiv preprint arXiv:2310.12945* (2023).

- [136] I. Huang, G. Yang, and L. Guibas. “BlenderAlchemy: Editing 3D Graphics with Vision-Language Models”. In: *arXiv preprint arXiv:2404.17672* (2024).
- [137] Y. Yang, F.-Y. Sun, L. Weihs, E. VanderBilt, A. Herrasti, W. Han, J. Wu, N. Haber, R. Krishna, L. Liu, et al. “Holodeck: Language guided generation of 3d embodied ai environments”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2024, pp. 16227–16237.
- [138] Y. Yamada, K. Chandu, Y. Lin, J. Hessel, I. Yildirim, and Y. Choi. “L3GO: Language Agents with Chain-of-3D-Thoughts for Generating Unconventional Objects”. In: *arXiv preprint arXiv:2402.09052* (2024).
- [139] A. Tewari, M. Zollhofer, H. Kim, P. Garrido, F. Bernard, P. Perez, and C. Theobalt. “Mofa: Model-based deep convolutional face autoencoder for unsupervised monocular reconstruction”. In: *Proceedings of the IEEE international conference on computer vision workshops*. 2017, pp. 1274–1283.
- [140] E. Riba, D. Mishkin, D. Ponsa, E. Rublee, and G. Bradski. “Kornia: an open source differentiable computer vision library for pytorch”. In: *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*. 2020, pp. 3674–3683.
- [141] H. Kato, Y. Ushiku, and T. Harada. “Neural 3d mesh renderer”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 3907–3916.
- [142] T.-M. Li, M. Aittala, F. Durand, and J. Lehtinen. “Differentiable monte carlo ray tracing through edge sampling”. In: *ACM Transactions on Graphics (TOG)* 37.6 (2018), pp. 1–11.
- [143] M. Nimier-David, D. Vicini, T. Zeltner, and W. Jakob. “Mitsuba 2: A retargetable forward and inverse renderer”. In: *ACM Transactions on Graphics (ToG)* 38.6 (2019), pp. 1–17.
- [144] M. M. Loper and M. J. Black. “OpenDR: An approximate differentiable renderer”. In: *Computer Vision—ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part VII* 13. Springer. 2014, pp. 154–169.
- [145] M. Fey and J. E. Lenssen. “Fast graph representation learning with PyTorch Geometric”. In: *arXiv preprint arXiv:1903.02428* (2019).
- [146] Y. Hu, L. Anderson, T.-M. Li, Q. Sun, N. Carr, J. Ragan-Kelley, and F. Durand. “DiffTaichi: Differentiable programming for physical simulation”. In: *arXiv preprint arXiv:1910.00935* (2019).
- [147] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimeshein, L. Antiga, et al. “Pytorch: An imperative style, high-performance deep learning library”. In: *Advances in neural information processing systems* 32 (2019).
- [148] L. A. Gatys. “A neural algorithm of artistic style”. In: *arXiv preprint arXiv:1508.06576* (2015).
- [149] L. A. Gatys, A. S. Ecker, and M. Bethge. “Image style transfer using convolutional neural networks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 2414–2423.
- [150] K. Simonyan and A. Zisserman. “Very deep convolutional networks for large-scale image recognition”. In: *arXiv preprint arXiv:1409.1556* (2014).



- [151] A. Krizhevsky, I. Sutskever, and G. E. Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems* 25 (2012).
- [152] S. Deguy and A. Benassi. “A Flexible Noise Model For Designing Maps.” In: *VMV*. Citeseer. 2001, pp. 299–308.
- [153] Y. Hu, T.-M. Li, L. Anderson, J. Ragan-Kelley, and F. Durand. “Taichi: a language for high-performance computation on spatially sparse data structures”. In: *ACM Transactions on Graphics (TOG)* 38.6 (2019), pp. 1–16.
- [154] L. Gatys, A. S. Ecker, and M. Bethge. “Texture synthesis using convolutional neural networks”. In: *Advances in neural information processing systems* 28 (2015).
- [155] P. Belotti, C. Kirches, S. Leyffer, J. Linderoth, J. Luedtke, and A. Mahajan. “Mixed-integer nonlinear optimization”. In: *Acta Numerica* 22 (2013), pp. 1–131.
- [156] J. Lee and S. Leyffer. *Mixed integer nonlinear programming*. Vol. 154. Springer Science & Business Media, 2011.
- [157] D. P. Kingma. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).
- [158] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. “Proximal policy optimization algorithms”. In: *arXiv preprint arXiv:1707.06347* (2017).
- [159] J. Wu, L. Ouyang, D. M. Ziegler, N. Stiennon, R. Lowe, J. Leike, and P. Christiano. “Recursively summarizing books with human feedback”. In: *arXiv preprint arXiv:2109.10862* (2021).
- [160] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. “High-dimensional continuous control using generalized advantage estimation”. In: *arXiv preprint arXiv:1506.02438* (2015).
- [161] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, et al. “Learning transferable visual models from natural language supervision”. In: *International conference on machine learning*. PMLR. 2021, pp. 8748–8763.
- [162] R. Zhang, P. Isola, A. A. Efros, E. Shechtman, and O. Wang. “The unreasonable effectiveness of deep features as a perceptual metric”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 586–595.
- [163] E. Heitz, K. Vanhoey, T. Chambon, and L. Belcour. “A sliced wasserstein loss for neural texture synthesis”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021, pp. 9412–9420.
- [164] A. Sauer, D. Lorenz, A. Blattmann, and R. Rombach. “Adversarial diffusion distillation”. In: *European Conference on Computer Vision*. Springer. 2025, pp. 87–103.
- [165] T. Yin, M. Gharbi, R. Zhang, E. Shechtman, F. Durand, W. T. Freeman, and T. Park. “One-step Diffusion with Distribution Matching Distillation”. In: *CVPR*. 2024.

- [166] A. Raistrick, L. Lipson, Z. Ma, L. Mei, M. Wang, Y. Zuo, K. Kayan, H. Wen, B. Han, Y. Wang, et al. “Infinite photorealistic worlds using procedural generation”. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2023, pp. 12630–12641.
- [167] B. Li, K. Zhang, H. Zhang, D. Guo, R. Zhang, F. Li, Y. Zhang, Z. Liu, and C. Li. *LLaVA-NeXT: Stronger LLMs Supercharge Multimodal Capabilities in the Wild*. May 2024. URL: <https://llava-vl.github.io/blog/2024-05-10-llava-next-stronger-llms/>.
- [168] H. Liu, C. Li, Q. Wu, and Y. J. Lee. *Visual Instruction Tuning*. 2023.
- [169] H. Liu, C. Li, Y. Li, and Y. J. Lee. *Improved Baselines with Visual Instruction Tuning*. 2023.
- [170] H. Liu, C. Li, Y. Li, B. Li, Y. Zhang, S. Shen, and Y. J. Lee. *LLaVA-NeXT: Improved reasoning, OCR, and world knowledge*. Jan. 2024. URL: <https://llava-vl.github.io/blog/2024-01-30-llava-next/>.
- [171] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan, et al. “The llama 3 herd of models”. In: *arXiv preprint arXiv:2407.21783* (2024).
- [172] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen. “Lora: Low-rank adaptation of large language models”. In: *arXiv preprint arXiv:2106.09685* (2021).
- [173] I. Loshchilov. “Decoupled weight decay regularization”. In: *arXiv preprint arXiv:1711.05101* (2017).
- [174] T. Dao. “Flashattention-2: Faster attention with better parallelism and work partitioning”. In: *arXiv preprint arXiv:2307.08691* (2023).
- [175] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He. “Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters”. In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2020, pp. 3505–3506.
- [176] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He. “Zero: Memory optimizations toward training trillion parameter models”. In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2020, pp. 1–16.
- [177] C. J. Geyer. “Practical markov chain monte carlo”. In: *Statistical science* (1992), pp. 473–483.
- [178] P. Diaconis. “The markov chain monte carlo revolution”. In: *Bulletin of the American Mathematical Society* 46.2 (2009), pp. 179–205.
- [179] E. Schkufza, R. Sharma, and A. Aiken. “Stochastic superoptimization”. In: *ACM SIGARCH Computer Architecture News* 41.1 (2013), pp. 305–316.
- [180] N. Bonneel, J. Rabin, G. Peyré, and H. Pfister. “Sliced and radon wasserstein barycenters of measures”. In: *Journal of Mathematical Imaging and Vision* 51 (2015), pp. 22–45.
- [181] C. Villani et al. *Optimal transport: old and new*. Vol. 338. Springer, 2009.
- [182] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. “Mastering the game of Go with deep neural networks and tree search”. In: *nature* 529.7587 (2016), pp. 484–489.