# A Novel Framework For Inverse Procedural Texture Modeling

YIWEI HU, Yale University
JULIE DORSEY, Yale University
HOLLY RUSHMEIER, Yale University

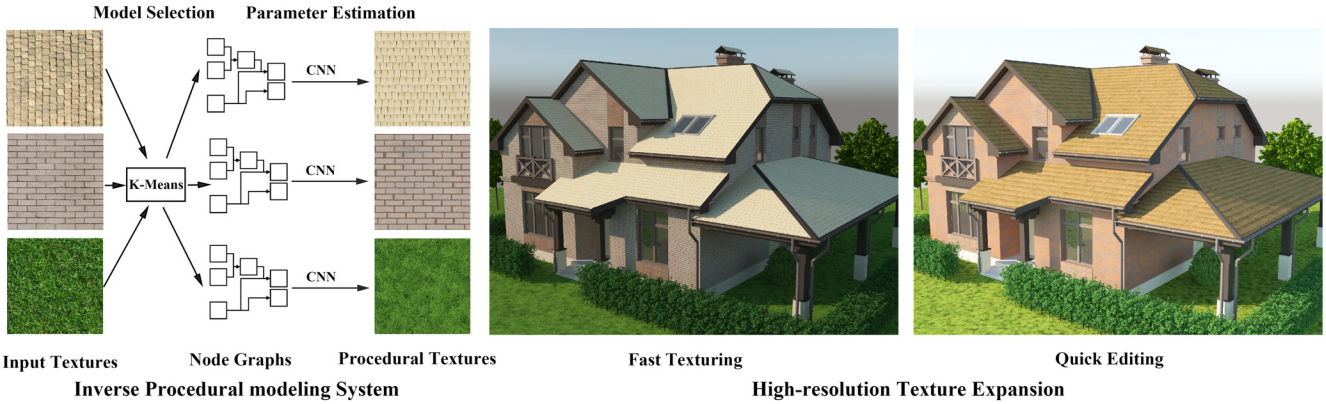


Fig. 1. Our novel framework for inverse procedural texture modeling by example. Given an input texture exemplar, our inverse modeling system can choose appropriate procedural texture models e.g. node graphs and apply pre-trained CNNs to estimate corresponding parameters (left). With chosen models and predicted parameters, procedural textures can be generated and expanded to arbitrary resolution such as 4K or 8K. High-quality and fast scene texturing can be achieved (average 160ms for 4K textures and 1.5s for 8K textures) and users can edit the procedural textures quickly and conveniently by tuning parameters (right).

Procedural textures are powerful tools that have been used in graphics for decades. In contrast to the alternative exemplar-based texture synthesis techniques, procedural textures provide user control and fast texture generation with low-storage cost and unlimited texture resolution. However, creating procedural models for complex textures requires a time-consuming process of selecting a combination of procedures and parameters. We present an example-based framework to automatically select procedural models and estimate parameters. In our framework, we consider textures categorized by commonly used high level classes. For each high level class we build a data-driven inverse modeling system based on an extensive collection of real-world textures and procedural texture models in the form of node graphs. We use unsupervised learning on collected real-world images in a texture class to learn sub-classes. We then classify the output of each of the collected procedural models into these sub-classes. For each of the collected models we train a convolutional neural network (CNN) to learn the parameters to produce a specific output texture. To use our framework, a user provides an exemplar texture image within a high level class. The system first classifies the texture into a sub-class, and selects the procedural models that produce output in that sub-class. The pre-trained CNNs of the selected models are used to estimate the parameters of the texture example. With the predicted parameters, the system can generate appropriate procedural textures for the user. The user can easily edit the textures by adjusting the node graph parameters. In a last optional step, style transfer augmentation can be applied to the fitted procedural textures to recover details lost in the procedural modeling process. We demonstrate our framework for four high level classes and show that our inverse modeling system can produce high-quality procedural textures for both structural and non-structural textures.

CCS Concepts: • **Computing methodologies** → **Appearance and texture representations**; **Texturing**.

Additional Key Words and Phrases: procedural textures, inverse procedural modeling, convolutional neural networks

**ACM Reference Format:**
Yiwei Hu, Julie Dorsey, and Holly Rushmeier. 2019. A Novel Framework For Inverse Procedural Texture Modeling. *ACM Trans. Graph.* 38, 6, Article 186 (November 2019), 14 pages. https://doi.org/10.1145/3355089.3356516

## 1 INTRODUCTION

Procedural textures have been used extensively in computer graphics because of their compactness and flexibility. With procedural models, texture synthesis becomes straightforward and texture images of arbitrary size can be generated rapidly. However, existing authoring systems for procedural textures rely on skilled users to handcraft procedural models and carefully choose a set of proper parameters to achieve a desired texture appearance. Frequently designers will use a node graph such as Fig. (2) to represent a complex

Authors' addresses: Yiwei Hu, Yale University, yiwei.hu@yale.edu; Julie Dorsey, Yale University, julie.dorsey@yale.edu; Holly Rushmeier, Yale University, holly.rushmeier@yale.edu.


0730-0301/2019/11-ART186 $15.00
https://doi.org/10.1145/3355089.3356516

procedural texture model and generate SVBRDF (spatially varying bidirectional reflectance distribution function) maps e.g. albedo, normal, roughness and metallic maps. The node graph can consist of dozens of nodes where each node represents a type of operation and may require many parameters. Building such a node graph from scratch and tuning parameters to achieve satisfactory textures is time-consuming and can be very hard for a non-professional user. On the other hand, exemplar-based texture modeling has been widely studied for texture synthesis [Efros and Freeman 2001] [Kwatra et al. 2003]. These methods take a texture exemplar to create new larger scale texture image with the same appearance. This *inverse* modeling process has many advantages over a forward texture design process. Even a novice user can specify their desired texture appearance by example. However, the ability to edit and rapidly generate large texture images is lost.

In this paper, we propose a new framework for inverse procedural texture modeling by example (shown in Fig. (1)). Our goal is to find appropriate procedural texture models, e.g. node graphs, and a set of parameters to fit the user-specified texture exemplar. Since node graphs can be quite complex, it can be difficult to directly infer the varied structures inside a node graph given only a texture sample. Instead, we design our system in a data-driven way. We note that in the designer community, many elaborate procedural models are shared and available online. These procedural texture models as well as real-world textures are naturally classified into different high level texture classes. For example, in the community sharing system in [https://share.substance3d.com/], material classes such as leather, rock, and wood are used. Based on this observation, we build a database of real-world textures and corresponding procedural texture models according to different high level texture classes. We study the relationship between real-world natural image textures and procedural textures so that we can select matched procedural models which share high visual similarity with the given texture exemplar. These selected node graphs can be powerful enough to approximate the given real-world texture exemplar. To achieve this, we perform unsupervised learning on collected real-world textures to find out different sub-classes within a texture class. Sub-classes can represent different styles of textures in a class. Afterwards, the system samples the procedural texture models of the same class in the database to generate procedural texture samples, and classify these procedural samples into sub-classes. For each procedural texture model we train a CNN (Convolutional Neural Network) to learn the procedural parameters to obtain a particular texture.

In our framework, a user presents an example texture image for a selected high level class. Our system then identifies the sub-class for the texture, selects appropriate procedural texture models, and uses the corresponding trained networks to find the model parameters. However, even elaborated procedural models may still lack of the ability to fully capture the fine-grained spatial variations in real-world textures. A visually non-negligible gap between procedural textures and captured images exists. To further enhance the fidelity of our generated procedural textures, we propose an optional style transfer augmentation step [Gatys et al. 2016] using our procedural texture as the content image and using the given sample image as the style image. This style transfer step can faithfully reconstruct original sample images via transforming predicted procedural textures back to pixel domain though at the expense of loss of editability. Our results show this style transfer is an effective way to bridge the gap between procedurally generated images and real-world images.

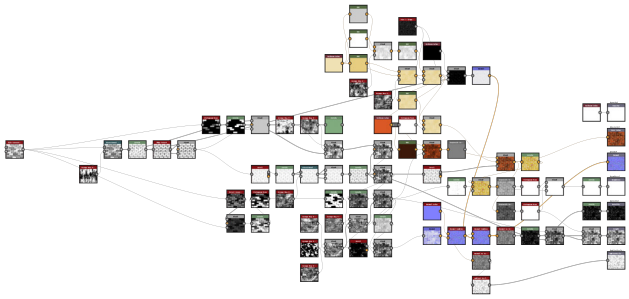

Fig. 2. A procedural texture model can be represented as a node graph. As an example, this node graph used in Allegorithmic Substance Designer [Allegorithmic 2019b] can generate a brick texture with 7 SVBRDF maps including albedo map, normal map, roughness map, metallic map, height map, ambient occlusion map and opacity map.

To illustrate our framework, we demonstrate a system built on a database of procedural texture assets including four texture classes for building materials – bricks, grass, shingles and stucco. Our results show our inverse modeling system can gain high-quality procedural textures for both structural and non-structural textures. Our experimental results further demonstrate that these approximated procedural textures can facilitate convenient 3D scene design.

Our main contributions are:

- We propose a novel framework for inverse procedural texture modeling based on a database of real-world textures and procedural texture assets to select style-matched procedural models.
- We present an example-based inverse procedural texture modeling method using Convolutional Neural Networks.
- We introduce a style transfer augmentation step to bridge the gap between procedural textures and real-world images.
- By predicting procedural texture parameters automatically, our approach offers a greater level of control and expressiveness in the generation of desired textures by enabling easy editing of a procedural model.

## 2 RELATED WORK

### 2.1 Texture Authoring Systems

Texture authoring systems frequently use databases [Allegorithmic 2019b; Blender 2019b] containing a wide range of different textures. Often users choose to search for their desired textures in the database rather than create their own textures from scratch. Since textures are naturally classified into different classes (fabric, wood, metal, etc.), users will first look for textures within a specific texture class and they will start a fine-grained visual search for different styles of texture in that class. Our framework is inspired by this general idea. We build our model selection scheme by studying the sub-classes of a texture class based on style differences.

### 2.2 Inverse Procedural Texture Modeling

Inverse procedural texture modeling can be considered as an example-based inverse procedural modeling problem. While early works [Dischler et al. 2002; Lefebvre and Poulin 2000] apply image analysis to extract the structural and local information from texture samples, their methods use manually crafted templates which are difficult to generalize into complex procedural texture models. For instance, Lefebvre and Poulin [2000] proposed a handcrafted method to estimate the parameters of procedural brick and wood textures. However, these handcrafted rules are very sensitive to noise and always need user intervention, while our CNN-based approach can invert more complex models without user intervention and manually designed rules.

Additional works have considered procedural noise models. Lagae et al. [2010] introduced a method to evaluate the weights of procedural multiresolution noise which can be used for texture synthesis for isotropic stochastic procedural texture samples. Galerne et al. [2012] proposed an example-based method to estimate the parameters of bandwidth-quantized Gabor noise using non-negative basis pursuit denoising. Later, Local Randon-Phase Noise [Gilet et al. 2014], Texton Noise [Galerne et al. 2017] and Histogram-Preserving Blending Operator [Heitz and Neyret 2018] were developed to achieve high quality noise textures with faster speed. In contrast to procedural noise by example, we consider the general procedural texture appearance problem and emphasize large-scale patterns. Other work e.g. [Nishida et al. 2016] has applied machine learning to inverse procedural modeling of 3D shape but not of texture.

### 2.3 Texture Synthesis

Texture synthesis has been an active research area in computer graphics for many years. Most texture synthesis approaches are based on pixel representation and do not synthesize procedural textures. Wei et al. [2009] provided a comprehensive overview of classic example-based texture synthesis methods. While pixel-based synthesis methods [Efros and Leung 1999; Wei and Levoy 2000] work by generating output texture pixel by pixel based on local neighborhood searching, patch-based synthesis methods [Efros and Freeman 2001; Kwatra et al. 2003] proceed by copying and pasting patches of sample textures to output textures. Optimization-based texture synthesis was proposed by [Kwatra et al. 2005] who posed texture synthesis as an energy optimization problem. It is worth noticing that Wei et al. [2008] proposed an inverse texture synthesis method. While their approach aims to produce a small texture compaction to summarize the original texture, our framework focuses on transforming pixel-based texture representation to graph-based representation.

In contrast to classic texture synthesis works, Gatys et al. [2015] proposed a deep learning based approach to synthesize new textures given the texture sample. They used a CNN pre-trained on object recognition task to extract features from the texture sample and solve the texture synthesis problem by running an optimization algorithm on a target image initialized as a noise image. The gradient descent process let features of the target image gradually match the features of the texture sample. Ulyanov et al. [2016] introduced an accelerated version of this texture synthesis process by replacing the optimization process with a single forward prediction pass. Zhou et al. [2018] proposed a non-stationary texture synthesis approach using a Generative Adversarial Network (GAN). They adapted a self-supervised training strategy and they demonstrated their approach can iteratively expand the source textures into extremely large size.

However, most of these texture synthesis approaches are still limited in the pixel domain which means a synthesized higher resolution texture will consume more storage space and is still not suitable for editing. Our framework can provide users a more compact procedural representation which is useful for design iteration.

### 2.4 SVBRDF Acquisition With Deep Learning

SVBRDF acquisition with deep learning aims to apply learning-based methods to decompose captured natural images into SVBRDF maps such as albedo maps, normal maps etc. Aittala et al. [2016] introduced an optimization-based SVBRDF reconstruction method to infer SVBRDF maps from a single head-lit near-plane texture sample via CNN-based texture statistics comparison. Li et al. [2017a] designed an auto-encoder style neural network with self-augmentated training process to directly decompose captured images into SVBRDF maps in a single forward pass. Subsequent works proceeded to refine the quality of the reconstructed results by introducing an in-network rendering layer [Deschaintre et al. 2018; Li et al. 2018a]. Recently, Li et al. [2018b] proposed a physically-motivated network to simultaneously reconstruct the unknown shape and SVBRDF maps from a single image of the object sample.

Our approach, to some extent, is similar to SVBRDF map decomposition. The aforementioned methods all essentially pose this problem as learning a mapping from one image space to another image space. However, our system attempts to directly predict the parameters of the node graphs and then use the node graphs with the predicted parameters to generate corresponding SVBRDF maps.

## 3 METHOD

In this section, we describe the detailed implementation of our framework for inverse procedural texture modeling. We build a data-driven inverse modeling system based on a database of real-world natural textures and procedural texture assets. Thanks to various exchange platform in the designer community, we first collect enough real-world textures and procedural texture models to drive our system (Sec. 3.1). We present a model selection method to automatically select best matched procedural assets based on the style similarity (Sec. 3.2). For each procedural texture model, we train a CNN to estimate appropriate parameters for texture sample inputs (Sec. 3.3). During the evaluation phase, given a user-specified texture exemplar, our system can provide suitable procedural texture models with predicted parameters, and an optional style transfer augmentation can be applied (Sec. 3.4). Fig. (3) shows an overview of our system's workflow. We also discuss the scalability of our system in Sec. 3.5.

### 3.1 Data Collection

Since textures can be naturally classified into different classes, our texture database is built by collecting real-world natural textures

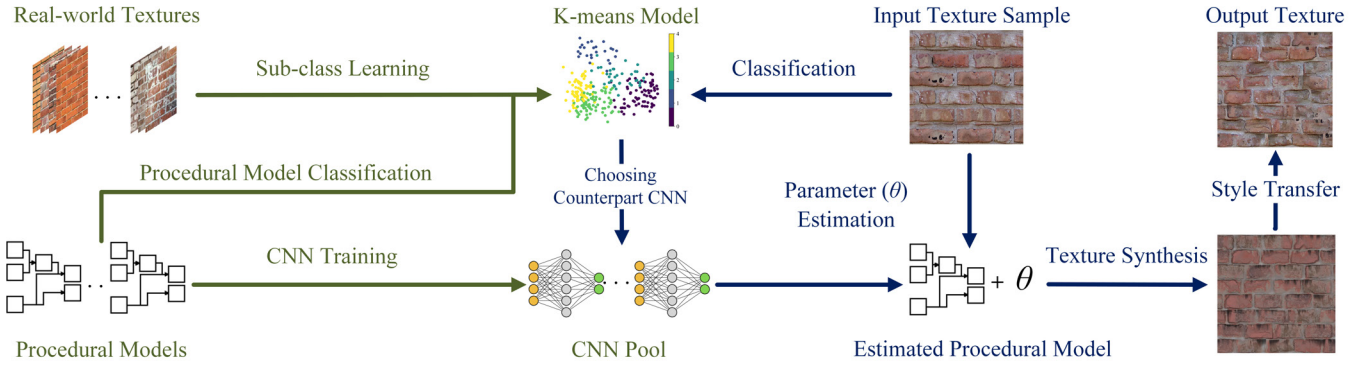


Fig. 3. Overview of our inverse modeling system. Our system starts by collecting real-world textures and corresponding procedural models. Real-world textures are used to learn style-based sub-classes while procedural models are classified into these sub-classes. A CNN pool is trained for all the procedural models to estimate parameters given a texture exemplar. The green lines show the training process of our system, while the blue lines are the evaluation process. During the evaluation phase, our system applies K-means model to select appropriate procedural models which share style similarity with given texture sample. We predict the parameters of the selected procedural models and generate procedural texture results. The final step can be an additional style transfer augmentation step.

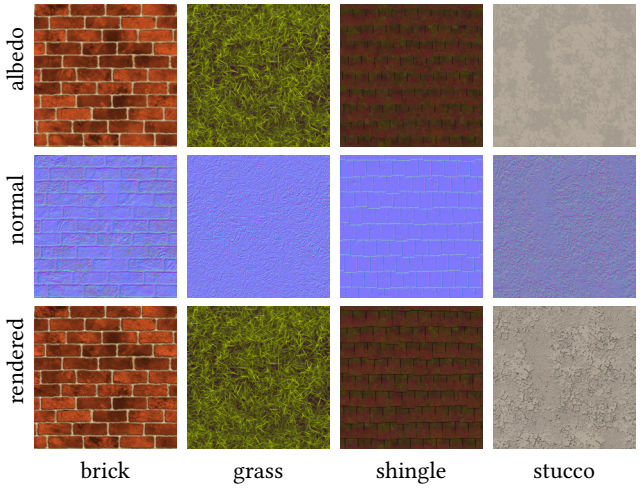


Fig. 4. Synthesized procedural texture samples including albedo maps, normal maps and rendered textures for the four texture classes in our database.

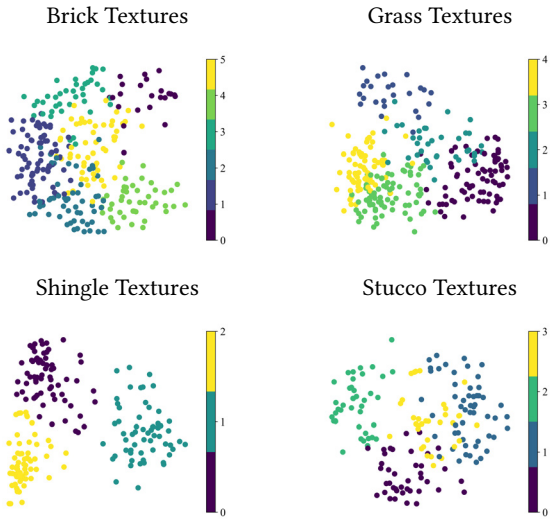


Fig. 5. Visualization of the distribution of style-based sub-classes for each of the high level texture class in our database. Visualization is achieved by using PCA to find a 2D embedding.

and procedural assets in given texture classes. In this paper, we build our experimental system based on the four kinds of textures–brick, grass, shingle and stucco–which are commonly used in architectural visualizations. These four classes contain both regular and randomized texture patterns with different scales. Such variety is suitable to demonstrate the potential to generalize our framework to generate procedural textures for other classes. We collect real-world textures by Google Images via keywords such as "brick texture". We gather procedural assets of these four texture classes from Allegorithmic Substance Share [Allegorithmic 2019b], a free exchange platform with abundant high-quality procedural texture models created by designers. The procedural texture models shared in this platform are in the form of node graphs e.g. Fig. (2) which can be edited by Allegorithmic Substance Designer [Allegorithmic 2019a]. To make our system more compact, we filter our collected procedural assets to choose a small set that produce diversified appearance. Five procedural assets are selected for brick textures, three for grass, five for shingles and four for stucco. Some procedural texture examples can be seem in Fig. (4).

## 3.2 Procedural Model Selection

Model selection is a necessary step in our framework. It would be very inefficient to use all the procedural texture models of a texture class to fit the given texture sample, especially when the system can be potentially extended to consist of dozens of procedural models per texture class. Each high level texture class includes a diverse range of styles, and not all styles can be produced by each procedural model. We want to select models that will reliably produce visually satisfactory results for the particular user input. To achieve this, we propose a procedural model selection method

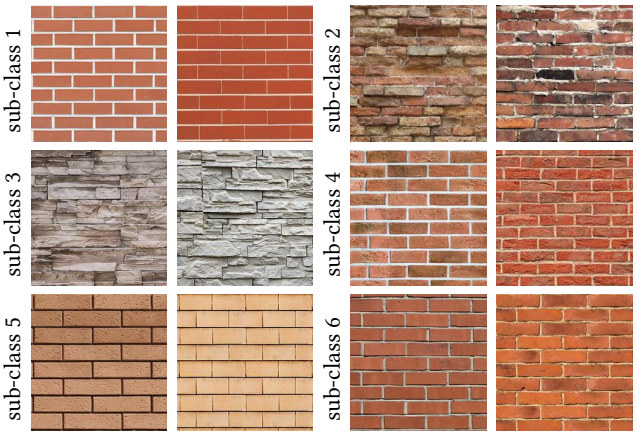


Fig. 6. Samples of 6 learned sub-classes based on style difference in the brick texture class. Clear style differences can be noticed among the exemplar textures in different sub-classes. Image credit: bottom row rightmost, *www.vectorstock.com*.

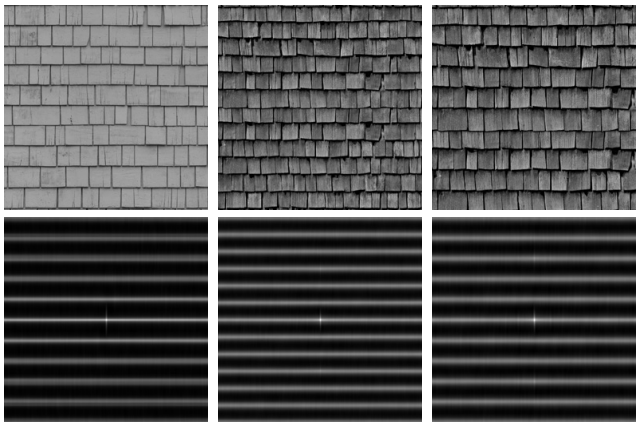


Fig. 7. An example for our auto-rescaling process. Grayscale images are used for better visualization. For each image in the dataset, we compute the normalized autocorrelation function (NACF) shown as the bottom row. We pick up an image sample (leftmost) as the baseline and rescale other images (middle) to the similar scale (rightmost) as the baseline. For this shingle texture case, we can measure the scale of the textures by computing the frequency of the horizontal bright lines in NACF maps. We then compute the relative scale factor for each shingle textures and rescale them by the scale ratio. Image credit: top row leftmost, *www.textures.com*; top row middle, *3docean.net*.

based on unsupervised clustering. Our model selection method will only choose the procedural models with the same style as the given texture sample. Specifically, our inverse modeling system performs unsupervised clustering on collected real-world textures to find the natural sub-classes in each high level class. Each sub-class is a type of style in the high level texture class.

Recently, many works analyze the style representation of an image using deep learning based methods. Gatys et al. [2016] measured the style similarity between two images by comparing the differences between the Gram matrices of two images. Li et al. [2017b] argued that the style discrepancy between two images can be measured as the difference between their feature distributions. Inspired by their ideas, we apply a pre-trained VGG19 network [Simonyan and Zisserman 2014] to extract deep feature maps from our collected real-world images. Instead of using Gram matrices, we define the style representation by computing the histogram on each layer of the feature maps and concatenate the histogram results together to construct our style representation. For instance, with deep feature maps with N layers, we compute the histogram with M bins for each layer, and can achieve a NxM size feature vector to represent style. Compared with the Gram matrix representation, our style definition has lower dimension which is more reliable for an unsupervised clustering algorithm. In practice, we extract the deep feature maps from Relu5_1 layer from VGG19 network and compute the histogram with the bin size of 30. The final style feature is a 15360 dimension vector.

With these style features, we use Principle Component Analysis (PCA) to reduce the redundant information to stabilize unsupervised clustering and then run a K-means algorithm to get sub-classes based on their style difference. We determine the number of clusters $K$ via the gap statistic [Tibshirani et al. 2001]. We enumerate $K$ in the range of 3 to 10 since too many clusters are unwanted, and finally we choose $K = 6$ for brick textures, $K = 5$ for grass textures, $K = 3$ for shingle textures and $K = 4$ for stucco textures in our experiment. The details of K selection are given in the supplemental document. The final clustering results can be visualized in Fig. (5). Style differences can also be clearly noticed visually such as among brick texture samples in Fig. (6). Having acquired the sub-classes of one texture class, our system can then sample the procedural models belonging to that class and apply the trained K-means model to classify them into different sub-classes. In our experiments, our system samples 50 textures for each of our procedural models and applies the K-means model to classify them into different clusters. It is reasonable that some procedural models produce results in multiple sub-classes. An average 3~5 of total 5 procedural brick models are clustered into a learned sub-class, while 1~4 of 5 for procedural shingle models, 2~3 of 3 for procedural grass models and 2~4 of 4 for procedural stucco. An interesting observation is that for grass textures, all of our 3 procedural grass models are classified into 3 of the 5 sub-classes, while two other sub-classes do not contain any procedural model. This is an important indication that additional procedural models are needed to cover the full range of naturally occurring grass textures (see Sec. 3.5).

Since we attempt to re-purpose the VGG19 network (which was pre-trained on the object recognition task) to compute style features for our task, a potential problem is that the scale information is embedded in the extracted features which will bias our unsupervised clustering results towards scale differences rather than style differences. We solve this problem by automatically normalizing our collected real-world images into the same scale. We compute the normalized autocorrelation function (NACF) similar to [Pintus et al. 2015] to identify the scale of the natural textures. We pick a sample from our collected real-world textures and compute its NACF as the baseline and rescale other real-world textures to the same scale. Auto-rescaling processes are applied on all of our textures except stucco textures. Fig. (7) shows one auto-rescaling example

on shingle textures. See the supplemental document for other kinds of textures.

### 3.3 Learning Inverse Mapping With CNN

For each procedural model in our database, we need to estimate their parameters when given a texture example. Formally, a node graph $g$ can be defined as a function $g(\theta)$ where $\theta$ is the parameter vector, the output of this function will be SVBRDF maps. Hence, this inverse problem can be posed as an optimization problem given an example texture $I$

$$\theta^* = \arg\min_{\theta} d(I, R(g(\theta))) \tag{1}$$

where $d(\cdot,\cdot)$ is a similarity metric and $R$ is a rendering operator. Directly solving this problem by traditional gradient-based optimization algorithms, may not yield a satisfactory result (see Fig. (15)) because the node graph $g$ can be non-continuous. The analytical gradient is inaccessible, and for many situations, the detailed implementation of node graph $g$ is unknown (e.g. an unknown implementation in a commercial software). Early works [Dischler et al. 2002; Lefebvre and Poulin 2000] circumvent this ill-posed optimization problem by image analysis on given texture samples using manually designed criterion. Their approaches are only suitable for simple procedural models. In this section, we introduce a detailed implementation of our CNN-based method to learn an inverse mapping from given texture sample to the appropriate parameter vector of a pre-defined node graph $g$. The major elements are data generation, network structures and training process.

*3.3.1 Data Generation.* The dataset can be generated by the procedural texture model $g$ itself. We can synthesize the whole training and validation dataset by randomly sampling the parameter space and use the node graph itself to generate corresponding albedo maps and normal maps. While in most case, the structure information in the albedo maps is enough to represent the texture's main patterns, designers may prefer to add more detailed structural information to the normal maps (see Fig. (4)), such as grooves or scratches. Based on this observation, the albedo map and normal map should be rendered as a natural texture to combine the structural information in both of the maps.

While many works on SVBRDF acquisition suggested using environment illumination (Image-based lighting) [Li et al. 2017a, 2018a] or global illumination [Li et al. 2018b] to synthesize images in their data generation process, we propose rendering the albedo map and normal map with direct illumination to capture the structural information in the textures, making the rendering process much faster. Besides, even if the node graph can generate a roughness map and metallic map, in our model we do not make a specific assumption about lighting conditions such as the use of a flash [Deschaintre et al. 2018; Li et al. 2018a,b], so we have an under-defined problem for inferring the roughness or metallic values. Hence, we omit the roughness and metallic maps.

The rendering process is accomplished using Blender [Blender 2019a] where we render a plane textured with the albedo map and normal map using direct sun light. We use a diffuse reflectance model to render the plane with a perspective camera positioned so that the view vector in the direction of center is perpendicular to the plane. Fig. (4) provides a few examples of our synthesized texture data including albedo maps, normal maps and rendered textures.

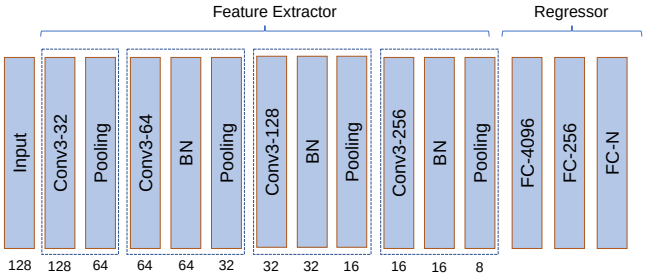


Fig. 8. Our network structure consists of four basic network blocks as feature extractor followed by three fully connection layers for parameter regression. The number below each layer represents spatial resolution. Each basic network block is composed by a convolutional layer, a batch normalization layer (except the first block) with subsequently a Leaky Relu activation unit and a pooling layer which can be either average or max pooling. For each convolutional layer Conv3-X, we set stride=1, kernel size=3, and number of output features = X. For each fully connection layer FC-X, we set number of output features = X. We insert a dropout layer after each of the first two fully connection layer. We add a tanh layer after the last fully connection layer to normalize the output between -1 and 1. The output of the final layer is the predicted N parameters.

*3.3.2 Network Structure.* As shown as Fig. (8), our CNN structure follows the classic network structure AlexNet [Krizhevsky et al. 2012] and is extended to fit our parameter regression task. The input texture is a 3-channel RGB image with the spatial resolution of 128x128. The feature extraction part of our network is built by four basic network blocks followed by a parameter regression part which is built by three fully connection layers. The last fully connection layer predicts normalized parameters of the procedural model. Please refer to the comments in Fig. (8) for detailed settings.

Transfer learning [Torrey and Shavlik 2010] can be a possible choice to train our model based on pre-trained neural networks such as VGG19 [Simonyan and Zisserman 2014] to avoid training from scratch. Our experiment results show though that re-targeting pre-trained deep neural networks to our parameter prediction task does not give better results.

*3.3.3 Training Process.* Since our data are all synthesized by procedural texture model itself, the training can be modeled as an online training process. When the training result is unsatisfactory, we can sample more data points to further reduce the loss. The training process proceeds until the loss is reduced to a small threshold. We augment the training data by adding Gaussian noise to input textures. To stabilize and accelerate the training process, we pre-process the parameter data and texture data by normalizing them between -1 and 1, and the output of CNN is also normalized within -1 and 1 by a tanh layer. We use Adam Optimizer with a 0.0001 learning rate. The training is regularized by weight decaying of 0.0005 ($L_2$ regularization). We adapt weighted Mean Squared Error (MSE) loss as our loss function. From our initial experimental results, we found sampling 80000 data points is sufficient to reduce the error to a reasonable threshold (0.15) for most of the procedural models. Hence,

for remained procedural models, we directly sample enough data points in order to accelerate the training process.

### 3.4 Evaluation And Style Transfer Augmentation

The evaluation step is straightforward. The user will provide a texture sample to our system. Our system will first rescale the input sample using auto-rescaling process (Fig. (7)) and compute style features of the rescaled texture sample and classify it into a sub-class by the pre-trained K-means model. The system will select those procedural texture models which are also classified into this sub-class, and the pre-trained CNNs of the chosen models will be used to predict parameters. The system will finally generate procedural textures for the user via the chosen procedural textures models and their predicted parameters.

Ideally, our predicted procedural textures will match the overall visual appearance of the given image samples, but a visual gap between synthetic procedural textures and real-world images may still exist. We observe that rich spatial details in naturally captured images often cannot be sufficiently modeled procedurally. To close this gap, we introduce an optional neural style transfer step [Gatys et al. 2016]. We treat our generated procedural texture as the content image and the given texture sample as the style image. Similar to [Gatys et al. 2016], we run optimization algorithm on our procedural texture to make its style gradually match the texture sample. Several experimental results (see Fig. (12), (13), (16) and (17)) show this image augmentation can make our generated procedural textures become more realistic and close to naturally captured textures. One drawback of this optional step is the loss of editability because it enforces a transformation from the graph representation back to pixel domain. Another concern is the degeneration of the quality of transferred results when users gradually edit procedural textures away from initial prediction (see Sec. (4.5) for more discussion). Though optimization of style can take a few minutes, the state-of-the-art algorithms such as [Huang and Belongie 2017] may accelerate this procedure into real-time with pre-trained neural network.

### 3.5 Scalability

It is worth noticing that performing style-based clustering on real-world textures can improve the scalability of our framework, since we can add new procedural textures to existing classes of textures guided by style-based clustering results. For instance, we can visualize the relationship between real-world textures and procedural textures by projecting the style features of procedural textures into the embedding space of real-world textures. Fig. (9) illustrates a visualization for procedural grass textures. We can see our three procedural models only cover three sub-classes of real-world grass textures which indicates that we would need to add new procedural models to our system to cover the all of the texture sub-classes (sub-class 1 and sub-class 3). New procedural models should have the similar visual appearance as the real-world images in the uncovered sub-classes. Further, our system can warn the user when their example falls into an uncovered sub-class that the results will be poor, and that additional models need to be collected or created to improve the results.

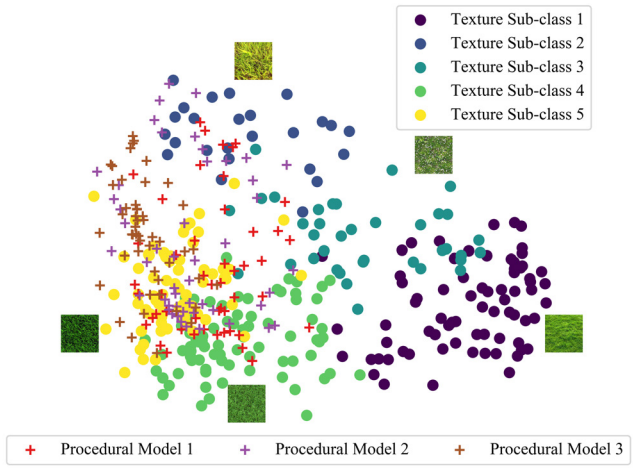


Fig. 9. Visualization of 2D embeddings of procedural grass textures. Labels "o" with different colors represent real-world textures which are classified into five sub-classes, while labels "+" with different colors indicate different procedural models.

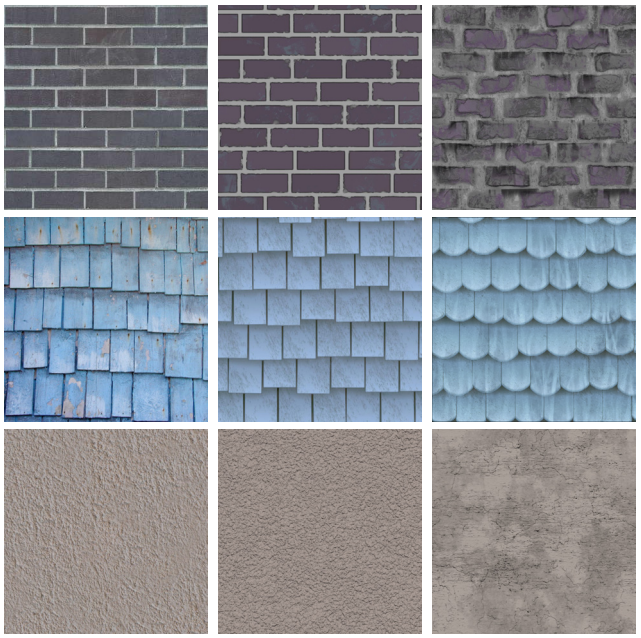


Fig. 10. Model selection. Our system will only select models which have a style similar to the sample texture, and will only estimate parameters for the chosen models. The prediction results from those unselected models do not achieve visually similar results because of style differences. Image credit: top row leftmost, *www.textures.com*.

## 4 EXPERIMENTAL RESULTS

We demonstrate our framework with a system that includes four texture classes – brick, grass, shingle and stucco. For each procedural model, we sample 80000 procedural textures and separate them into a training set and a validation set where 75000 textures are used

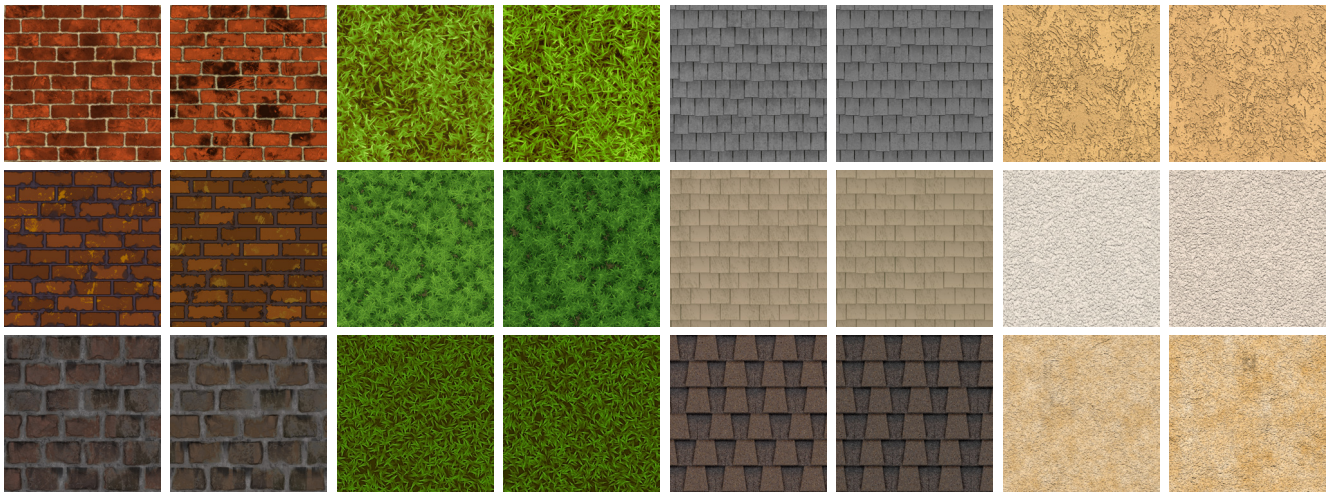


Fig. 11. Validation on trained CNN models. We show the prediction results from given input textures using our CNN models. Each input texture is sampled from the validation sets of our procedural texture models. Results show our predicted procedural textures can achieve good visual similarity with the input texture.

for training and 5000 textures for validation. The data generation process per procedural model takes about 6 to 8 hours on a PC with Intel i7-5820k 3.30Hz. Our CNN is implemented by Pytorch and trained on Nvidia GTX 980M 4GB GPU. For each CNN model, we train 24 epochs with initial learning rate of 0.0001 and after every 8 epochs, the learning rate will decay by a factor of 10. The overall training time per CNN takes 3 hours. For most of our procedural models, the validation loss will stop decreasing after 15 epochs. The pre-computation time cost for training our system can be significantly decreased by using more powerful CPUs and GPUs. When node graphs and parameters are chosen, the time cost for generating procedural textures can vary among different procedural models due to the difference in their complexity and implementations. High resolution textures can be generated fast enough for interactive editing and texturing. On average, 4K textures can be generated in around 160ms while 8K textures can be generated in about 1.5s. All the procedural textures presented in this paper are 1024x1024 which takes less than 50ms.

### 4.1 Validation on trained CNN models.

We first verify the performance of our trained CNN models. The final numerical errors, weighted MSEs are less than 0.15 for all procedural models and the minimum MSE loss can be less than 0.005. Our model achieves satisfactory visual approximations. Fig. (11) shows some procedural texture samples in our validation sets and our predicted procedural textures using CNN. Different texture samples demonstrate that our trained CNN model is capable of inferring reasonable parameters to reproduce visual appearance similar to input procedural textures. Note that our predicted results are not exactly the same as the given input exemplar since (1) errors exist in our predicted parameters (2) and randomness in the procedural texture generation process effects the distribution of many random patterns such as the dirt and paint effects on brick texture samples.

### 4.2 Validation on Inverse Modeling System.

We then demonstrate our inverse modeling system can select the appropriate procedural models to predict parameters for given input real-world texture sample. The effects of our model selection using sub-classes can be seen as Fig. (10). Predicted textures from models selected by our system give visually appealing results, while synthesized textures from unselected models are not visually similar to the input. For instance, when given input is a concrete-like brick texture, fitting the texture with a stone-style procedural model is impossible. Similarly, given a rectangularly shaped shingle texture, a rounded shaped procedural shingle model cannot fit it well. Style differences are also salient in non-structural textures such as stucco textures. Fig. (12) and Fig. (13) demonstrate our system can generalize very well to given real-world texture samples by using model selection to chose the appropriate model to predict the parameters. Various styles of textures can be approximated via procedural models. All the rendered textures here are lit by direct sunlight as diffuse surfaces. However, our system generates SVBRDF maps for users, so high-quality rendering can be achieved if using global illumination and sophisticated reflection models.

Figures (12)(13) show that the last optional style transfer augmentation step, though causing loss of the editability of procedural modeling, can further improve the visual quality and add detail information to the generated procedural textures. Style transfer can also help produce extended photo-realistic textures e.g. Fig. (17).

### 4.3 Comparison with Optimization-based Method

For comparisons, we know of no previous method that deals with parameter prediction for large-scale procedural texture models. We compare our CNN-based method with optimization-based methods which aim to solve Eq. (1) directly. Results (Fig. (15)) show that our CNN-based method can outperform both gradient-based (L-BFGS-B) or gradient-free (Powell) optimization methods in term of visual

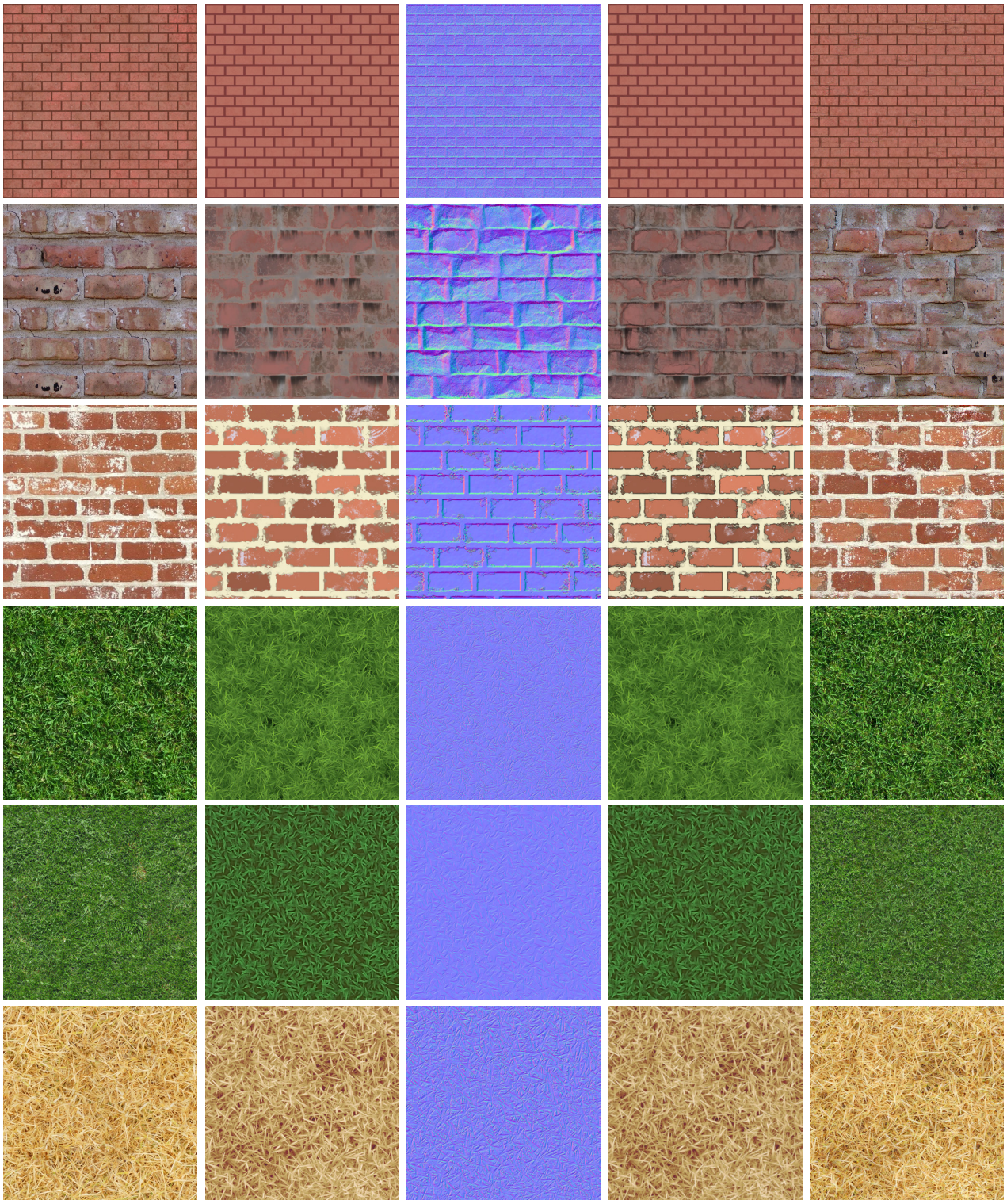


Fig. 12. Validation on inverse modeling system. Given an input real-world natural image sample, our system can chose proper procedural models to predict parameters and generate albedo maps and normal maps. Results show that our rendered texture can have high visual similarity with natural input textures. The final style-transfer augmentation step can further improve the quality of the procedural texture. Examples include brick textures and grass textures. Image credit: top row leftmost, *share.substance3d.com*, CC BY 4 license; fourth row leftmost, *seamless-pixels.blogspot.com*; fifth row leftmost, *creativemarket.com*; bottom row leftmost, *depositphotos.com*.

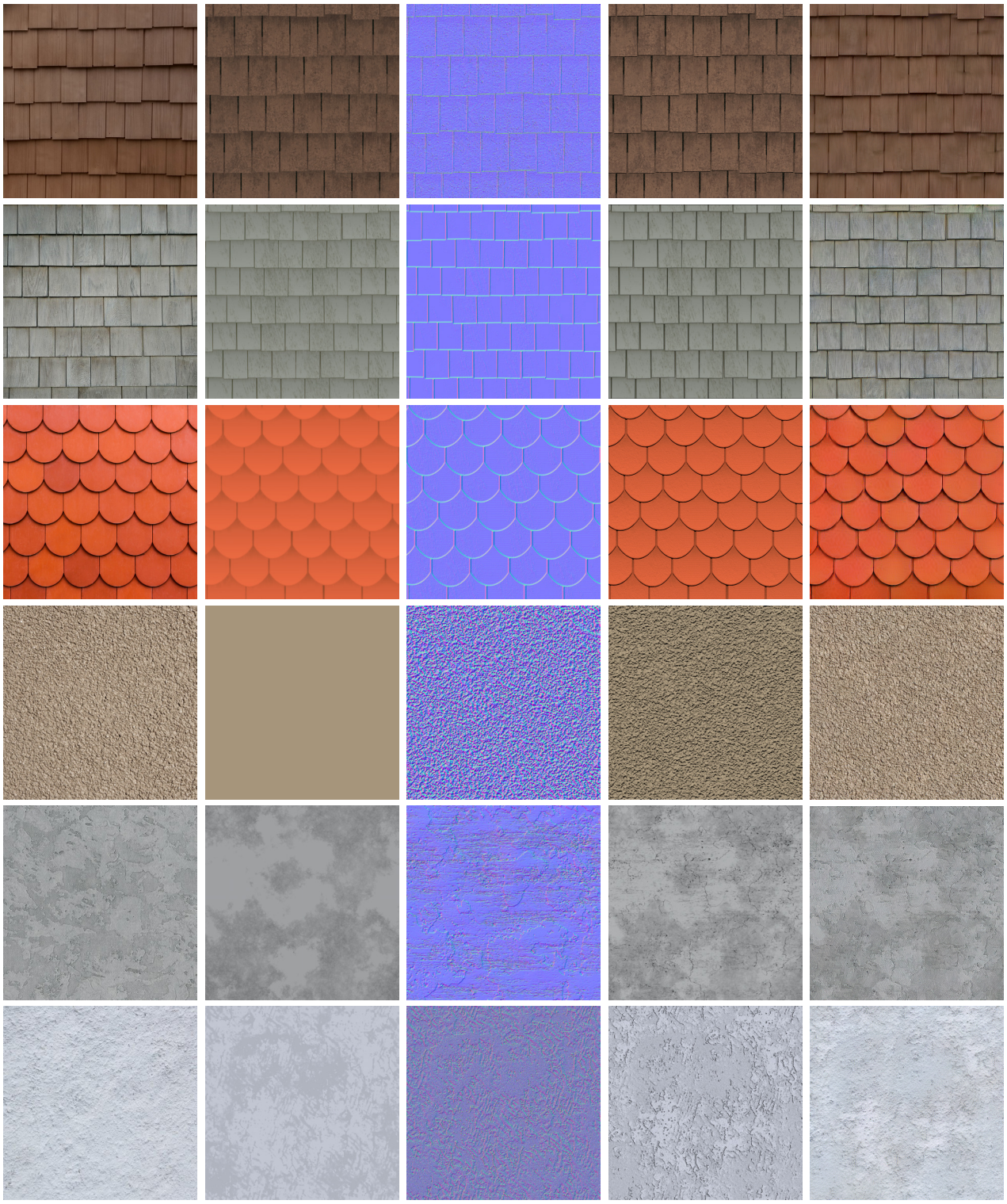


Fig. 13. Validation on inverse modeling system (Cont.). Examples include shingle textures and stucco textures. Image credit: third row leftmost, *depositphotos.com*.

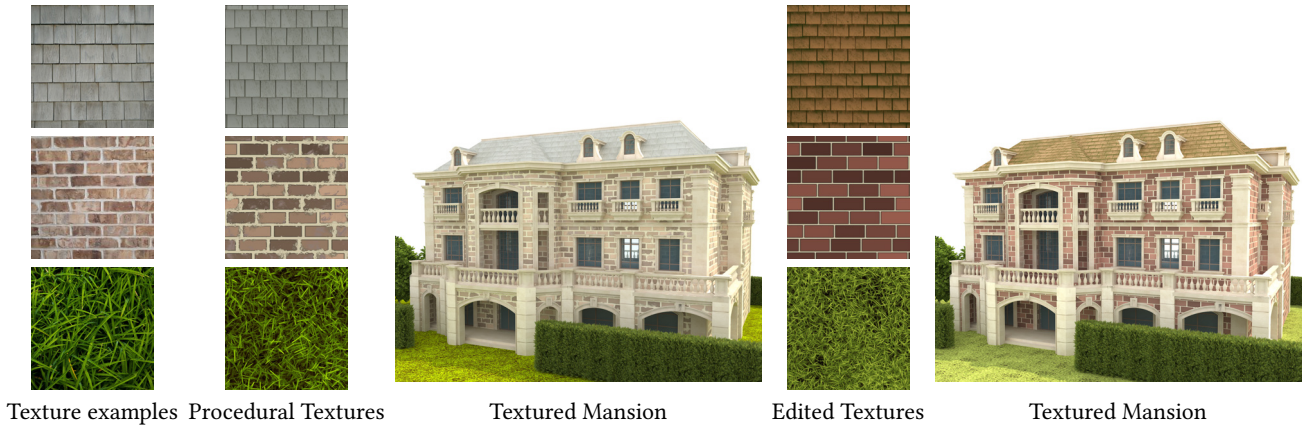


Fig. 14. Texturing large-scale scenes in high-resolution. Procedural textures provide a powerful tool to texture complex scene with high-resolution non-repetitive textures. Given exemplar textures, our system converts them into procedural textures and textures the mansion model. Users can also tune the parameters and achieve fast visual feedback using the procedural models.

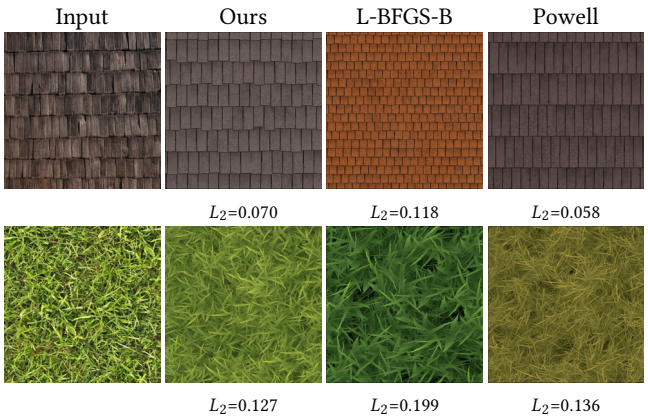


Fig. 15. Comparison with optimization-based methods for parameter estimation. Our CNN-based method can outperform both traditional optimization-based methods in visual similarity of structure and hue. Average $L_2$ error is provided below each fitted result. Image credit: second row leftmost, *seamless-pixels.blogspot.com*.

similarity. Both optimization methods are implemented using SciPy [Jones et al. 2001] package and we set maximum iteration number as 1000. Since the gradient of procedural model $g$ is unreliable, gradient-based approach cannot estimate appropriate parameters in both cases, and it is trapped into local minimum after 50~100 iterations. The Powell method can achieve plausible results with a smaller average $L_2$ error than ours in the shingle case, but it still fails to recover structures of the shingle exemplar and also estimates incorrect hue and density values for the grass texture sample. The Powell method takes more than 10 minutes to converge while our CNN prediction takes only microseconds.

## 4.4 Applications

Converting a pixel-based texture into a procedural texture has many applications other than simply being treated as a mathematical problem. Procedural modeling provides a powerful tool to control

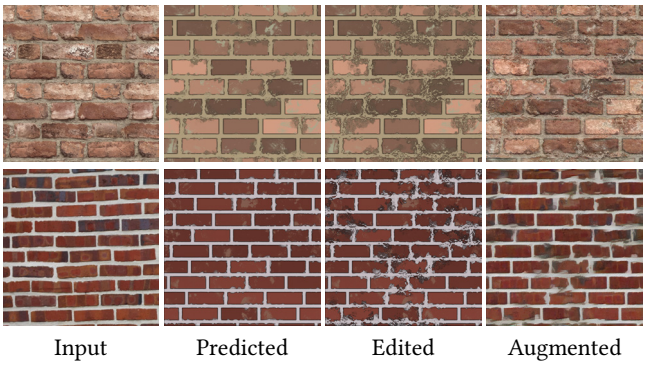


Fig. 16. Procedural texture editing. By converting given input texture sample to a procedural texture using our system, editing becomes convenient and users can only change few parameters to alter texture appearance such as damaging more edges. The edited image can be further augmented by a style-transfer process.

Fig. 17. High-resolution Texture Expansion. Given an input texture (lower right corner), we use our system to generate a procedural counterpart. With this procedural model, we can procedurally expand its size and generate randomized patterns. Finally, we transfer it back to pixel representation to achieve a high-resolution photo-realistic texture (2048x1024). Image credit: inset, *share.substance3d.com*, CC BY 4 license.

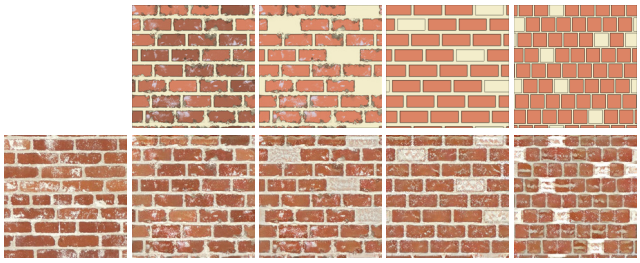


Fig. 18. Applicability of Style Transfer Augmentation. The upper row shows procedural textures and the lower row shows their corresponding style-transferred results using the input as style image. We show the quality of style-transferred gradually degrade when the procedural models are modified away from the original predicted results. From predicted results to Edited(1), we remove the color variation among bricks and delete several bricks procedurally. From Edited(1) to Edited(2), we clear the painting and restore damaged edges. From Edited(2) to Edited(3), we change the overall structure of the brick wall.

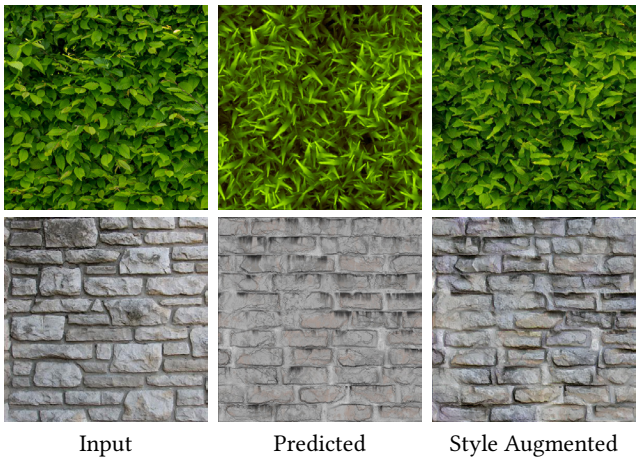


Fig. 19. Failure cases. Upper row shows an example of reconstructing leaf textures by procedural grass, and lower row shows a case of using our procedural brick models to estimate irregular-structured bricks.

the appearance of the textures and allows textures to be extended to arbitrary resolution. Fig. (17) shows an example of high-resolution texture expansion.

*4.4.1 Procedural Texture Editing.* Texture appearance can be edited procedurally. Seen as Fig. (16), we can utilize our inverse modeling system to convert the given texture into procedural representation and then change the appearance of the procedural texture only by tuning a few parameters or directly changing the node graph without requiring manually editing the image pixel-by-pixel. When the user wants to apply small changes to the texture appearance, they can also apply style-transfer augmentation to transfer the procedural textures back to pixel-based representation to improve the quality of the final result. The applicability of style transfer in significant appearance changes will be discussed in Sec. (4.5)

*4.4.2 Fast Scene Texturing in High-resolution.* Procedural textures enables users to texture large-scale 3D scenes in high resolution. With the predicted procedural model, designers can be free from manually tiling small texture patches and struggling with low resolution and repetitive texture patterns. They can either bake high-resolution e.g. 4K or 8K procedural textures to texture their scenes or directly evaluating pixel values procedurally on texture coordinates. Fig. (1) and (14) demonstrate this effective and efficient texturing process. The user can also have quick visual feedback when editing the appearance of the textures by altering the parameters of procedural models.

### 4.5 Applicability of Style Transfer Augmentation

The last optional style transfer step in our framework is an efficient way to bridge the gap between predicted procedural texture images and real-world textures with highly stylized details e.g. Figs. (12)(13)(17). The style transfer works well on slightly changed procedural textures, but significant editing can modify the procedural textures far from original input texture samples, making the input sample no longer an appropriate style image target. As shown in Fig. (18), after obtaining the procedural texture, we gradually edit the parameters of the procedural model to simplify the texture appearance. The style-transferred result in Edited(1) achieves good visual quality since minor changes are applied. Style transfer approach fails in Edited(2) and Edited(3). For instance, we actually want to eliminate white paint in Edited(2), but the transferred result has unwanted white noises. When we significantly change the structure of the brick texture, the style transfer produces incorrect results (Edited(3)) because the overall appearance of the procedural texture is very different from the original input.

## 5 LIMITATIONS AND FUTURE WORK

In the previous section, we have demonstrated that our inverse modeling framework can provide high-quality procedural textures to users by example. Our framework can be extended to arbitrary numbers of classes by collecting additional natural images and user-shared procedural models. However, challenges still exist.

First, our system requires each procedural texture model to train a counterpart CNN to regress the parameters. Too many pre-trained neural network will occupy a large amount of storage. Training one single neural network that can simultaneously choose the procedural model and predict parameters for all the procedural models is an appealing solution but may degrade the quality of prediction results.

Second, our system relies on training multiple neural networks in the training process requiring a lot of computation both in the data generation phase and in the back-propagation phase which are about 9~11 hours per procedural model. A possible solution is a more sophisticated sampling approach for efficient parameters space sampling. In our implementation, we randomly sample the parameter space and generate the SVBRDF maps. However, procedural textures generated by randomly sampled parameters may look unnatural. Only some parameter combinations can produce visually satisfactory textures. Mathematically, unnatural textures will not effect the solution of a parameter regression problem, but a refined sampling strategy could accelerate the training process.

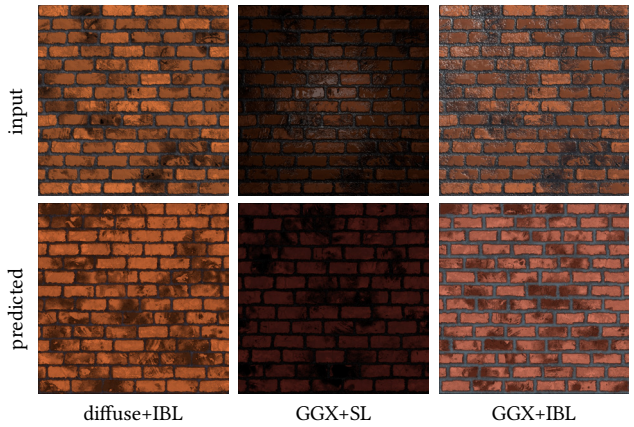


Fig. 20. Variation in Illumination Conditions. We test our our model's sensitivity to illumination variations by synthetic data. Input textures are rendered under different reflection models and lighting conditions. IBL refers to Image-Based Lighting and SL refers to Spherical Lighting. We use our CNN to estimate parameters for these inputs and render the predicted albedo map and normal map using diffuse model and direct illumination, which is the same as our data generation process. Our estimation is reasonable for the first case but cannot reproduce high-frequency lighting in later two cases.

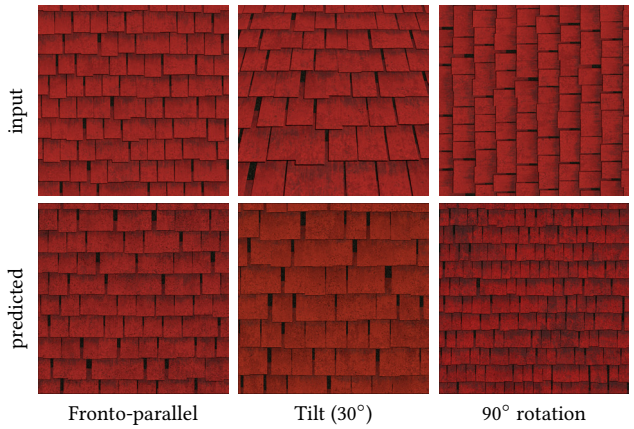


Fig. 21. Variation in Orientations. We validate our CNN model on synthetic textures without rectification. Input textures are posed in different orientations such as fronto-parallel, tilt (30°) or 90° rotated.

Third, additional impact factors such as lighting conditions and orientations are worth considering in our framework to further improve the robustness of our model. For instance, since our training data are rendered using a diffuse BRDF model with direct illumination, high-frequency effects may not be modeled reasonably in our framework. As seen in Fig. (20), when testing on a texture rendered under a diffuse model with environment maps (not direct sunlight), our model can correctly generate visually similar textures. While testing on textures rendered under a specular BRDF model (GGX) which yield high-frequency variations, our model estimates incorrect albedo values because the diffuse model cannot reproduce specular highlights. Nevertheless, the overall structures of estimated textures still match. Also, Fig. (21) shows our CNN prediction perform robustly with a slightly tilted view (30°) but it cannot handle larger rotations. A promising solution is to consider a complex shading model and illumination in our framework, as well as to expose more parameters in the procedural models e.g. rotation or tilt to simulate possible situations in the real world.

Finally, since our system is driven by a collection of real-world textures and procedural assets, the entire texture space that our system can cover is limited to classes in the system's database. Given an arbitrary unseen class of textures, our framework cannot provide a correct procedural model. Fig. (19) shows two representative failure cases where given images are not considered in our prototype system. Since the shape of leaves and unique structure of the given brick sample are not covered in our collected procedural models, our system fails to provide matched visual appearance. The style transfer step can eliminate some defects but still suffer from incorrect shape and structure. The major problem is that our system lacks the ability to infer the diversified internal structure of a node graph from an arbitrary texture sample, and thus cannot take advantage of the powerful representation ability of the node graph itself. We rely on collecting a wide variety of existing node graphs. A grand challenge is to make full use of the possible combinations of nodes and powerful structures in the node graph to automatically construct (rather than select) a node graph given a texture sample.

## 6 CONCLUSION

We present a novel framework for inverse procedural texture modeling by example. We design a data-driven inverse modeling system based on a collection of real-world textures and procedural texture assets. The user provides a texture sample to our system, and our system generates visually similar high-quality procedural textures. The core idea is to train an unsupervised clustering model to perform procedural model selection and train a CNN pool to learn an inverse mapping from image space to parameter space for each procedural models in our database. Given a texture sample, our system will first identify appropriate node graphs in the database which can best match the exemplary visual appearance, and then predict parameters with pre-trained CNNs. With predicted parameters and selected procedural texture models, the user can obtain procedural textures that can approximate the texture example. A style transfer step can further improve the visual quality of fitted procedural textures. Experiments show our proposed framework can yield high-quality procedural textures and allow texture editing and fast texturing. We hope our work can inspire more studies on procedural representation of textures using new methods.

## ACKNOWLEDGMENTS

We would like to thank all the reviewers for their valuable comments. We also thank Benedict Brown, Ezra Davis, Sherry Qiu, Weiqi Shi and Zeyu Wang for their thoughtful suggestions on our work. This work was supported in part by NSF grant IIS-1747522. Finally, we thank all the contributors of Allegorithmic Substance Share.

## REFERENCES

Miika Aittala, Timo Aila, and Jaakko Lehtinen. 2016. Reflectance Modeling by Neural Texture Synthesis. *ACM Trans. Graph.* 35, 4, Article 65 (July 2016), 13 pages. https:

//doi.org/10.1145/2897824.2925917
Allegorithmic. 2019a. Substance Designer. https://www.allegorithmic.com/products/substance-designer
Allegorithmic. 2019b. Substance Share. https://share.allegorithmic.com
Blender. 2019a. https://www.blender.org/
Blender. 2019b. Blender Materials. https://matrep.parastudios.de/
Valentin Deschaintre, Miika Aittala, Fredo Durand, George Drettakis, and Adrien Bousseau. 2018. Single-image SVBRDF Capture with a Rendering-aware Deep Network. *ACM Trans. Graph.* 37, 4, Article 128 (July 2018), 15 pages. https://doi.org/10.1145/3197517.3201378
J-M Dischler, Karl Maritaud, Bruno Lévy, and Djamchid Ghazanfarpour. 2002. Texture particles. In *Computer Graphics Forum*, Vol. 21. Wiley Online Library, 401–410.
Alexei A. Efros and William T Freeman. 2001. Image quilting for texture synthesis and transfer. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. ACM, 341–346.
Alexei A. Efros and Thomas K. Leung. 1999. Texture Synthesis by Non-Parametric Sampling. In *Proceedings of the International Conference on Computer Vision-Volume 2 - Volume 2 (ICCV '99)*. IEEE Computer Society, Washington, DC, USA, 1033–. http://dl.acm.org/citation.cfm?id=850924.851569
Bruno Galerne, Ares Lagae, Sylvain Lefebvre, and George Drettakis. 2012. Gabor Noise by Example. *ACM Trans. Graph.* 31, 4, Article 73 (July 2012), 9 pages. https://doi.org/10.1145/2185520.2185569
Bruno Galerne, Arthur Leclaire, and Lionel Moisan. 2017. Texton noise. In *Computer Graphics Forum*, Vol. 36. Wiley Online Library, 205–218.
Leon Gatys, Alexander S Ecker, and Matthias Bethge. 2015. Texture synthesis using convolutional neural networks. In *Advances in neural information processing systems*. 262–270.
Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. 2016. Image style transfer using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2414–2423.
Guillaume Gilet, Basile Sauvage, Kenneth Vanhoey, Jean-Michel Dischler, and Djamchid Ghazanfarpour. 2014. Local random-phase noise for procedural texturing. *ACM Transactions on Graphics (TOG)* 33, 6 (2014), 195.
Eric Heitz and Fabrice Neyret. 2018. High-Performance By-Example Noise Using a Histogram-Preserving Blending Operator. *Proc. ACM Comput. Graph. Interact. Tech.* 1, 2, Article 31 (Aug. 2018), 25 pages. https://doi.org/10.1145/3233304
Xun Huang and Serge Belongie. 2017. Arbitrary style transfer in real-time with adaptive instance normalization. In *Proceedings of the IEEE International Conference on Computer Vision*. 1501–1510.
Eric Jones, Travis Oliphant, Pearu Peterson, et al. 2001. SciPy: Open source scientific tools for Python. http://www.scipy.org/ [Online; accessed <today>].
Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.
Vivek Kwatra, Irfan Essa, Aaron Bobick, and Nipun Kwatra. 2005. Texture optimization for example-based synthesis. In *ACM Transactions on Graphics (ToG)*, Vol. 24. ACM, 795–802.
Vivek Kwatra, Arno Schödl, Irfan Essa, Greg Turk, and Aaron Bobick. 2003. Graphcut textures: image and video synthesis using graph cuts. *ACM Transactions on Graphics (ToG)* 22, 3 (2003), 277–286.
Ares Lagae, Peter Vangorp, Toon Lenaerts, and Philip Dutré. 2010. Procedural Isotropic Stochastic Textures by Example. *Comput. Graph.* 34, 4 (Aug. 2010), 312–321. https://doi.org/10.1016/j.cag.2010.05.004
Laurent Lefebvre and Pierre Poulin. 2000. Analysis and synthesis of structural textures. In *Graphics Interface*, Vol. 2000. 77–86.
Xiao Li, Yue Dong, Pieter Peers, and Xin Tong. 2017a. Modeling Surface Appearance from a Single Photograph Using Self-augmented Convolutional Neural Networks. *ACM Trans. Graph.* 36, 4, Article 45 (July 2017), 11 pages. https://doi.org/10.1145/3072959.3073641
Yanghao Li, Naiyan Wang, Jiaying Liu, and Xiaodi Hou. 2017b. Demystifying Neural Style Transfer. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17)*. AAAI Press, 2230–2236. http://dl.acm.org/citation.cfm?id=3172077.3172198
Zhengqin Li, Kalyan Sunkavalli, and Manmohan Chandraker. 2018a. Materials for masses: SVBRDF acquisition with a single mobile phone image. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 72–87.
Zhengqin Li, Zexiang Xu, Ravi Ramamoorthi, Kalyan Sunkavalli, and Manmohan Chandraker. 2018b. Learning to Reconstruct Shape and Spatially-varying Reflectance from a Single Image. *ACM Trans. Graph.* 37, 6, Article 269 (Dec. 2018), 11 pages. https://doi.org/10.1145/3272127.3275055
Gen Nishida, Ignacio Garcia-Dorado, Daniel G. Aliaga, Bedrich Benes, and Adrien Bousseau. 2016. Interactive Sketching of Urban Procedural Models. *ACM Trans. Graph.* 35, 4, Article 130 (July 2016), 11 pages. https://doi.org/10.1145/2897824.2925951
Ruggero Pintus, Ying Yang, and Holly Rushmeier. 2015. ATHENA: Automatic text height extraction for the analysis of text lines in old handwritten manuscripts. *Journal on Computing and Cultural Heritage (JOCCH)* 8, 1 (2015), 1.
K. Simonyan and A. Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR* abs/1409.1556 (2014).
Robert Tibshirani, Guenther Walther, and Trevor Hastie. 2001. Estimating the number of clusters in a data set via the gap statistic. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 63, 2 (2001), 411–423.
Lisa Torrey and Jude Shavlik. 2010. Transfer learning. In *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*. IGI Global, 242–264.
Dmitry Ulyanov, Vadim Lebedev, Andrea Vedaldi, and Victor S Lempitsky. 2016. Texture Networks: Feed-forward Synthesis of Textures and Stylized Images.. In *ICML*, Vol. 1. 4.
Li-Yi Wei, Jianwei Han, Kun Zhou, Hujun Bao, Baining Guo, and Heung-Yeung Shum. 2008. Inverse Texture Synthesis. In *ACM SIGGRAPH 2008 Papers (SIGGRAPH '08)*. ACM, New York, NY, USA, Article 52, 9 pages. https://doi.org/10.1145/1399504.1360651
Li-Yi Wei, Sylvain Lefebvre, Vivek Kwatra, and Greg Turk. 2009. State of the art in example-based texture synthesis. In *Eurographics 2009, State of the Art Report, EG-STAR*. Eurographics Association, 93–117.
Li-Yi Wei and Marc Levoy. 2000. Fast Texture Synthesis Using Tree-structured Vector Quantization. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '00)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 479–488. https://doi.org/10.1145/344779.345009
Yang Zhou, Zhen Zhu, Xiang Bai, Dani Lischinski, Daniel Cohen-Or, and Hui Huang. 2018. Non-stationary Texture Synthesis by Adversarial Expansion. *ACM Trans. Graph.* 37, 4, Article 49 (July 2018), 13 pages. https://doi.org/10.1145/3197517.3201285