MatFormer: A Generative Model for Procedural Materials

PAUL GUERRERO and MILOŠ HAŠAN, Adobe Research
KALYAN SUNKAVALLI, RADOMÍR MĚCH, and TAMY BOUBEKEUR, Adobe Research
NILOY J. MITRA, Adobe Research and University College London

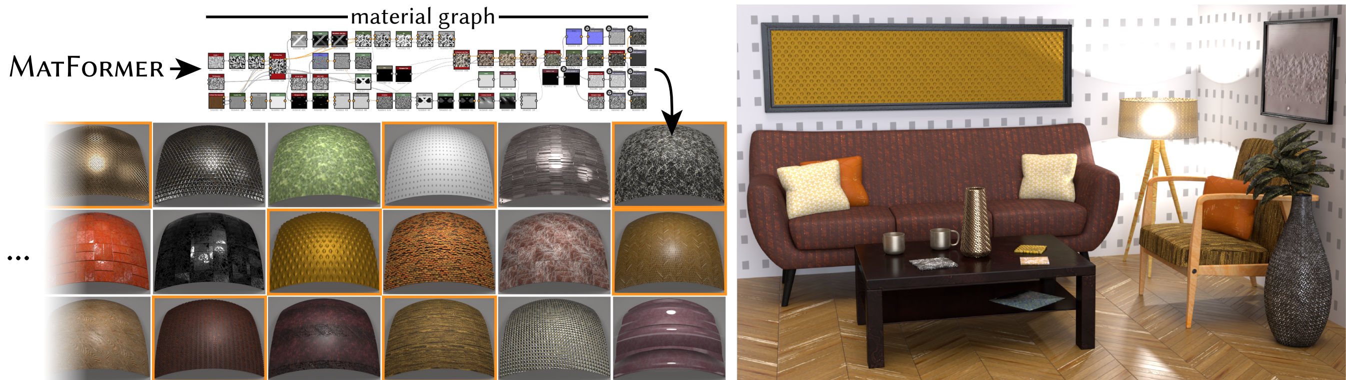

Fig. 1. We present MatFormer, a generative model for procedural materials that are represented as node graphs. MatFormer generates an arbitrary number of editable, resolution-independent materials (left), that can be used in realistic scenes (right). All materials in the scene were generated by our method; some of them are shown highlighted in orange on the left. Materials are generated as node graphs (top), where nodes correspond to image operators (each node shows the operator output), and edges control the flow of information between nodes. User can adjust parameters directly using the parameterized material graphs.

Procedural material graphs are a compact, parameteric, and resolution-independent representation that are a popular choice for material authoring. However, designing procedural materials requires significant expertise and publicly accessible libraries contain only a few thousand such graphs. We present MatFormer, a generative model that can produce a diverse set of high-quality procedural materials with complex spatial patterns and appearance. While procedural materials can be modeled as directed (operation) graphs, they contain arbitrary numbers of heterogeneous nodes with unstructured, often long-range node connections, and functional constraints on node parameters and connections. MatFormer addresses these challenges with a multi-stage transformer-based model that sequentially generates nodes, node parameters, and edges, while ensuring the semantic validity of the graph. In addition to generation, MatFormer can be used for the auto-completion and exploration of partial material graphs. We qualitatively and quantitatively demonstrate that our method outperforms alternative approaches, in both generated graph and material quality.

CCS Concepts: • **Computing methodologies → Image manipulation**; *Machine learning*.

Additional Key Words and Phrases: node graphs, procedural materials, transformers, generative models

ACM Reference Format:
Paul Guerrero, Miloš Hašan, Kalyan Sunkavalli, Radomír Měch, Tamy Boubekeur, and Niloy J. Mitra. 2022. MatFormer: A Generative Model for Procedural Materials. *ACM Trans. Graph.* 41, 4, Article 46 (July 2022), 12 pages. https://doi.org/10.1145/3528223.3530173

Authors' addresses: Paul Guerrero, guerrero@adobe.com; Miloš Hašan, mihasan@adobe.com, Adobe Research; Kalyan Sunkavalli, sunkaval@adobe.com; Radomír Měch, rmech@adobe.com; Tamy Boubekeur, boubek@adobe.com, Adobe Research; Niloy J. Mitra, n.mitra@cs.ucl.ac.uk, Adobe Research and University College London.

© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *ACM Transactions on Graphics*, https://doi.org/10.1145/3528223.3530173.

1 INTRODUCTION

High-quality materials are an essential ingredient for creating virtual assets for a range of computer graphics applications including games, movies, and AR/VR. Procedural *node graphs* are particularly popular as a controllable, resolution-independent material representation. By combining geometry with such procedural textures, artists regularly create realistic and compelling virtual assets.

Procedural material design (e.g., using a product such as Adobe Substance 3D Designer) typically involves creating a directed node graph, referred to as a *material graph*. Such graphs consist of a set of nodes—representing noise and pattern generators or operations on textures (e.g., filter kernels, transformations)—and edges—representing the flow of information from the output of the nodes to inputs of the subsequent nodes—finally producing image maps (e.g., roughness, normal, diffuse) for an analytic SVBRDF model. The output maps can be controlled by editing the parameters of the individual nodes. With complex material definitions regularly needing 50+ nodes, authoring such graphs is a form of black magic, limited to a select handful of practitioners. Not surprisingly, the largest publicly-available texture dataset [Adobe 2021a] has only a few thousand such definitions, and non-expert users mainly select from these limited options. Hence, there is a demand for automatically generating procedural materials, or assisting with their creation.

In the recent years, deep learning has pushed the state-of-the-art in generative models for images [Karras et al. 2020], animations [Houthooft et al. 2016], videos [Kumar et al. 2020], geometry [Mo et al. 2019]; and even the direct generation of material maps

has been explored [Guo et al. 2020b]. However, procedural materials, because of their controllability, are a desirable representation, and previous methods cannot be directly applied to generate them because of multiple challenges. First, unlike images/videos, material graphs do not have a regular structure and can have an arbitrary number of nodes with varying depth and breadth. Second, material graphs are typically composed of heterogeneous nodes that have different numbers and types of parameters, and different numbers of edge slots. Third, each node's input and output slots have different functional specifications, based on its (node) type, that need to be accounted for. Lastly, material graphs contain long chains of operations, with long-range connections between distant nodes that are often critical to the appearance of the material.

In this work, we introduce MatFormer, the *first autoregressive generative model for material graphs* that addresses the challenges detailed above. MatFormer leverages a transformer-based [Vaswani et al. 2017] architecture to model a probability distribution over the space of procedural materials, and subsequently allows sample from this distribution. We found the choice of transformers, as opposed to LSTM, GRU, or graph networks, to be particularly suitable in this context, as transformers effectively handle sparse long-distance connections between graph nodes. However, in order to model the specific structure of material graphs, MatFormer does not generate them in a single pass. Instead, it runs in three sequential stages, each modeled with a dedicated transformer to capture dependencies: first, we generate a sequence of nodes; second, we generate parameters for each of the generated nodes; and finally, we generate directed edges connecting the input and output slots of the generated nodes.

MatFormer is trained on a dataset of 2816 procedural graphs [Adobe 2021b] and can generate a diverse set of high-quality material graphs. In Figure 1 we show example procedural materials *automatically* generated from MatFormer; please refer to the supplementary for a thousand such material graph generations. These materials exhibit a wide range of spatial patterns, material appearance, and geometric detail. Moreover, each procedural material can be manipulated, using the node parameters, to generate many more variations. For example, in Figure 1 (right) a user chose 21 materials from a palette of 200 auto-generated graphs and combined them with 3D models to create photorealistic renderings of a complex scene. We demonstrate clearly, via various quantitative metrics, that MatFormer outperforms alternate approaches in terms of the quality of the graphs it generates, as well as the plausibility and diversity of the materials that can then be sampled from these graphs.

We also enable a novel auto-complete application where a user starts by picking a set of noise generators, and optionally a few associated starting nodes with connections, thus providing a partial material graph. Based on this specification, we can construct multiple completed graphs, that the user can iteratively explore and refine (please see Fig. 8). MatFormer thus enables a fundamentally different form of material authoring that is considerably simpler and more intuitive that current procedural material design tools.

2 RELATED WORK

Procedural materials. Procedural modeling, given its relevance in synthesizing assets including patterns, shapes, buildings, landscapes, animations, materials, has a long history in computer graphics. Here, we focus on procedural methods only in the context of textures. These methods define functions that map spatial locations to pixel values [Galerne et al. 2012; Peachey 1985; Perlin 1985; Worley 1996]. Other methods simulate complex physical processes like reaction-diffusion to generate textures [Witkin and Kass 1991]. All these methods have parameters that give users control over texture generation. Modern material authoring tools, like Adobe Substance Designer, expand on these ideas: they allow users to combine filter nodes, which represent simple image processing operations, to build graphs that process noise and patterns to generate complex, spatially-varying materials. However, designing such graphs requires significant time and expertise.

To address this, researchers have proposed *inverse* procedural material design methods that fit the parameters of a procedural function to an exemplar texture image. Lefebrve and Poulin [2000] use heuristics based on image features to estimate the parameters of tile and wood procedural models. Motivated by texture synthesis approaches [Gatys et al. 2015b], researchers have also introduced neural version of material generators [Henzler et al. 2019] to directly output texture/material images. More recently, Guo et al. [2020a] use MCMC sampling to optimize for the parameters of procedural material models to fit a target photograph; however, their procedural are simple and hand-coded (PyTorch) programs. Hu et al. [2019] propose training deep neural networks to predict the parameters of procedural node graphs given a captured image, and Hu et al. [2022] propose to reconstruct SVBRDF maps with a procedural representation that consists of procedural region masks combined with procedural noise functions inside each region. However, this is not a generative model. The recently proposed MATch framework [Shi et al. 2020] converts procedural node graphs into differentiable programs and uses stochastic gradient descent to fit the graph parameters to captured images.

Methods like MATch assume that a procedural graph (or function) is given. In contrast, we present a *generative* model that can *create* new procedural material graphs from scratch. Because of the expertise required to manually create them, even the largest procedural material libraries [Adobe 2021a,b] have 1500-5000 assets. Instead, MatFormer allows users to create plausible, custom graphs on the fly, dramatically expanding the space of material appearance that can be generated and enabling a new set of applications in material design, such as free generation and partial graph auto-complete.

Generative models in graphics. The recent years have seen an impressive advancement in the ability of generative models [Goodfellow et al. 2014; Karras et al. 2019] to model 2D images. Furthermore, these generative models have been used in inverse pipelines, projecting images into their latent spaces [Abdal et al. 2019; Richardson et al. 2021; Zhu et al. 2020]. Similarly, methods have been proposed for the generation of 3D shapes [Mo et al. 2019; Nash et al. 2020]. In the context of materials, several methods have been proposed [Guo et al. 2020b; Li et al. 2019; Zhou and Kalantari 2021] to generate realistic per-pixel material maps that can be used for material capture by projecting a set of flash photographs of a material sample into the latent space. However, the results are image maps, limited in resolution, and not parameterized. In contrast, we develop on a

generative model for directly outputting procedural material graphs, which can then be adjusted for further variations by adjusting its parameters.

Procedural node graphs can be represented as sequences of nodes with edges between them. This allows us to use auto-regressive models to generate them. In particular, we use the Transformer model [Vaswani et al. 2017], which has an inbuilt self-attention mechanism, that has been applied to natural language processing applications. Such auto-regressive models have also been used to generate images [Oord et al. 2016], sketches [Ribeiro et al. 2020], geometry [Nash et al. 2020], and layout [Yang et al. 2021]. However, the structure of material graphs, with their arbitrary number of nodes, varying number of input and output edges, and functional constraints on these edges, makes material graph generation significantly more challenging than generating text, images, or meshes. As described later, we propose a three-pass generation algorithm that addresses these challenges.

Program and/or Graph Generation in Graphics. In the context of generative models for irregularly structured domains, [Li et al. 2017; Mo et al. 2019] proposed recursive neural networks for shape synthesis by modeling intra-shape dependencies in the form of adjacency, symmetry, and hierarchical relations, expressed as binary or, more generally, n-ary graphs. Subsequently, these generators were extended to directly produce shape programs [Jones et al. 2020], in hand-coded domain specific languages, using a GRU-based recurrent network. Beyond recurrent networks, PolyGen [Nash et al. 2020] utilized transformers to capture long-range relations to directly output shapes in the form of triangle meshes. Du et al. [2018] were one of the first to explore CGS program construction, by mapping the CGGtrees to a purely discrete representation, and invoking state-of-the-art program synthesizers. With access to large-scale CAD instructions (e.g., Fusion360 [Willis et al. 2021]), researchers have treated shapes as texts, and investigated usage of natural language generators to propose DeepCAD [Wu et al. 2021] for directly producing CAD models. Continuing this line of work we propose an auto-regressive generator that is specifically designed to tackle the unique challenges associated with handling material graphs.

3 METHOD

MatFormer is a generative model for procedural materials; these materials are represented as directed node graphs that output a set of scalar- or vector image channels describing surface properties like albedo, roughness, metallicity, surface normals and height.

A node graph is a parametric model that generates an output given the parameters of each node. Nodes are instances of operations that define named *input slots*, named *output slots*, and a set of heterogeneous *parameters*. Directed edges from output slots to input slots describe the flow of information in the node graph: an output of a given node is used as input by another node. Figure 2 shows an example of a node graph. A node graph is a representation of a functional program, where functions with multiple input and output arguments are represented by nodes. The output arguments correspond to output slots of a node, while the input arguments are split into (a) parameters, which are constants, and (b) input slots, which are arguments that are outputs of other (preceding) functions.

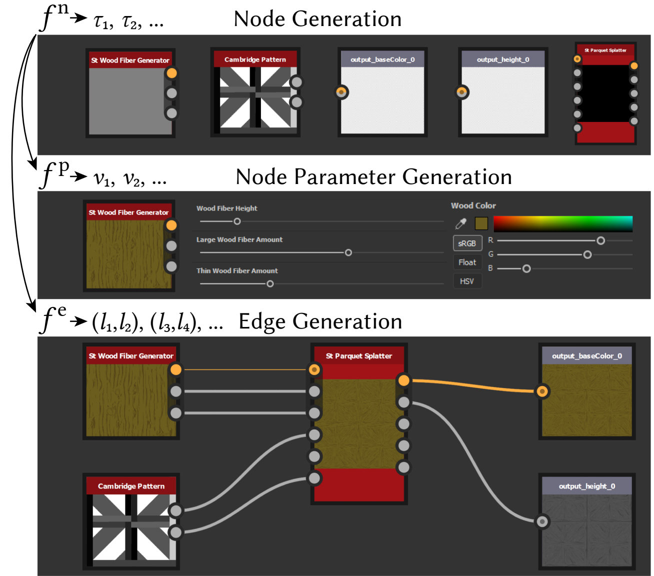

Fig. 2. **Material graph representation.** We generate a material graph in three steps, each step uses a Transformer-based generator f. First, we generate a set of operator types τ that describe the types of nodes in the graph. In the second step, we generate a sequence of node parameter values v for each node, conditioned on the generated node types. Finally, we connect node pairs with edges based on a sequence of node index pairs (l_i, l_{i+1}) to construct the complete procedural material. Here, nodes are illustrated as boxes that display one of the node output images. The small circles to the left and right of any node denote input and output slots, respectively.

There are no loops in these programs. More details on node graphs are given in Section 3.1.

Generating node graphs for realistic materials poses a set of unique challenges, compared to generative models in more traditional pixel-based domains. First, material graphs do not have a simple regular representation like an image grid and their node parameters are heterogeneous, i.e., each node type has a different set of parameters that may include scalars, vectors, and variable-length arrays. Second, material graphs form long chains of operations that cause strong long-distance dependencies between nodes, making traditional approaches to sequence and graph generation ill-suited for the task, because information flow degrades with distance (e.g., Graph Neural Networks [Scarselli et al. 2009] and traditional Recurrent Neural Networks like GRUs [Cho et al. 2014]). Third, material graphs used in practice typically require a large number of nodes and node parameters. Finally, material graphs have multiple functionally and semantically different input and output slots. Connecting the wrong slots may make the resulting graph not only sub-optimal, but invalid.

The above challenges motivate our design choices. Due to the long-distance dependencies between nodes, we choose Transformers [Vaswani et al. 2017] as our backbone network architecture, where information is available in equal fidelity from all parts of the graph, and an *attention mechanism* is used to shape the information flow between different graph regions. Transformers originally come

from natural language processing and operate on sequences, requiring a sequential ordering of the graph nodes. Some previous work has tackled graph generation with Transformers in the context of 2D or 3D shapes [Nash et al. 2020; Para et al. 2021], by generating all nodes and their parameters as a single sequence. This approach would be prohibitively expensive in our case, due to the memory and computation cost of the attention mechanism that is quadratic in the length of the sequence. Instead, we create node graphs in three sequence generation steps, corresponding to nodes, per-node parameters, and directed edges. Each step is implemented by a specialized Transformer, and the last two steps are conditioned on the output of the first step. Figure 2 illustrates this process,

First, we generate nodes as a sequence of node types. However, multiple different node ordering strategies are possible, having different performance and applications (Section 3.3). Second, we generate the heterogeneous parameters of each node as separate sequences, conditioned on the node types that were generated in the previous step (Section 3.4). Finally, we focus on edges. However, edges do not connect nodes directly, but rather connect input slots to output slots. A node can only have as many incoming edges as the number of input slots, given the node type; two nodes can be connected by multiple edges, as long as they terminate in different input slots. Inspired by recent work [Nash et al. 2020; Para et al. 2021], we generate edges using Transformers with Pointer Networks [Vinyals et al. 2015], which allow us to generate pointers into a predefined set of slots rather than nodes (Section 3.5).

3.1 Node Graphs

A node graph $g = (N, E)$ is a directed acyclic multigraph of image operators that consists of a set of nodes $N = \{n_1, n_2, \dots\}$ and edges $E = \{e_1, e_2, \dots\}$. Given a set of node parameters, it outputs a set of material channels such as *albedo* and *roughness*. Figure 2 shows an example of a node graph.

Nodes $n = (\tau, P)$ are instances of image operators and are defined by an operator type τ, and a set of parameters values P. The operator type τ is an index into a library of image operators $O = (o_1, o_2, \dots)$ that each take in a set of parameters and a set of images and output a set of images. The k-th operator is thus a function mapping input images into output images:

$$(I_1^{\texttt{out}}, I_2^{\texttt{out}}, \dots) = o_k(I_1^{\texttt{in}}, I_2^{\texttt{in}}, \dots | P), \text{ with } P = (p_1, p_2, \dots), \quad (1)$$

where each I is a grayscale or RGB image and p_j is a parameter that may be a scalar, vector, or variable-length array of vectors. The number of input images, output images, and the number and type of parameters are defined by the operator.

A node n_i of type τ_i has *input slots* $(\texttt{in}_1^i, \texttt{in}_2^i, \dots)$ and *output slots*, $(\texttt{out}_1^i, \texttt{out}_2^i, \dots)$, which are ports that edges can attach to, to provide input images and receive output images, respectively. We call an operator that does not define any input images a *generator*.

Directed edges $e = (\texttt{out}_k^i, \texttt{in}_l^j)$ from output slots to input slots describe the flow of information in the node graph, i.e., the output image $I_k^{\texttt{out}}$ of node n_i is used as input $I_l^{\texttt{in}}$ of node n_j. Since an input slot can only accept a single input image, each input slot can only have one or zero incoming edges, while output slots can have an arbitrary number of outgoing edges.

Node graphs, which only define a partial ordering among the nodes, can be evaluated in any topological order. Note that cycles would cause infinite looping and are therefore not allowed. Further, not all input slots of a node need to be connected to edges. If no edge is attached, a default value is used for the corresponding operator input, typically an image of all zeros. The node outputs that are used for the final material channels are marked by connecting special *output nodes* to their output slots. These output nodes do not perform any operation, they just annotate the graph output.

Next, we describe how to represent this graph with multiple token sequences, created with Transformer-based sequence generators.

3.2 Transformers

We generate a node graph in three steps for nodes, node parameters, and edges, respectively. Transformers are used as generators in each step. Transformers [Vaswani et al. 2017] are sequence generators that were originally used for natural language processing. Nodes, node parameters, and edges are each generated as a sequence of discrete tokens $S = (s_1, \dots, s_m)$, one token s_i at a time. More details on converting these graph components to sequence representations are given in the next sections. Unlike earlier sequence generators like GRUs [Cho et al. 2014] and LSTMs [Hochreiter and Schmidhuber 1997], Transformers use an attention mechanism that allows them to more accurately capture long-distance relations in a sequence.

Sequence generator. A *Transformer-based generator* f_θ models the probability distribution over sequences S as a product of conditional probabilities over individual tokens:

$$p(S|\theta) := \prod_i p(s_i|s_{<i}, \theta), \quad (2)$$

where $s_{<i} := s_1, \dots, s_{i-1}$ denotes the partial sequence up to the token s_{i-1}. The model f_θ estimates a probability distribution over the possible value assignments for token s_i, conditioned on the partial sequence: $p(s_i|s_{<i}, \theta) = f_\theta(s_{<i})$. Once trained, we can directly sample the predicted distribution to obtain the token s_i. A complete sequence can then be generated, in a typical autoregressive setup, by repeatedly evaluating the model, starting from a special *starting token* α and growing the partial sequence by one token in each step, until a special *stopping token* ω is generated, or a maximum sequence length is reached. We will apply some generators to multiple parallel sequences, denoted as $p_i^a, p_i^b = f_\theta(s_{<i}^a, s_{<i}^b)$, where p_i^* is short for $p(s_i^*|s_{<i}^*, \theta^*)$.

Semantic validity. Some value choices for a token may be semantically invalid. For example, a token that describes the end point of an edge should not refer to input slots that are already occupied by another edge. At inference time, we manually set generated probabilities $p(s_i|s_{<i}, \theta)$ for invalid choices to zero, and re-normalize the remaining probabilities. Validity checks for each step are described in Appendix A.

Conditioning and sequence encoders. We condition the generation of node parameter sequences and edge sequences on the generated node sequence. Conditioning on a sequence requires a sequence encoder. We use *Transformer-based encoders* g_ϕ, which have an architecture that is nearly the same as the generator described above,

but each step takes as input the whole sequence and a token index and outputs an embedding of the token that may include information about the whole sequence: $\bar{s}_i := g_\phi(i, S)$, where $\bar{s}_i$ is a sequence-aware embedding of the token s_i. All our generators f_θ and encoders g_ϕ are implemented as GPT-2 models [Radford et al. 2019]. To avoid notational clutter, we will omit the parameters θ and ϕ from the generator and encoder models f and g from here on.

Positional encoding. Both generators and encoders receive a *positional encoding* of the tokens in a sequence as input. A positional encoding consists of additional sequences that are used as input, but do not need to be generated, since they can be derived from the other sequences. They are application-specific and provide additional context for each token in the sequence, such as the sequence index of a token.

3.3 Node Generation

In the first step, we generate the operator types τ of all nodes $n = (\tau, P)$ in a graph as sequence $S^{\text{n}} = (\alpha, \tau_1, \tau_2, \dots, \omega)$, using a Transformer-based model f^{n}. α and ω are the sequence starting and stopping tokens defined in Section 3.2.

Sequence representation. A canonical ordering of the node sequence makes the generation task easier, as it provides more consistency between data samples, but also makes the model more inflexible, as it limits the diversity of the partial sequences $s_{<i}$ that the model is trained on. We experiment with four ordering strategies with different degrees of consistency; from most to least consistent:

- π_{r}: A back-to-front breadth-first traversal, starting from the output nodes, moving opposite to the edge directions from child nodes to parents, and traversing the parents in the order of a node's input slots.
- π_{rr}: Reversed π_{r} (i.e. from last to first node of π_{r}).
- π_{b}: A front-to-back breadth-first traversal, where children of a node are visited in the order of the node's output slots. We randomize the order of children that are connected to the same output slot.
- π_{t}: A random but valid topological node ordering.

We found that different orderings are suitable for different applications. We use π_{r} for unconditional generation, since it is the most consistent, and π_{rr} for our graph autocompletion, where a front-to-back ordering is beneficial.

Positional encoding and generation. In addition to the sequence S^{n}, we define two sequences as positional encoding. The sequence $S^{\text{ni}} = (1, 2, 3, \dots)$ provides the global position of a token in the sequence and $S^{\text{nd}} = (0, d_1, d_2, \dots, 0)$ describes the depth of each node in the graph, where d_i is the graph distance from n_i to the closest output node when using π_{r} or to the closest generator node for all other orderings. Since we cannot obtain the depth from the sequence S^{n} alone, we estimate it as additional output sequence during generation. The model f^{n} uses all three sequences as partial input sequences and is trained to output probabilities for the next operator type and depth:

$$p_i^{\text{n}}, p_i^{\text{nd}} = f^{\text{n}}(s_{<i}^{\text{n}}, s_{<i}^{\text{nd}}, s_{<i}^{\text{in}}). \quad (3)$$

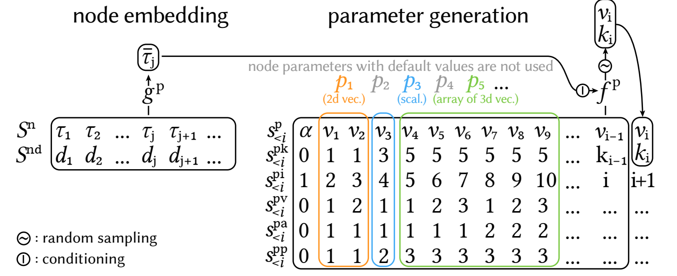

Fig. 3. **Parameter generation.** The parameter list $P = (p_1, p_2, \dots)$ of a node n_j is flattened into several sequences (right). Only parameters with non-default values are added to the sequence. Sequences are generated one token at a time by the Transformer-based generator f^{p}, which is conditioned on an embedding of n_j that is computed by the sequence encoder g^{p}.

3.4 Parameter Generation

Next, we generate the parameters P of nodes $n = (\tau, P)$ as a sequence $S^{\text{p}} = (\alpha, v_1, v_2, \dots, \omega)$ of scalar values v, using the Transformer-based model f^{p}. One parameter sequence is defined per node in a graph, with parameters sorted alphabetically by name. Figure 3 illustrates the process.

Conditioning on the node type. Parameter values depend on the operator type of the node n_i they are generated for, and to a lesser extent on the other nodes in the graph. We condition generation of each sequence S^{p} on a sequence-aware embedding $\bar{\tau}_j$ of the operator type τ_j that also includes information about the other nodes in the scene. The embedding is computed by a Transformer-based encoder $\bar{\tau}_j = g^{\text{p}}(j, S^{\text{n}}, S^{\text{nd}})$ that is trained jointly with the generator f^{p}.

