On Demand Solid Texture Synthesis Using Deep 3D Networks
Jorge Gutierrez, Julien Rabin, Bruno Galerne, Thomas Hurtut

To cite this version:
Jorge Gutierrez, Julien Rabin, Bruno Galerne, Thomas Hurtut. On Demand Solid Texture Synthesis Using Deep 3D Networks. 2018. hal-01678122v2

HAL Id: hal-01678122
https://hal.archives-ouvertes.fr/hal-01678122v2

Submitted on 30 Dec 2018 (v2), last revised 20 Dec 2019 (v3)
On Demand Solid Texture Synthesis Using Deep 3D Networks

J. Gutierrez\textsuperscript{1}  J. Rabin\textsuperscript{2}  B. Galerne\textsuperscript{3}  T. Hurtut\textsuperscript{1}

\textsuperscript{1}Polytechnique Montréal, Canada  
\textsuperscript{2}Normandie Univ. UniCaen, ENSICAEN, CNRS, GREYC, France  
\textsuperscript{3}Institut Denis Poisson, Université d’Orléans, Université de Tours, CNRS, France

Abstract
This paper describes a novel approach for on demand volumetric texture synthesis based on a deep learning framework that allows for generation of high quality 3D data at interactive rates. Based on a few example images of textures, a generative network is trained to synthesize coherent solid textures that reproduce the visual characteristics of the examples along some directions. To cope with memory limitations and computation complexity that are inherent to both high resolution and 3D processing on GPU, only 2D textures referred to as “slices” are generated during the training stage. These synthetic textures are compared to exemplar images via a perceptual loss function based on a pre-trained deep network. The proposed network is very light (less than 100k parameters), therefore it only requires sustainable training (i.e. few hours) and is capable of very fast generation (around a second for $256^3$ voxels) on a single GPU. The synthesized volumes have good visual results that are at least equivalent to the state-of-the-art patch based approaches. They are naturally seamlessly tileable and can be fully generated in parallel.

Keywords: Solid texture; On demand texture synthesis; Generative networks; Deep learning;

1. Introduction

2D Textures are ubiquitous in 3D graphics applications. Their visual complexity combined with a widespread availability allow the enrichment of 3D digital objects’ appearance at a low cost. In that regard, solid textures, which are the 3D equivalent of stationary raster images, offer several visual quality advantages over their 2D counterparts. Solid textures eliminate the need for a surface parametrization and its accompanying visual artifacts. They produce the feeling that the object was carved from the texture material. Additionally, the availability of consistent volumetric color information allows for interactive manipulation and fragmentation of the texturized models. However, unlike scanning a 2D image, digitization of volumetric color information is impractical. Hence a substantial quantity of the existing solid textures is synthetic.

One early way to generate solid textures is by procedural generation [Pea85,Per85]. In procedural methods the color of a texture at a given point only depends on its coordinates. This allows for a localized evaluation to generate only the required portions of texture at a given moment. We refer to this characteristic as \textit{on demand} evaluation, and in our case it is critical because storing the values of a whole solid texture at a useful resolution is prohibitive for most applications. Procedural methods are indeed fast and memory efficient. Unfortunately finding the right parameters of a procedural model to synthesize a given texture requires a high amount of expertise and trial and error. Photo-realistic textures with visible elemental patterns are particularly hard to generate by these methods.

In order to give up the process of empirically tuning the model for a given texture, several \textit{by-example} solid texture synthesis methods have been proposed [HB95,KFCO∗07,DLTD08,CW10]. These methods are able to generate solid textures that share the visual characteristics of a given \textit{target} 2D texture example through all the cross-sections in a given set of slicing directions (inferring the 3D structure given the constrained directions). Although they do not always deliver perfect results, by-example methods can be used to approximate the characteristics of a broad set 2D textures. One convenient approach to synthesize textures by-example is called \textit{lazy synthesis} [DLTD08,ZDL∗11]. It consists of an \textit{on demand} synthesis i.e. synthesizing only voxels at a specific location in contrast to generating a whole volume of values. Current lazy synthesis methods however are not among the ones that deliver the best visual quality.

Several solid texture synthesis methods arise as the extrapolation of a 2D model. While some approaches can be trivially expanded (e.g. procedural), others require a more complex strategy, e.g. pre-computation of 3D neighborhoods [DLTD08] or averaging of synthesized slices [HB95]. Currently, 2D texture synthesis methods based on Convolutional Neural Networks (CNN) [GEB15,UVL17] roughly define the standards in the 2D literature. \textit{Texture networks} methods [ULVL16,UVL17,LFY∗17a] stand out thanks to their fast synthesis times. In this work we introduce a similar CNN generator capable of synthesizing 3D textures on demand. There are substantial challenges in designing a CNN to accomplish such task. On demand evaluation requires the association of the synthesis to a co-
ordinate system and determinism to enforce spatial coherence. For the training, first we need to devise a way to evaluate the quality of the synthesized samples as there is no volumetric equivalent of the descriptor networks used in 2D methods. Second, we need a strategy to surmount the enormous amount of memory demanded by the task.

We propose a compact solid texture generator model based on CNN capable of on demand synthesis at interactive rates. On training, we assess the samples’ appearance via a volumetric loss function that compares slices of the generated textures to the target image. We exploit the stationarity of the model to propose a fast and memory efficient single-slice training strategy. This allows us to use target examples at higher resolutions than those in the current 3D literature. The resulting trained network is simple, lightweight, and powerful at reproducing the visual characteristics of the example on the cross sections of the generated volume along one or up to three directions.

2. Related works

To the best of our knowledge, the method proposed here is the first to employ a CNN to generate solid textures. Here we briefly outline the state of the art on solid texture synthesis, then we describe some successful applications of CNN to 2D texture synthesis. Finally we mention relevant CNN methods that use 2D views to generate 3D objects.

2.1. Solid texture synthesis

Procedural methods [Per85, Pea85] are quite convenient for computer graphics applications thanks to their real time computation and on demand evaluation capability. Essentially one can add texture to a 3D surface by directly evaluating a function given the coordinates of only the required (visible) points in the 3D space. The principle is as follows for texture generation on a surface: a colormap function, such as a simple mathematical expression, is evaluated at each of the visible 3D points. In [Per85], authors use pseudo-random numbers that depend on the coordinates of the local neighborhood of the evaluated point which ensures both the random aspect of the texture and its spatial coherence. Creating realistic textures with a procedural noise is a trial and error process that necessitates technical and artistic skills. