Supplemental Document: Generating Procedural Materials from Text or Image Prompts

YIWEI HU, Yale University, USA and Adobe Research, USA
PAUL GUERRERO, Adobe Research, UK
MILOŠ HAŠAN, Adobe Research, USA
HOLLY RUSHMEIER, Yale University, USA
VALENTIN DESCHAINTRE, Adobe Research, UK

ACM Reference Format:
Yiwei Hu, Paul Guerrero, Miloš Hašan, Holly Rushmeier, and Valentin Deschaintre. 2023. Supplemental Document: Generating Procedural Materials from Text or Image Prompts. In *Special Interest Group on Computer Graphics and Interactive Techniques Conference Conference Proceedings (SIGGRAPH '23 Conference Proceedings), August 6–10, 2023, Los Angeles, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3588432.3591520

1 OVERVIEW

In this supplementary material, we provide implementation details and show additional results and comparisons. In Sec. 2, we show more conditional generation results and a comparison to the original MatFormer from our unconditional version. In Sec. 3, we provide additional implementation details for the generative models, dataset processing and graph sequence sampling. In Sec. 4, we discuss alternative implementations i.e. another image encoding method and another parameter generator architecture.

2 ADDITIONAL RESULTS.

2.1 Statistical Analysis

We present a detailed statistical analysis of our quantitative comparisons. We define our metric, the style loss, as a combination of L1 difference between the Gram Matrices of VGG19 activations [Simonyan and Zisserman 2015] and 16x16 downsampled thumbnails ($\alpha = 0.1$) of the image prompt I and the node graphs $g(\theta)$ after evaluation:

$$L = ||GM(I) - GM(R(g(\theta)))||_1 + \alpha||I_{16\times16} - R(g(\theta))_{16\times16}||_1 \quad (1)$$

where GM is an operator that computes Gram Matrices of the VGG features extracted from the image, and R is an rendering operator that renders the material maps on a planar surface. In addition to the mean loss among all test samples, we report and plot the 95% confidence intervals as shown in Table 1 and Fig. 1. This confirms that our models 1) show a significant improvement before optimization, and 2) perform on par with baselines after optimization.

SIGGRAPH '23 Conference Proceedings, August 6–10, 2023, Los Angeles, CA, USA

ACM ISBN 979-8-4007-0159-7/23/08.
https://doi.org/10.1145/3588432.3591520

2.2 Multi-Modal Conditioning

Conditional generation. We show a more extensive set of image conditioning, text conditioning and autocompletion results in Figures 3, 4 and 5, respectively. Visualizations of the graphs corresponding to these results can be found the zip file.

Unconditional generation. In Fig. 7, we show that our substantially augmented dataset and regularized sampling process benefits unconditional material graph generation as well. We train an unconditional version of our model and compare it to the unconditional model presented in MatFormer. Our unconditional model generates material graphs with a more diverse appearance. For example, note the larger range of different patterns and structures produced by our generator.

2.3 Comparison to Class-conditioned Generation

We compare our approach to class-conditioned version of our generator, showing that our CLIP-based conditioned generation provides more accurate information than a simple class.

During training, instead of conditioning on CLIP embeddings, we make all the three generators conditional on material categories using a learnable embedding layer which accepts category tokens as inputs and transforms the tokens to the class embeddings. We use the material categories associated to each graph in the Substance Source Dataset [Adobe 2023] (16 different classes). We use the same network architectures, configuration and hyperparameters as for training our CLIP-conditional model, but use the class conditional embedding. During inference, given an input image, we first classify this image into one of the 16 material categories using CLIP before generating material graphs conditioned on the classified categories.

In Fig. 2, we show that our CLIP-based generative model performs significantly better than simple class-conditioned generation for matching the general material appearance (e.g. structures and patterns). We show here the direct output of the network and do not apply post optimization. We further conduct the same experiment as in the main paper and report the statistics in Table 2, showing that our CLIP-conditioned generation generates results closer to the target than class-conditioned generation.

We believe that the existing color mismatch before optimization is caused by the limited training data available, and by the difference between image-space and parameter space error i.e., a relatively small error in parameter space of the color results in a large error in image space. Evaluating image-space loss during training/inference is an interesting and challenging avenue for future work.

Table 1. In addition to the mean loss values reported in the main paper, we report the **[lower bound, upper bound]** of 95% confidence intervals of computed loss values. See Fig. 1 for a visualization.

Unoptimized	Ours	Ours (VGG)	Ours Uncond	Dataset
Best-of-5	0.0314[0.027,0.036]	**0.0300[0.026,0.034]**	0.0382[0.033,0.043]	0.0384[0.033,0.043]
Avg-of-5	0.0346[0.030,0.039]	**0.0329[0.029,0.037]**	0.0539[0.048,0.060]	0.0499[0.044,0.055]
Optimized	Ours	Ours (VGG)	Ours Uncond	Dataset
Best-of-5	0.0218[0.018,0.025]	0.0199[0.016,0.023]	0.0194[0.016,0.023]	**0.0191[0.015,0.023]**
Avg-of-5	0.0242[0.020,0.028]	**0.0221[0.018,0.026]**	0.0227[0.019,0.027]	0.0237[0.019,0.028]

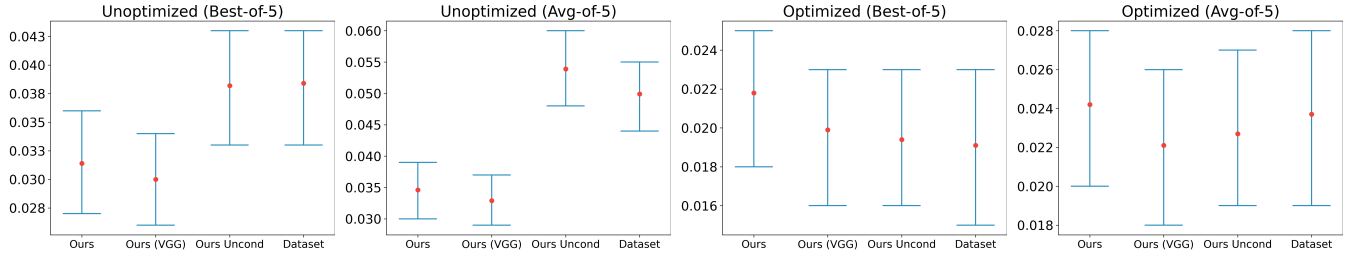

Fig. 1. Plots of 95% confidence intervals of quantitative results. **Lower is better.** Left to Right: Unoptimized (Best-of-5), Unoptimized (Avg-of-5), Optimized (Best-of-5) and Optimized (Avg-of-5). Our method shows a significant improvement over the query baselines before optimization, and performs on par with baselines after optimization as they seem to improve more from the optimization step.

Table 2. We report the style loss measured over our test set of real image data, similar to what is shown the main paper (similar to the unoptimized results shown in Table 1). We report the mean loss values as well as the [lower bound, upper bound] of 95% confidence intervals. Our model shows statistically significant improvement over class-conditioned generation. **Lower is better.**

Unoptimized	Ours	Class-conditioned
Best-of-5	**0.0314[0.027,0.036]**	0.0544[0.048,0.060]
Avg-of-5	**0.0346[0.030,0.039]**	0.0774[0.071,0.084]

2.4 Visual Comparisons to Baselines

We show additional qualitative comparisons to baselines in Fig. 6, demonstrating our model's capability to reproduce material appearance similar to querying a huge pre-generated dataset (102,400 materials for **Ours Uncond** and 466,700 for **Dataset**), while having a memory footprint that is 4-5 orders of magnitude smaller than the memory required for dataset retrieval (15 MB for our model compared to 115 GB or more required for dataset retrieval).

3 IMPLEMENTATION DETAILS

3.1 Generative Model Details

Below we describe the graph linearization, auxiliary tokens, and auxiliary input sequences used by each of our generative models. Unless otherwise noted, these are the same as in MatFormer[Guerrero et al. 2022].

Graph linearization. We use a back-to-front breadth-first traversal of a node graph to obtain the order for our sequences. This traversal starts at the output nodes and moves from child to parent in a breadth-first order. As shown in MatFormer, this results in a more canonical linearization than other order strategies, such as front-to-back ordering. One exception is autocompletion, where we use a front-to-back breadth-first traversal, as the nodes given by the user, which are typically at the start of the graph, should form the start of the node sequence.

Node generation. During node generation, we train the model to output the node depth (the graph distance of a node from the closest output node) in addition to the node type. The depth of previously generated nodes is then used as auxiliary input sequence to give the model more information about the rough position of the node in the graph. In total, the node model uses 3 input sequences:

- the sequence of previously generated node types;
- the sequence of previously generated node depths;
- the global position of a token in the node sequence.

We only use two auxiliary tokens per sequence that denote the start and end of a sequence, unlike MatFormer, which also uses auxiliary tokens to denote the end of a set of child nodes in the breadth-first order. We did not use these additional tokens used by MatFormer as we found that they did not increase our performance.

Edge generation. The edge model generates tuples of pointers into a list of input or output slots. It is implemented as a Pointer Network [Vinyals et al. 2015] that consists of a transformer encoder and a transformer decoder. The transformer encoder outputs a list of slot embeddings, while the transformer decoder outputs a query feature vector in each step. The affinity of the query feature vector with each slot embedding, computed as a dot product, determines the probability that an edge starts or ends in this slot. The transformer encoder uses 5 input sequences to describe a slot:

- the operator type of the slot's node;
- the index of the node;

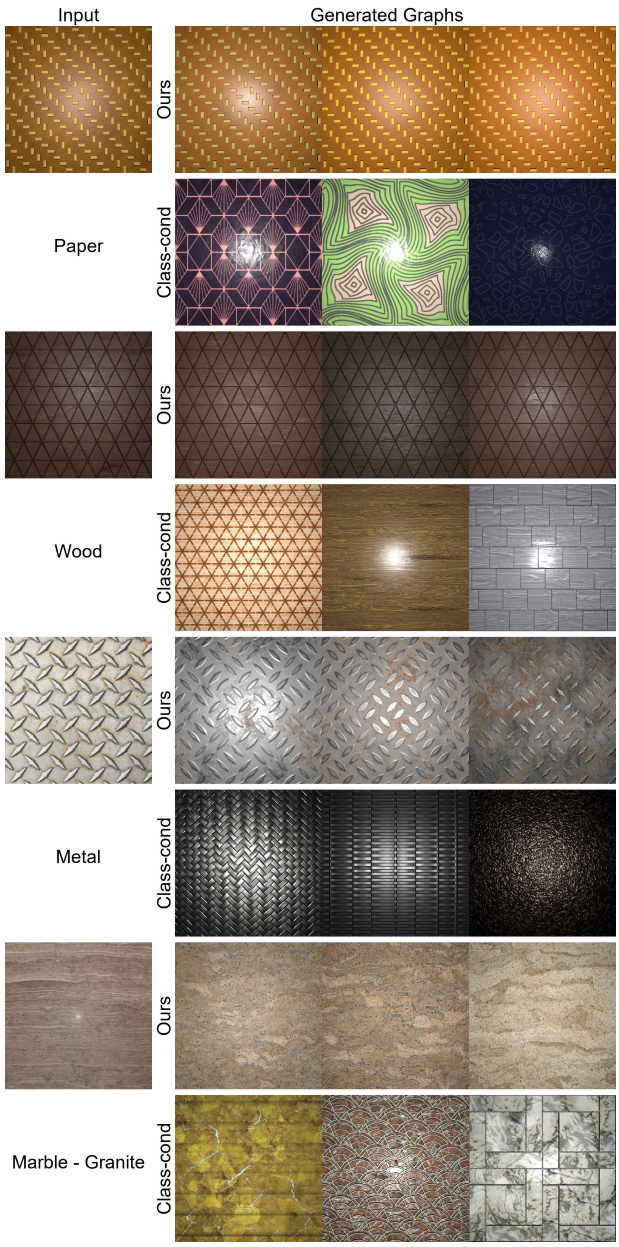

Fig. 2. Comparison to class-conditioned generation. We show predicted material graphs without post optimization. The first two examples are synthetic materials while the last two are real photographs. For the class-conditioned generation, we generate graphs conditioned on the classified material categories (below each image prompt) using CLIP. Though color differences exist, our conditional model matches the input images much better compared to simple class-conditioned generation.

- the node depth;
- the index of the slot inside the node;
- the global position of a token inside the slot sequence.

The transformer decoder uses 3 input sequences to describe edge start or end slots:

- the slot's embedding;
- the tuple index of a slot in the edge (i.e. 1 if the edge starts at the slot, 2 if it ends at the slot);
- the global position of a token inside the edge sequence.

We use two auxiliary tokens per sequence that denote the start and end of a sequence.

Parameter generation. The transformer encoder g^P that computes the node embeddings receives two input sequences, the node type and the node depth, as output by the node generator. The parameter model uses 7 input sequences:

- the sequence of previously generated parameter values;
- the sequence of node embeddings for the node each token is generated for;
- the index of the parameter in the node, not counting skipped parameters (parameters that are at their default values are skipped during generation, to shorten parameter sequences);
- the index of the parameter in the node, counting skipped parameters;
- the index of the array element for parameters that contain arrays of values, or 0 if the parameter is not an array;
- the index of the vector element for parameters that are vectors, or 0 if the parameter is not vector-valued;
- the global position of a token inside the parameter sequence.

We use auxiliary tokens that denote the start and end of the parameter sequence. Unlike MatFormer, our parameter generator generates all graph parameters as a single list. For this reason, we also use auxiliary tokens that mark the start of the per-node parameter subsequences.

3.2 Dataset Processing Details.

Reformatting. The raw dataset contains material graphs created over several years, using different versions of the node graph systems. We first ensure that all material graphs are properly formatted. Here is a checklist for dataset reformatting mentioned in our paper:

- Upgrade material graph to the latest version to resolve compatibility problems. The same type of node can have slightly different implementations and behaviors in different versions of node graph systems.
- Fix numerical precision problems with the bit depth in input or output images of some nodes in the graph. Potential differences in the bit depth of input/output 2D images will affect the precision of some nodes.
- Fix missing dependencies in some graphs. Each graph may use other existing graphs as operators, in which case a dependency to the existing graph is created. Some graphs had incorrect dependencies that needed to be fixed.
- Enable alpha channels for all input and output images of nodes in each graph. We found that some graphs require alpha channels to give a correct output, and for the other graphs using an alpha channel does not degrade the output quality.

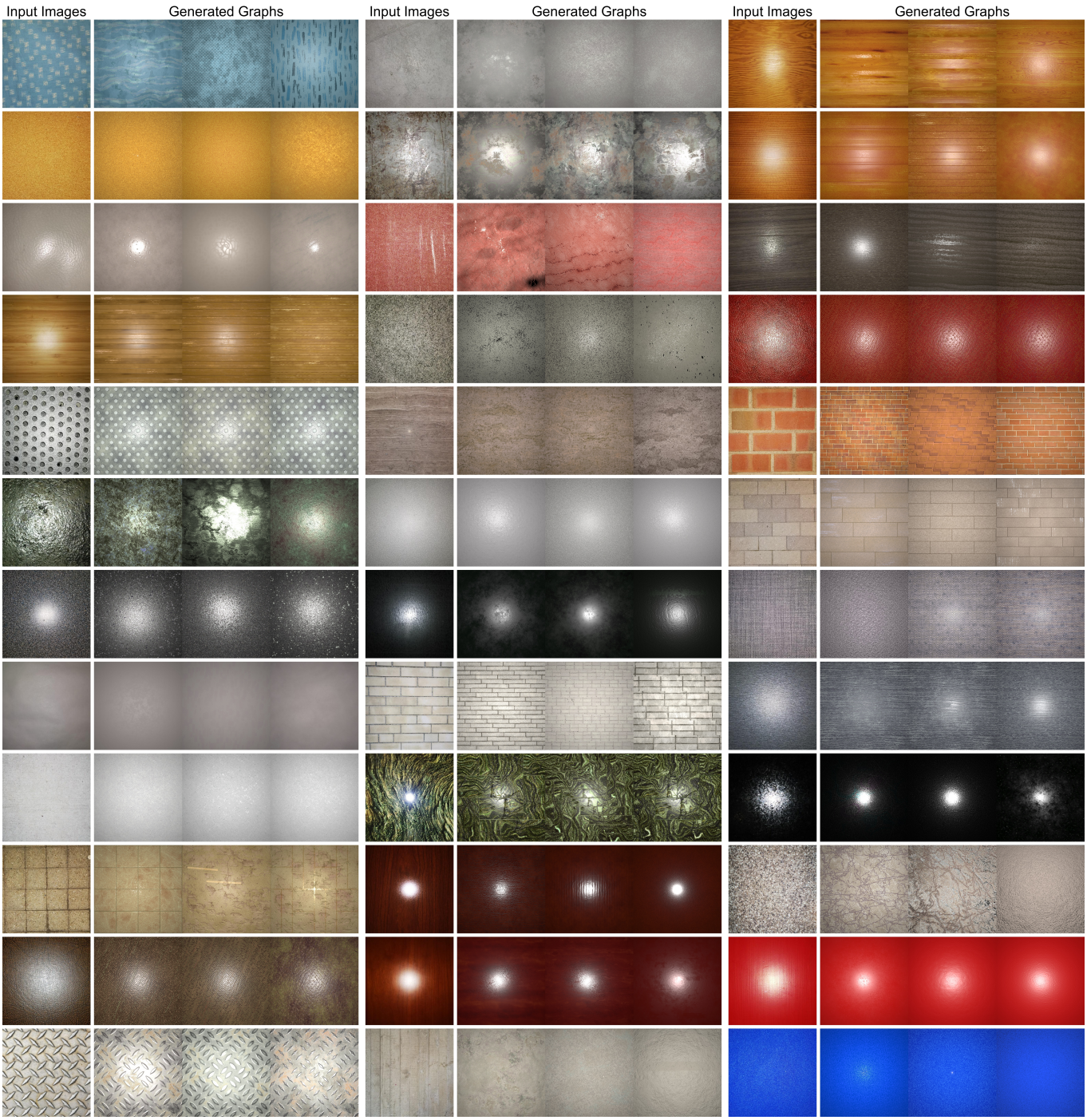

Fig. 3. Conditioning on Real Images. We show real photos as inputs and the output of three of our generated material graphs.

3.3 Sampling Details

Semantic Validation. While MatFormer ensures semantic validity as a post-processing step, once the entire sequence has been generated, we perform the validity checks during sampling, making sure to choose only among semantically valid choices for any given token. This makes sure that the chosen tokens are semantically consistent with the previously generated tokens. Here, we list our rules for validity checks during sampling:

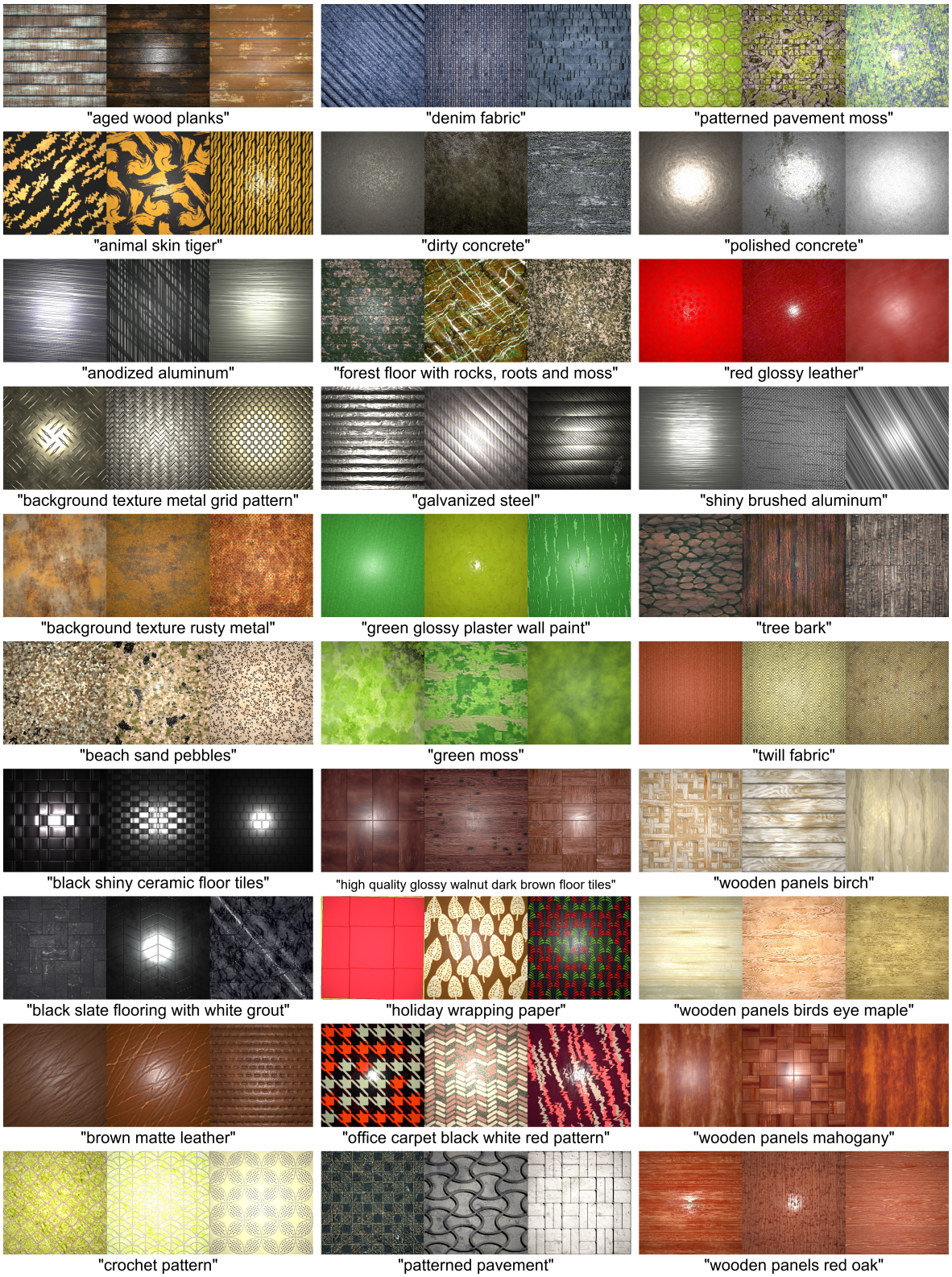

Fig. 4. Text Conditioning. Our model generates multiple procedural material graphs given various text prompts.

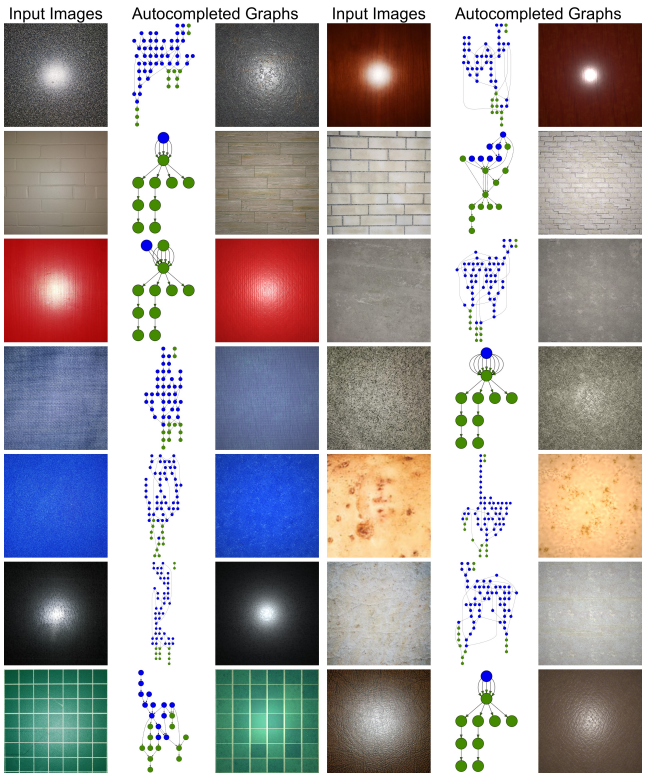

Fig. 5. Autocompletion. As a sequential model, our models can accept partial sequences e.g., partially completed material graphs and generate the rest of the structures and parameters toward given image prompts. Existing nodes and edges are blue and our predicted nodes and are green.

- The generated node depth has to be monotonically increasing/decreasing over a node sequence for back-to-front/front-to-back node ordering.
- Each edge can only connect an output slot of a source node to an input slot of a different target node. Additionally, edges are not allowed to form cycles.
- Parameter values may only be sampled within the valid range for a given parameter. Each parameter has a known minimum and maximum. Additionally, a new parameter may not be started if the current parameter is vector-valued and is still missing some of its elements. This ensures that only complete parameters are generated.

4 ALTERNATIVE IMPLEMENTATIONS

In this section, we introduce alternatives implementation of our model mentioned in our main paper. We describe the details and discuss their performance.

4.1 Alternative conditioning

Conditioning on CLIP embedding allows our model to accept both image and text prompts as inputs, while other encoding methods are possible. We experiment with an alternative image encoding approach. While CLIP captures the high-level semantic information

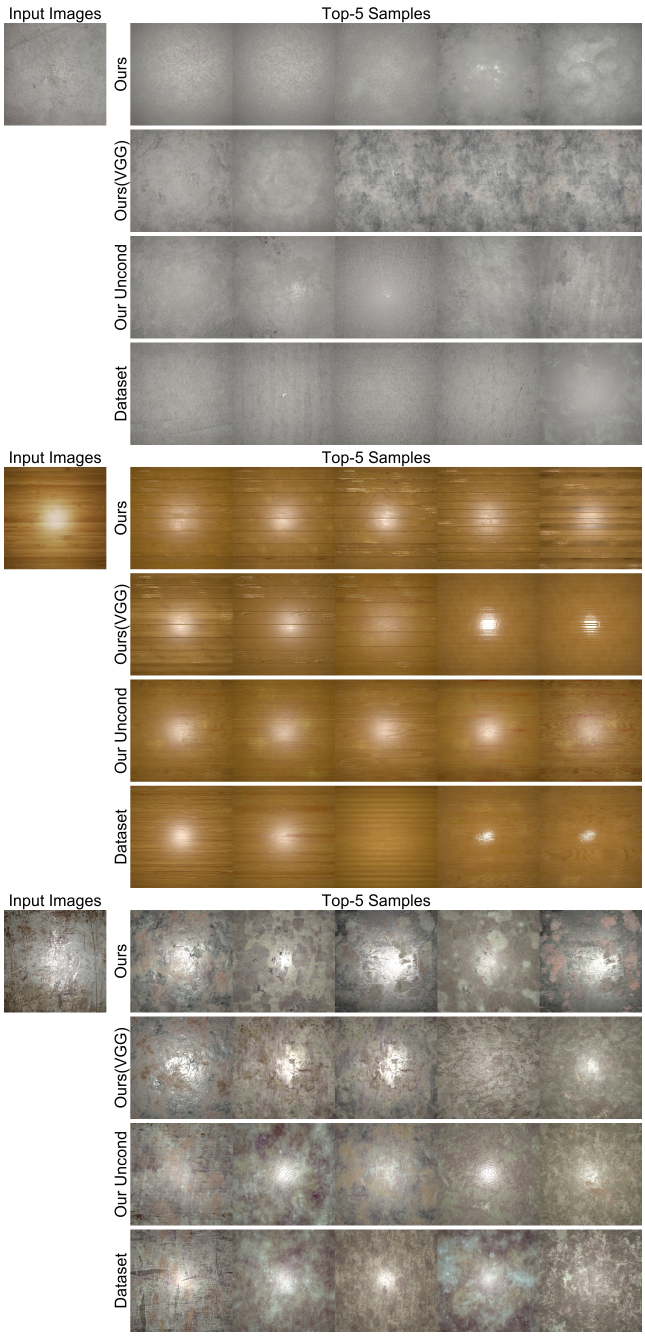

Fig. 6. Here, we show more visual comparisons to our baselines. Our model generates material graphs visual similar to a query in a giant database.

in an image, we would like to encode low-level texture statistics to see if it can better capture lower-level details. We therefore add VGG feature statistics to capture fine-scale texture detail and a 16x16 downsampled thumbnail to summarize the main color of the input image. The VGG feature statistics we compute are based on the layer-wise mean and standard derivation from extracted deep layer

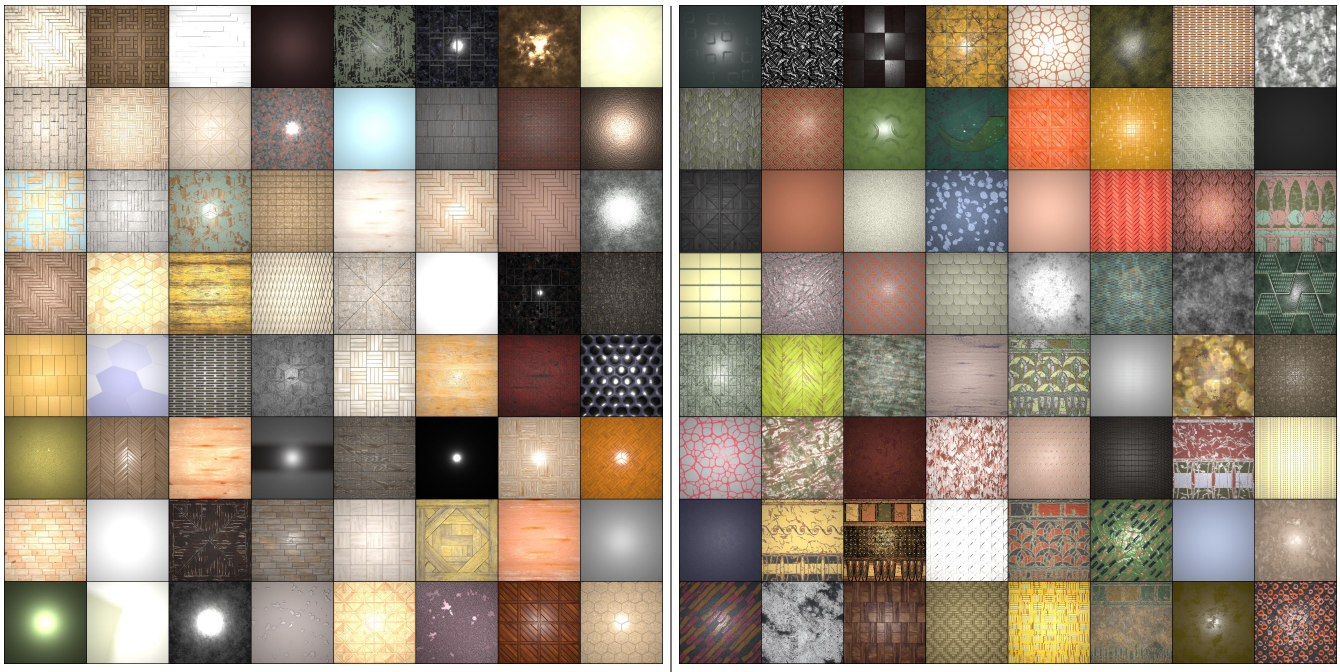

Fig. 7. We show that our augmented data processing and sampling pipeline helps improve unconditional graph generation. On the left are sampled material graphs by MatFormer [Guerrero et al. 2022], which biases toward certain patterns and the wood category. On the right we show our results with a wider variety of semantically-meaningful and more realistic appearances. In both cases, we unconditionally randomly sample 64 graphs.

features (ReLu2_1, ReLu3_1, ReLu4_1) of a pretrained VGG model. We flatten and concatenate all the features and map the feature vector to the dimension of hidden states by a trainable MLP. In the main paper, we denote this variation as **Ours(VGG)** while our used model as **Ours**. This alternative encoding produces slightly lower numerical error as presented in the main paper, but its construction is however slightly more complicated and prevents the use of text encoding and we do not observe visible qualitative differences with the CLIP-only encoding.

4.2 Alternative Parameter Generator

Our parameter generator is a graph-conditioned parameter generator using GCNs. Another choice of parameter generator is to extend MatFormer's parameter generator to accept conditional text or image embedding. In this variant, the parameter generator generates parameters for each node individually without GCN. To examine the performance of our parameter generator with GCN, we create a synthetic dataset containing 5120 graph/parameters data points. We evaluate the model's accuracy by cross entropy error on the predicted tokens conditioning on ground-truth node and edge sequences. Our architecture gains marginal improvement with an average error of 0.729 compared to 0.740 on predicted parameters. Combined with a more parallelizable training (as we process per graph rather than per node) leading to a 1.5x speedup in the training time, we choose to use our graph-conditioned parameter generator as part of our conditional model.

REFERENCES

Adobe. 2023. Substance Designer. https://www.substance3d.com/.
Paul Guerrero, Milos Hasan, Kalyan Sunkavalli, Radomir Mech, Tamy Boubekeur, and Niloy Mitra. 2022. MatFormer: A Generative Model for Procedural Materials. *ACM Trans. Graph.* 41, 4, Article 46 (2022). https://doi.org/10.1145/3528223.3530173
Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1409.1556
Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. *Advances in neural information processing systems* 28 (2015).