Sequence representation. Typically, only a small fraction of node parameters are modified by an artist, the remaining parameters are left at their default values. We shorten the sequences S^{p} significantly by only including parameters that are not at their defaults. To identify which node parameter a given value v_i in our shortened sequence refers to, we define a second sequence $S^{\text{pk}} = (0, k_1, k_2, \dots, 0)$ of indices k into the full list of node parameters, i.e. value v_i corresponds to the node parameter with index k_i.

Node parameters are heterogeneous and may be scalars, vectors, and variable-length arrays of vectors. We flatten all parameters into a single sequence of scalars. Given the node type and the sequence S^{pk}, we can reconstruct parameters from the flattened sequence, since a node type and the parameter index fully define the type and vector dimension of a parameter. For variable length arrays of vectors, we obtain the array length by dividing the number of values generated for the parameter by the parameter's vector dimension. If the number is not evenly divisible, we discard the last values to make the number divisible. Since transformers operate with discrete token values, continuous parameters are quantized uniformly between their minimum and maximum observed values in the dataset. In our experiments, we use 32 quantization levels.

Positional encoding and generation. The positional encoding consists of the global token position sequence S^{pi}, and the sequences S^{pv}, S^{pa}, S^{pp}, which describe the index of the vector element, the

index of the array element, and the index of the parameter associated with a token, respectively. Probabilities over parameter values and indices are thus generated as:

$$p_i^{\mathrm{p}}, p_i^{\mathrm{pk}} = f^{\mathrm{p}}(s_{<i}^{\mathrm{p}}, s_{<i}^{\mathrm{pk}}, s_{<i}^{\mathrm{pi}}, s_{<i}^{\mathrm{pv}}, s_{<i}^{\mathrm{pa}}, s_{<i}^{\mathrm{pp}} \mid \bar{\tau}). \tag{4}$$

Appendix B gives architecture details of the conditional model f^{p}.

3.5 Edge Generation

In the last step, we generate all edges $e = (\mathtt{out}, \mathtt{in})$ in a graph as a sequence $S^e = (\alpha, l_1, l_2, l_3, l_4, \dots, \omega)$ of slot indices using the Transformer-based model f^{e}. Each pair of consecutive indices forms an edge. Figure 4 illustrates edge generation.

Slot embedding. We create a sequence of all input and output slots in the graph. An edge is a pair of indices into this sequence. A straightforward approach to edge generation would be generate a sequence of slot indices directly using a standard Transformer generator. However, this would not give the generator any information about the slots in the graph, thus it would be hard for the transformer to reason about slot connectivity. Instead, we follow the work of [Vinyals et al. 2015] and operate in the space of learnt slot embeddings. We compute embeddings $S^{\bar{\varsigma}} = (\bar{\varsigma}_1, \bar{\varsigma}_2, \dots)$ for every slot using a Transformer-based encoder g^{e} from multiple input sequences that together uniquely describe each slot in a node graph: $S^{\varsigma\tau}$, $S^{\varsigma\mathrm{ni}}$, $S^{\varsigma\mathrm{nd}}$ are sequences of operator types, node indices, and node depths, respectively, and provide information about the node associated with each slot, while sequence $S^{\varsigma\mathrm{k}}$ provides the index of the slot inside the node. A global token position sequence $S^{\varsigma\mathrm{i}}$ provides additional positional encoding. The slot embedding is then computed as $\bar{\varsigma}_j = g^{\mathrm{e}}(j, S^{\varsigma\tau}, S^{\varsigma\mathrm{ni}}, S^{\varsigma\mathrm{nd}}, S^{\varsigma\mathrm{k}}, S^{\varsigma\mathrm{i}})$. g^{e} is trained jointly with f^{e}.

Pointer Network. To generate indices into a list of embeddings, we use the approach proposed in Pointer Networks [Vinyals et al. 2015]. Our model f^{e} outputs a feature vector $\bar{l}_i$ in each step instead of a probability distribution. This feature vector acts as a query into the list of slot embeddings $S^{\bar{\varsigma}}$. Affinities between the query and the embeddings are computed as dot products and normalized into a probability distribution: $p_i^e = \mathrm{Softmax}_j(\bar{\varsigma}_j \cdot \bar{l}_i)$, where $\cdot$ denotes the dot product and $\mathrm{Softmax}_j$ denotes the softmax over all indices j. This distribution over slots is sampled to obtain the slot index l_i. The partial sequence of slot embeddings selected in the previous steps $S^{\mathrm{e}\bar{\varsigma}}$ is used as input to f^{e}, giving the model information about existing edges.

Positional encoding and generation. The positional encoding for the edge sequence consists of the global token position sequence S^{ei} and the tuple index of a token inside an edge $S^{\mathrm{et}} = (0, 1, 2, 1, 2, \dots, 0)$. Probabilities over slots indices are then generated as:

$$p_i^e = \mathrm{Softmax}_j \left(\bar{\varsigma}_j \; \cdot \; f^{\mathrm{e}}(s_{<i}^{\mathrm{e}\bar{\varsigma}}, s_{<i}^{\mathrm{ei}}, s_{<i}^{\mathrm{et}}) \right). \tag{5}$$

3.6 Training Setup

All models are trained with a binary cross-entropy loss over the probabilities estimated by the generators g. We use teacher forcing at training time, i.e. we use ground truth tokens for the partial sequences $s_{<i}$. Each step is trained separately using ground truth nodes as input for the node parameter and edge generation steps. We stop training once a minimum of the validation loss is reached to avoid overfitting. Additional details are provided in Appendix D.

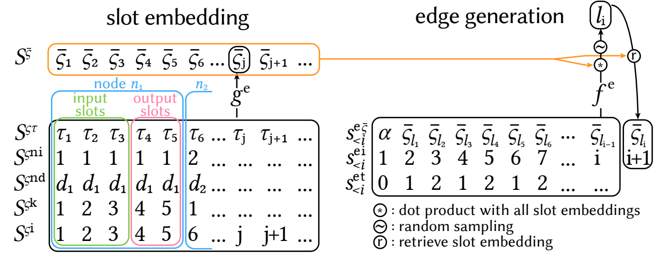

Fig. 4. **Edge Generation.** Edges are generated by a Transformer-based generator f^e as a sequence of index tuples that reference a sequence of input/output slots. These indices are generated using the mechanism proposed in Pointer Networks [Vinyals et al. 2015]. Slots embeddings are computed using the sequence encoder g^e, based on a set of sequences that uniquely identify each slot.

4 RESULTS

We evaluate MatFormer in two applications: *unconstrained generation* of materials and *guided authoring*, where our generative model assists an artist in material graph authoring by providing suggestions for possible completions of a partially constructed material graph. We first provide a few details about the dataset and the proceeded to experiments with the two applications.

Dataset. We use a subset of the Substance Source dataset [Adobe 2021b] to train our models, which is made of ~4k material graphs that were generated by experienced artists and are regularly used by a large community of professional artists. We filter out graphs with nodes that have a complex parameterization, such as nodes that are parameterized by a small pixel shader, leaving us with ~3.5k graphs. From these graphs, we remove graphs with an exceedingly large number of nodes (> 400) or edges (> 700), or with nodes that have an unusually large number of input (> 21) or output slots (> 14), resulting in a dataset of ~2.8k graphs.

We augment the dataset with random node parameter variations. Specifically, we perturb each parameter value v uniformly in the range $[0.8v, 1.2v]$. We create 100 graphs with random parameter perturbations for each graph in the original dataset, giving us ~280k graphs. Since we are interested in generation, we only keep a small validation set of 500 graphs, generated by perturbing the parameters of 5 distinct graphs in the original dataset. Random examples of augmented dataset graphs are shown in Figure 5, left.

4.1 Unconstrained Generation

We demonstrate the ability of our method to generate a wide range of diverse and realistic materials by generating ~1k samples (see supplementary material) and comparing them to several baselines, both qualitatively and quantitatively.

4.1.1 Baselines. We are not aware of any existing complete generator for procedural materials; therefore we introduce our baselines.

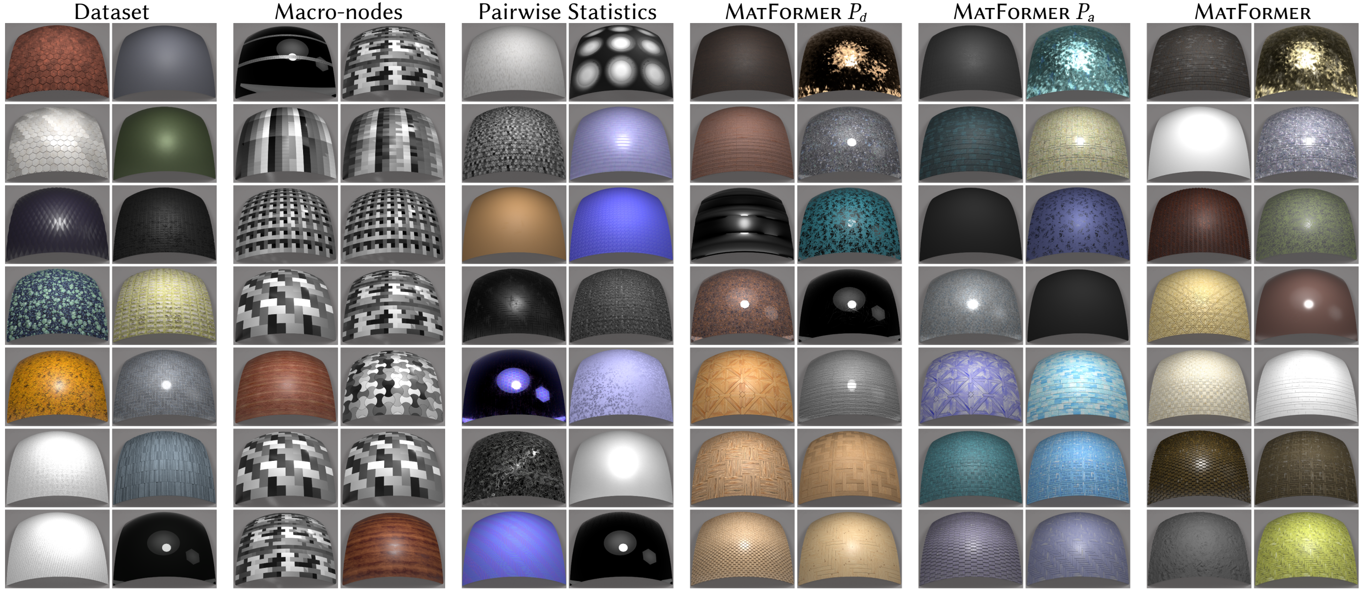

Fig. 5. **Qualitative comparison.** We compare 14 random samples taken from MatFormer to random samples from the dataset, the baselines, and the parameter ablations. From left to right: (i) dataset samples; (ii) generation from single *macro-nodes*; (iii) generation using pairwise node connectivity statistics; (iv) MatFormer P_d, our method without the node parameter generation step and all parameters are at their defaults; (v) MatFormer P_a, our method with all parameters at the dataset average; and (vi) MatFormer, our full method. Due to the long-range dependencies between nodes, our multistage Transformer-based approach generates material graphs that produce more diverse and realistic materials.

First, the *Macro-nodes* baseline uses graphs that consist only of a single *macro node.* A macro node is a node that is defined by a subgraph and typically performs complex tasks that are needed frequently, and that would need to be re-created in many graphs. Macro nodes can be quite powerful, sometimes allowing for fairly complex output on their own. We compare to this baseline to show that the complexity seen in our graphs does not originate solely from the functionality encapsulated in these macro nodes.

Second, we compare to the *Pairwise Statistics* baseline that uses pairwise node and slot connectivity statistics to generate a material graph. The baseline first collects dataset statistics that allowing us to create a probability distribution over the output slots and operator types that connect to each input slot on each operator type (including a *null* type if the input slot is not connected). Starting from the output nodes, we can then iteratively grow the graph, going through each unconnected input slot in turn, and deciding which slot on which operator type it should connect to, if any. Once decided, we can either connect the input slot to a newly created node of the given operator type, or connect it to any existing node of the given type, as long as edges do not form a cycle. We repeat this process until all input slots have been visited, or until a maximum node count is reached. After the graph nodes and edges have been generated, we create material parameters using our parameter generator.

4.1.2 Ablations. We evaluate the necessity for our parameter generator, and how different node orders influence the generation performance.

Two ablations compare our node parameter generation approach to simpler strategies. The ablation MatFormer P_d sets all node parameters to their default values, while MatFormer P_a sets them to the average observed in our dataset, and adds the same random perturbation we use to augment our dataset.

Additionally, we evaluate each of the node ordering strategies ($\pi_\mathrm{r}, \pi_\mathrm{rr}, \pi_\mathrm{b}, \pi_\mathrm{t}$) proposed in Section 3.3. We discuss the ablation results together with the baseline comparisons in Section 4.1.4.

4.1.3 Metrics. We use four quantitative metrics.

(i) The *graph statistics* $\mathcal{E}_\mathrm{g}$ over the material graph structure, measuring various graph properties including the longest path and the typical graph distance between each pair of operator types. A full description is given in Appendix C. We represent each of the statistics with a histogram and compare the histograms of our generated graphs with those of the dataset using the Earth Mover's Distance (EMD) [Rubner et al. 1998]; lower is better.

(ii) The *Frechet Inception Distance* (FID) [Heusel et al. 2017] $\mathcal{E}_\mathrm{fid}$ over the generated material samples. FID measures the similarity of two image distributions using simple statistics over the activations of an Inception network [Szegedy et al. 2015] when applied to an image set. We render material samples on a flat plane that is facing the camera, with a spot light illuminating the sample from the same direction as the camera, and compute the FID between our generated samples and the dataset distribution; lower is better.

(iii) The *normalized nearest neighbor style distance* $\mathcal{D}_\mathrm{nns}$ computes the distance from each rendered material of a generated graph to the closest dataset sample, using a style distance [Gatys et al. 2015a] based on the Gram matrix of VGG [Simonyan and Zisserman 2015] feature maps. We normalize this distance by dividing it with the average distance from each dataset sample to its closest neighbor in

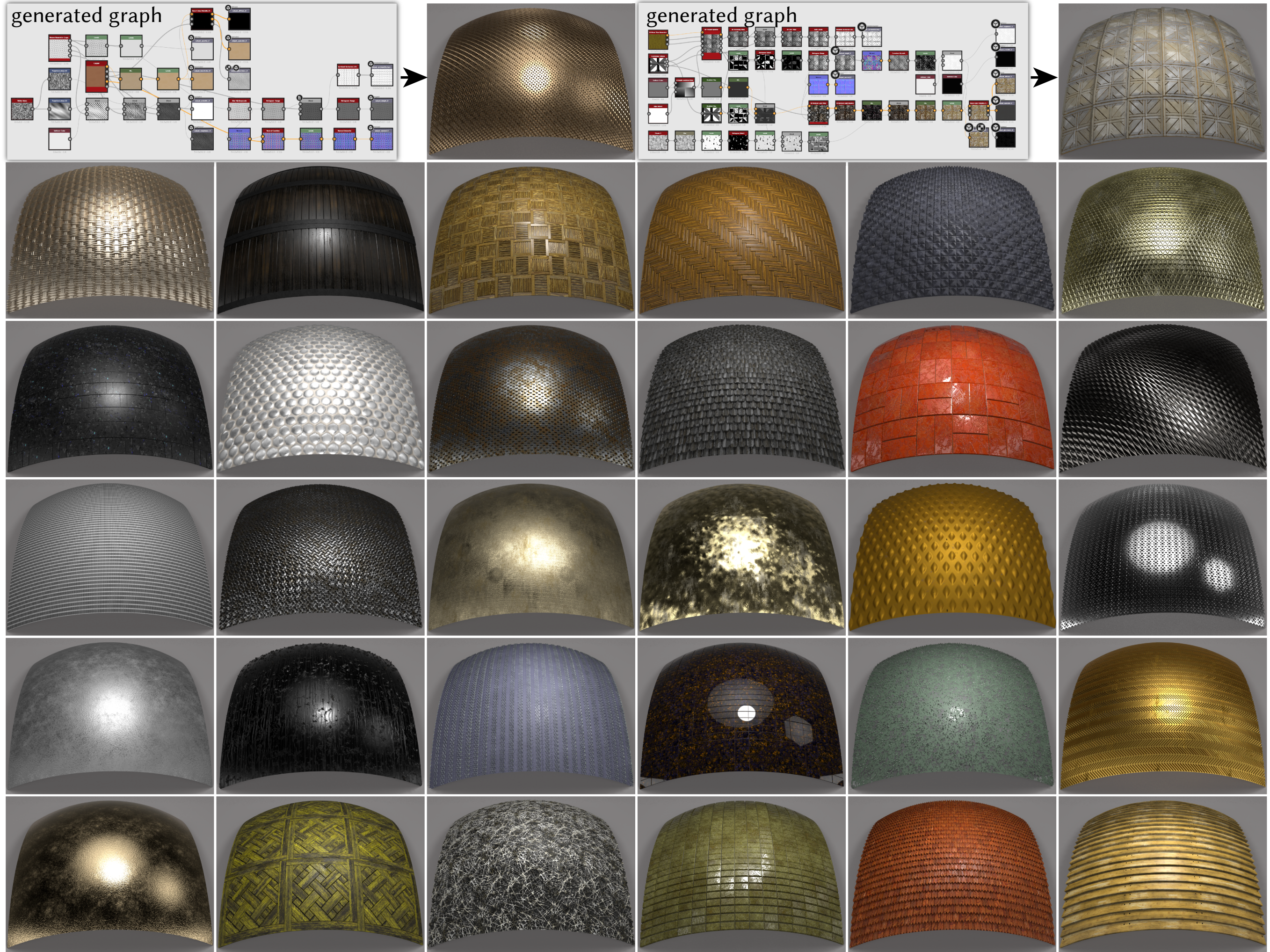

Fig. 6. **Material graph generation.** We show several materials obtained from our automatically generated material graphs. In the top row we illustrate the material graphs of two examples. Boxes are nodes displaying the node output; input slots are on the left side, and output slots on the right side of each node. Please note the wide variety of spatial patterns, normal and height variations (bumps), and specular highlights. Please refer to the supplemental for 1k additional materials.

the dataset. Values significantly below 1.0 indicate overfitting, while values significantly above 1.0 indicate a distribution of generated samples that is not fully aligned with the dataset distribution, thus values around 1.0 are ideal.

(iv) The *normalized nearest neighbor edit distance* $\mathcal{D}_{\text{nne}}$ is analogous to $\mathcal{D}_{\text{nns}}$, but uses the edit distance between material graphs as distance metric (insertions and removals of both nodes and edges). Due to the large computational cost of the graph edit distance, and since simple graphs are less likely to show interesting structural differences, we only consider graphs with ≥ 50 nodes.

4.1.4 Evaluation and Discussion. Quantitative comparisons to all baselines and ablations with ~1k generated graphs are shown in Table 1, and qualitative comparisons are given in Figure 5. For the quantitative comparison, we have trained four versions of our method, one for each node order π_r, π_{rr}, π_b, π_t described in Section 3.3, while for the qualitative comparison we use only π_r.

The results confirm that macro-nodes by themselves do not produce satisfactory results. Many macro-nodes produce output that needs further processing by additional nodes, such as the grayscale structures visible on many materials. Some materials, like the two wood materials, are reasonable but lack interesting structure and detail. The high FID scores and low distance D_{nns} confirm the trend we see in the qualitative results. By construction, this baseline does not have graphs with ≥ 50 nodes, thus D_{nne} cannot be computed.

The generator based on pairwise statistics performs better, but since it lacks the ability to model long-distance relationships, it can often get stuck with unrealistic outputs. The blue tint visible in many materials is likely due to normal conversion operators being

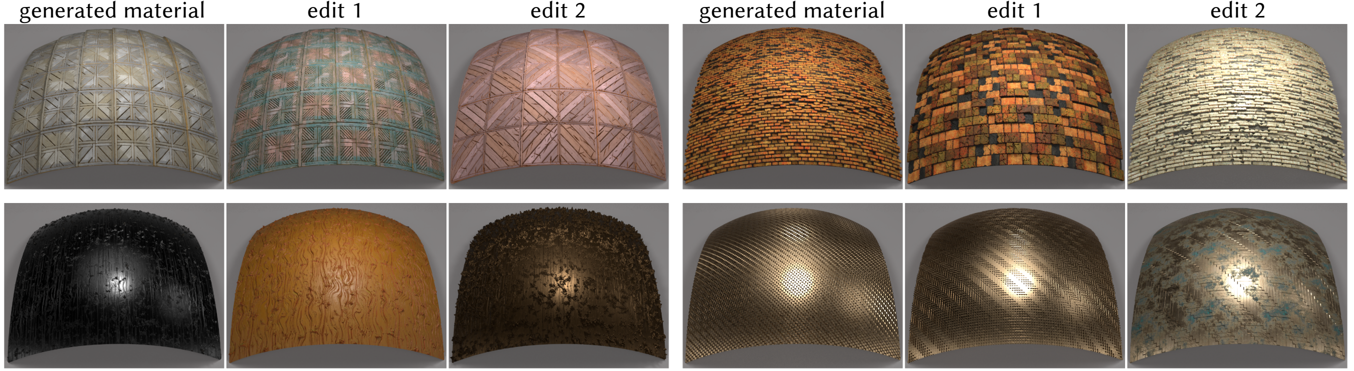

Fig. 7. **Editing generated materials.** Our generated materials being procedural, they are, by construction, fully editable. We show two edits for each of four generated materials. Each edit was done in under 10 minutes and no more than 10 node parameters were changed per edit.

applied incorrectly to a color channel. This is reflected in the high FID score. The high nearest neighbor distance $\mathcal{D}_{\text{nne}}$ shows that generated graphs are too far away from the manifold of dataset graphs.

The ablation MatFormer P_d with parameter values at their defaults uses node order π_r, so the most direct comparison is to the last row of Table 1. We can see that the graph structure metrics are comparable, since it is not affected by parameter generation. The FID score, however is significantly higher; we believe that this is due to the lack in diversity we can see in the qualitative results, resulting from parameters always being set to their default values. The FID score improves slightly in the ablation MatFormer P_a, where parameter values are set to the average value observed in the dataset plus a perturbation, but the FID is still significantly higher than when using our parameter generator. The style nearest neighbor distance is higher here due to less realistic materials, as evidenced by the high FID score.

Among the four node orderings, the back-to-front ordering π_r is the clear winner. We can see that there is a trend of increasing performance when going from most ambiguous ordering (π_t) to least ambiguous (π_r). Compared to the baselines, MatFormer has a clear advantage in all the metrics.

Unless noted otherwise, we will use the model trained with node order π_r for all unconstrained generation results, and the best-performing front-to-back node order π_{rr} for all user-guided authoring results.

Figure 6 shows several additional qualitative results of our method. We can see a diverse range of materials and structures, including metallic materials, bricks, wooden panels, etc. We show the corresponding generated material graphs for the two topmost examples. The supplementary material contains graph visualization and rendered materials for ~1k random samples.

Note that in our current approach parameter and edge generation are completely independent. However, the types and counts of graph nodes give strong cues about which parameters and edges to generate. Our results show that this already generates realistic graphs. Conditioning parameter generation on edges is an interesting direction for future work, for example by describing the local graph neighborhood around a node with a graph neural network.

Table 1. **Quantitative evaluation and comparison.** We compare MatFormer to two baselines, and evaluate its performance under different ablations. As metrics, we measure diversity and plausibility of the generations using FID scores ($\mathcal{E}_{\text{fid}}$); structure of the generated materials using graph statistics ($\mathcal{E}_{\text{g}}$); and novelty of the generations using a normalized distance to the nearest neighbors in the training dataset using both a style metric ($\mathcal{D}_{\text{nns}}$) and a graph edit distance ($\mathcal{D}_{\text{nne}}$), where values around 1.0 are optimal. Please refer to the text for details.

	$\mathcal{E}_{\text{g}}\downarrow$	$\mathcal{E}_{\text{fid}}\downarrow$	$\mathcal{D}_{\text{nns}}$	$\mathcal{D}_{\text{nne}}$
Baselines				
Macro-nodes	0.142	232.0	0.65	n/a
Pairwise Statistics	0.068	86.9	0.93	2.16
Ablations				
MatFormer P_d (n. order π_{r})	0.047	94.2	0.88	1.52
MatFormer P_a (n. order π_{r})	0.047	76.4	1.04	1.52
MatFormer (n. order π_{t})	0.068	66.4	0.93	2.21
MatFormer (n. order π_{b})	0.070	68.4	0.84	2.08
MatFormer (n. order π_{rr})	0.060	62.8	0.80	1.84
MatFormer (n. order π_{r})	**0.046**	**48.6**	**1.01**	**1.34**

4.1.5 Editing materials. Since the generated materials are procedural, we have full control over their appearance. Figure 7 shows examples of editing generated materials. We performed two edits on each of the four materials using Substance Designer [Adobe 2019], each editing session taking less than 10 minutes. Most of the time was spent on re-arranging the spatial layout of nodes (which is generated with a standard graph layout tool) in UI. The actual edits required at most 10 slider changes. The edits add diverse changes to the materials, such as weathering effects, color changes, or structure changes, such as different brick sizes, an increased number of wooden planks, and changed wood grain structure.

4.1.6 User Study. In addition to the editing examples, we evaluate the practical usability of our generated graphs compared to the baselines with a user study. We contacted two professional material

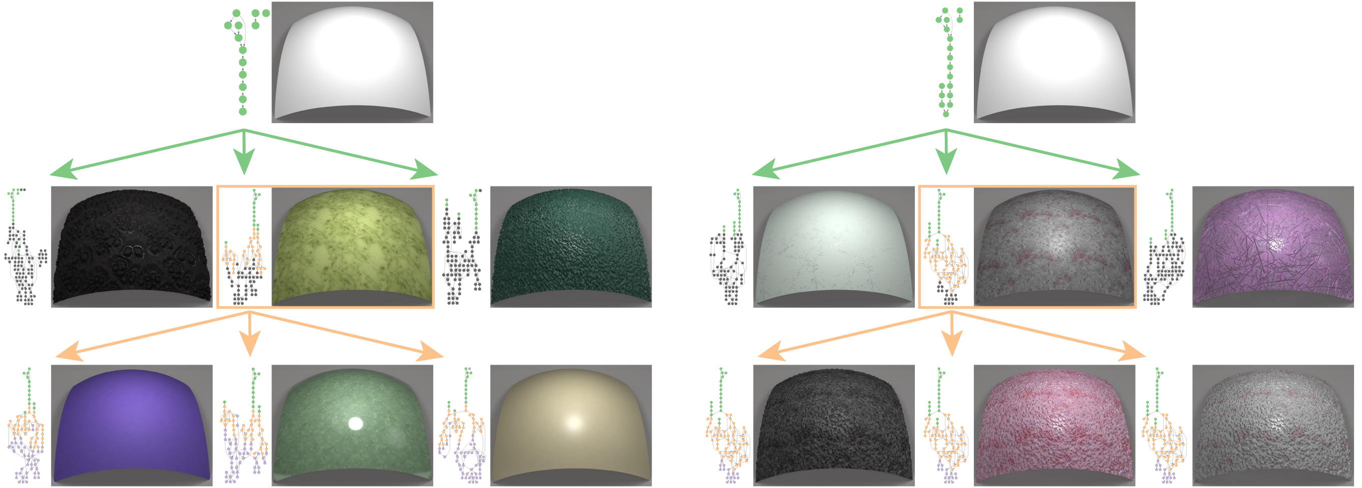

Fig. 8. **Guided material authoring.** We show two examples of a novel guided-authoring workflow. Starting from a partial material graph (top row), we generate three completions of the graph using our method (middle row). Material graphs are shown next to their output material. Nodes are illustrated as colored circles, where the color indicates node provenance. Green nodes come from the initial partial graph. The user selects one of the suggested completions (orange border), selects a subset of the nodes they want to keep (orange), and ask for another completion round (bottom row). In the left example, the final round of completions re-generates a large part of the graph (purple nodes), resulting in a varied material appearance. In the right example, a smaller part of the graph is re-generated, resulting in similar material structure for all three completions, since in this example, the structure is determined by the nodes that were held fixed (orange).

artists to evaluate generated graphs in a blind user study. Each artist was shown 20 graphs generated by MatFormer (π_r node order) and 10 graphs from each baseline, and tasked to (i) rate the usability of graph on a scale from 0 to 5, with 5 being the best score, and (ii) give a short feedback on each graph. The artists could inspect and interact with each graph as long as they wanted in Substance Designer. We received a score of 1.82 (40% of scores $\geq$ 3) compared to 0.35 (10% scores $\geq$ 3) for the Macro-nodes and 0.48 (0% scores $\geq$ 3) for the Pairwise baselines. This demonstrates the significant improvement provided by our method, as well as showing that there is still room for further research. Most prominent feedback for our method was that some materials are good and interesting, some are hard to understand, some produce oversaturated albedos, and some have unrealistic normals.

4.2 Guided Material Authoring

We can condition MatFormer on a partial graph by using the set of existing nodes and edges as partial input sequences $s_{<i}$. This enables an interesting application where our method provides guidance to a user that is working on a partial graph. We can suggest multiple different completions of the graph, and can show the final result of each completion (similar to using autocomplete in a text editor). The user can then select either all nodes of a preferred completion, obtaining a completed graph, or select a subset of the nodes in the completions, and continue exploring additional completions. If more nodes are selected from the completion, more properties of the material are held fixed in the following completions.

A few example completions are shown in Figure 8. Here we use a model trained with the node ordering π_{rr}; since users typically start with the front of the material graph, we need a front-to-back ordering. Each tree shown in the figure corresponds to two rounds of graph completions. We start with the graph at the root, and create three different completions (second row). A preferred completion is then selected, together with a subset of nodes (orange), and a second round of completions is performed, shown in the bottom row. The node color indicates which of the tree levels each node comes from: green is the top (root) level, orange is the middle level, purple the bottom level. Gray nodes are not passed to the next level.

In the examples shown in the Figure 8, we randomly select the nodes at each level of the tree. This approach can help novice users and can also be used by advanced users to explore material spaces.

5 CONCLUSION AND FUTURE WORK

We have presented MatFormer—a transformer-based autoregressive model that can generate material graphs, consisting of operation nodes, node parameters, and edges that describe information transmission between nodes. We quantitatively and qualitatively evaluated the quality and diversity of the generated materials. As an application, we presented a novel guided-authoring workflow where the user can progressively create complex material graphs by selecting from autocompletions proposed by our generator. While MatFormer is constrained by the diversity of the procedural graphs it is trained on, we expect it to improve as these datasets expand.

MatFormer is, to the best of our knowledge, the first generative model for procedural material graphs. We believe it is an important step towards bringing the power of deep generative models to a critical part of 3D content creation pipelines. We also believe that our work opens up a number of important research directions that we would hope will be addressed in the course of future efforts.

Projecting images to material graphs. In its present form, MatFormer does not support matching a target material image by conditioning the generation on a target image. Unlike GAN inversion in the context of images [Abdal et al. 2019], here the task is more complex due to the autoregressive generator setup, and the fact that generating an image requires evaluating the *entire* graph. An efficient scheme to backpropagate from images to graphs would enable many possibilities including image-conditioned graph generation and training graph generation supervised by only (large) image datasets instead of (much smaller) procedural material datasets.

A possible avenue is to train another network (i.e., an encoder network) to predict intermediate subgraphs of the target graph, and then use the proposed autocomplete framework, possibly in conjunction with a beam search, to iteratively search and explore the solution space. Once we obtain a good material graph structure, we can do a parameter refinement using a material-specific optimization setup [Shi et al. 2020]. A challenge here is to generate material graphs that preserve any mesostructure contained in the target images, possibly requiring a new material similarity metric.

Semantic control handles. MatFormer can be sampled to produce novel material graphs along with default parameters. However, these graphs often span 50+ nodes, making them complex to work with, without knowing the effect of the different node parameters. Artists, in contrast, often expose a small subset of parameter sliders, usually with semantic associations, to simplify editing the material graphs. In the future, it would be interesting to similarly expose a subset of the parameters, along with relevant value ranges, ideally with semantic attributes similar to latent spaces for portrait editing with StyleGAN [Abdal et al. 2021; Härkönen et al. 2020].

Discovered macros. An interesting next step would be to discover commonly recurring subgraphs across multiple generated graphs. If successful, this would allow discovering *metanodes*, which capture group of operations recurring across examples in the training set. This is similar to the problem of program synthesis where program macros are extracted by considering e-graphs of equivalent operations [Ellis et al. 2021] in a bottom-up fashion. If successful, automatically discovered high-level metanodes can be reused for more abstracted material graph generation.

Hierarchical output. At present, an end user has no guidance as to what the different parts of a generated graph represent. As a result, manipulating the graph remains tedious. In the future, we would like to produce hierarchically structured graphs. Similar to coarse-to-fine synthesis in the case of progressive GANs, a possible option is to have stages to first produce coarse (macro) structures, then add local properties (e.g., color, shininess), and finally spatially varying effects arising from weathering. An immediate challenge is to find training data for this hierarchy.

ACKNOWLEDGMENTS

We would like to thank Romain Rouffet, Luc Chamerlat, Geoffrey Rosin, and Gaetan Lassagne for their time, suggestions, and helpful feedback.

REFERENCES

Rameen Abdal, Yipeng Qin, and Peter Wonka. 2019. Image2StyleGAN: How to Embed Images Into the StyleGAN Latent Space?. In *ICCV*.

Rameen Abdal, Peihao Zhu, Niloy J. Mitra, and Peter Wonka. 2021. StyleFlow: Attribute-conditioned Exploration of StyleGAN-Generated Images using Conditional Continuous Normalizing Flows. In *ACM SIGGRAPH*.

Adobe. 2019. Substance. www.substance3d.com.

Adobe. 2021a. Substance. https://substance3d.adobe.com/community-assets.

Adobe. 2021b. Substance. https://substance3d.adobe.com/assets.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450* (2016).

Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the Properties of Neural Machine Translation: Encoder–Decoder Approaches. In *SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*. Association for Computational Linguistics.

Kingma Diederik, Ba Jimmy, et al. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014), 273–297.

Tao Du, Jeevana Priya Inala, Yewen Pu, Andrew Spielberg, Adriana Schulz, Daniela Rus, Armando Solar-Lezama, and Wojciech Matusik. 2018. InverseCSG: Automatic Conversion of 3D Models to CSG Trees. In *ACM SIGGRAPH Asia*.

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B Tenenbaum. 2021. Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *ACM SIGPLAN*.

Bruno Galerne, Ares Lagae, Sylvain Lefebvre, and George Drettakis. 2012. Gabor Noise by Example. In *ACM SIGGRAPH*.

Leon A Gatys, Alexander S Ecker, and Matthias Bethge. 2015a. A neural algorithm of artistic style. *arXiv preprint arXiv:1508.06576* (2015).

Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. 2015b. Texture synthesis and the controlled generation of natural stimuli using convolutional neural networks. *CoRR* abs/1505.07376 (2015).

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative Adversarial Nets. In *NeurIPS*.

Yu Guo, Miloš Hašan, Lingqi Yan, and Shuang Zhao. 2020a. A Bayesian Inference Framework for Procedural Material Parameter Estimation. *Computer Graphics Forum* 39, 7 (2020).

Yu Guo, Cameron Smith, Miloš Hašan, Kalyan Sunkavalli, and Shuang Zhao. 2020b. MaterialGAN: Reflectance Capture Using a Generative SVBRDF Model. *ACM Trans. Graph.* 39, 6, Article 254 (2020), 13 pages.

Erik Härkönen, Aaron Hertzmann, Jaakko Lehtinen, and Sylvain Paris. 2020. GANSpace: Discovering Interpretable GAN Controls. In *NeurIPS*.

Trevor Hastie, Robert Tibshirani, Jerome H Friedman, and Jerome H Friedman. 2009. *The elements of statistical learning: data mining, inference, and prediction*. Vol. 2. Springer.

Philipp Henzler, Niloy J Mitra, and Tobias Ritschel. 2019. Learning a Neural 3D Texture Space from 2D Exemplars. In *IEEE CVPR*.

Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. 2017. Gans trained by a two time-scale update rule converge to a local nash equilibrium. In *NeurIPS*.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

Rein Houthooft, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel. 2016. Curiosity-driven Exploration in Deep Reinforcement Learning via Bayesian Neural Networks. *CoRR* abs/1605.09674 (2016).

Yiwei Hu, Julie Dorsey, and Holly Rushmeier. 2019. A Novel Framework for Inverse Procedural Texture Modeling. *ACM Trans. Graph.* 38, 6 (Nov. 2019), 186:1–186:14.

Yiwei Hu, Chengan He, Valentin Deschaintre, Julie Dorsey, and Holly Rushmeier. 2022. An Inverse Procedural Modeling Pipeline for SVBRDF Maps. *ACM Trans. Graph.* 41, 2 (2022), 1–17.

R. Kenny Jones, Theresa Barton, Xianghao Xu, Kai Wang, Ellen Jiang, Paul Guerrero, Niloy J. Mitra, and Daniel Ritchie. 2020. ShapeAssembly: Learning to Generate Programs for 3D Shape Structure Synthesis. In *ACM SIGGRAPH Asia*.

Tero Karras, Samuli Laine, and Timo Aila. 2019. A Style-Based Generator Architecture for Generative Adversarial Networks. In *IEEE CVPR*.

Tero Karras, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. 2020. Analyzing and Improving the Image Quality of StyleGAN.

Manoj Kumar, Mohammad Babaeizadeh, Dumitru Erhan, Chelsea Finn, Sergey Levine, Laurent Dinh, and Durk Kingma. 2020. VideoFlow: A Flow-Based Generative Model for Video. In *ICML*.

Laurent Lefebvre and Pierre Poulin. 2000. Analysis and Synthesis of Structural Textures. In *Graphics Interface*.

Jun Li, Kai Xu, Siddhartha Chaudhuri, Ersin Yumer, Hao Zhang, and Leonidas J. Guibas. 2017. GRASS: Generative Recursive Autoencoders for Shape Structures. In *ACM SIGGRAPH*.

Xiao Li, Yue Dong, Pieter Peers, and Xin Tong. 2019. Synthesizing 3d shapes from silhouette image collections using multi-projection generative adversarial networks. In *IEEE CVPR*.
Kaichun Mo, Paul Guerrero, Li Yi, Hao Su, Peter Wonka, Niloy Mitra, and Leonidas Guibas. 2019. StructureNet: Hierarchical Graph Networks for 3D Shape Generation. In *ACM SIGGRAPH Asia*.
Charlie Nash, Yaroslav Ganin, S. M. Ali Eslami, and Peter Battaglia. 2020. PolyGen: An Autoregressive Generative Model of 3D Meshes. In *ICML*.
Aaron Van Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. 2016. Pixel Recurrent Neural Networks. In *ICML*.
Wamiq Para, Shariq Bhat, Paul Guerrero, Tom Kelly, Niloy Mitra, Leonidas J. Guibas, and Peter Wonka. 2021. SketchGen: Generating Constrained CAD Sketches. In *NeurIPS*.
Darwyn R. Peachey. 1985. Solid Texturing of Complex Surfaces. In *ACM SIGGRAPH*.
Ken Perlin. 1985. An Image Synthesizer. In *ACM SIGGRAPH*.
Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.
Leo Sampaio Ferraz Ribeiro, Tu Bui, John Collomosse, and Moacir Ponti. 2020. Sketchformer: Transformer-based Representation for Sketched Structure. In *IEEE CVPR*.
Elad Richardson, Yuval Alaluf, Or Patashnik, Yotam Nitzan, Yaniv Azar, Stav Shapiro, and Daniel Cohen-Or. 2021. Encoding in Style: a StyleGAN Encoder for Image-to-Image Translation. In *IEEE CVPR*.
Yossi Rubner, Carlo Tomasi, and Leonidas J Guibas. 1998. A metric for distributions with applications to image databases. In *IEEE ICCV*.
Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. The Graph Neural Network Model. *IEEE Transactions on Neural Networks* 20, 1 (2009), 61–80.
Liang Shi, Beichen Li, Miloš Hašan, Kalyan Sunkavalli, Tamy Boubekeur, Radomir Mech, and Wojciech Matusik. 2020. Match: Differentiable Material Graphs for Procedural Material Capture. In *ACM SIGGRAPH Asia*.
Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *ICLR*.
Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *IEEE CVPR*.
Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NeurIPS*.
Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer Networks. In *NeurIPS*.
Karl D. D. Willis, Yewen Pu, Jieliang Luo, Hang Chu, Tao Du, Joseph G. Lambourne, Armando Solar-Lezama, and Wojciech Matusik. 2021. Fusion 360 Gallery: A Dataset and Environment for Programmatic CAD Reconstruction. *ACM Trans. Graph.* 40, 4 (2021).
Andrew Witkin and Michael Kass. 1991. Reaction-Diffusion Textures. In *ACM SIGGRAPH*.
Steven Worley. 1996. A Cellular Texture Basis Function. In *ACM SIGGRAPH*.
Rundi Wu, Chang Xiao, and Changxi Zheng. 2021. DeepCAD: A Deep Generative Network for Computer-Aided Design Models. In *IEEE ICCV*.
Cheng-Fu Yang, Wan-Cyuan Fan, Fu-En Yang, and Yu-Chiang Frank Wang. 2021. LayoutTransformer: Scene Layout Generation With Conceptual and Spatial Diversity. In *IEEE CVPR*.
Xilong Zhou and Nima Khademi Kalantari. 2021. Adversarial Single-Image SVBRDF Estimation with Hybrid Training. *Computer Graphics Forum* 40, 2 (2021), 315–325.
Jiapeng Zhu, Yujun Shen, Deli Zhao, and Bolei Zhou. 2020. In-domain GAN Inversion for Real Image Editing. In *ECCV*.

A SEMANTIC VALIDITY CHECKS

Parameter generation. We use the semantic validity approach described in Section 3.2 to constrain generated token values to represent valid node graphs. We do not perform semantic validity checks for the node generation step. For node parameter generation, we constrain the value of discrete parameters within their valid range. Similarly, we constrain the parameter indices in the sequence S^{pk} to not exceed the number of number of parameters in the corresponding operator o_τ.

Edge generation. We use the semantic validity approach to constrain edges to always go from output slots to input slots, i.e., the first element of an edge tuple has to refer to an output slot, the second element to an input slot. Additionally, we make sure edges do not form cycles by constraining edge end points to nodes that are not ancestors of the node at the edge start point.

B CONDITIONAL TRANSFORMER GENERATOR

To condition a Transformer generator on a feature vector, we add three layers to each feed-forward block of the transformer (see Viswani et al. [2017] for a detailed description of the Transformer blocks). The original block performs the following operations:

$$\texttt{FFBlock}(x) \coloneqq x + \text{MLP}(\text{LN}(x)), \tag{6}$$

where x is the input feature vector, MLP is a Multilayer Perceptron [Hastie et al. 2009], and LN is Layer Normalization [Ba et al. 2016]. We modify this block to take the condition as additional feature vector c:

$$\texttt{CFFBlock}(x, c) \coloneqq \texttt{FFBlock}(x) + \text{MLP}(\text{LN}(c)), \tag{7}$$

C GRAPH STATISTICS

We compute the following graph statistics:

(1) The number of nodes.
(2) The number of disconnected components.
(3) The longest possible path in the graph.

Additionally, several statistics that are computed per operator type, or per pair of operator types give more fine-grained information:

(1) The number of nodes for each operator type in a graph.
(2) The graph distance between each node of a given operator type and the closest output node.
(3) The number of connected input slots for each operator type.
(4) The graph distance between all pairs of operator types.

D ARCHITECTURE AND TRAINING DETAILS

Architecture. All our transformer models use the GPT-2 architecture [Radford et al. 2019] with 2 attention layers, 4 attention heads in each layer, and 64-dimensional hidden features. We use a maximum sequence length (not counting start/stop tokens) of 400 nodes for the node generator f^{n} and node encoder g^{p}, 512 scalar parameter values for the parameter generator f^{p}, 700 edges (1400 tokens) for the edge generator f^{e}, and 800 slots for the slot encoder g^{e}. The number of different node types in our experiments is 1207, this is also the vocabulary size of the main input sequence of the node generator, node encoder, and slot encoder.

Training. We train all models with the Adam optimizer [Diederik et al. 2014] using a learning rate of 1e−4, a batch size of 64 for the node and parameter generators, and a batch size of 16 for the edge generator. For all three generators, we stop training once a minimum of the validation loss is reached, which happens in our experiments after 30 epochs for the node generator, 11 epochs for the parameter generator, and 10 epochs for the edge generator. With one V100 GPU for each of the three models, training takes roughly 2-3 days.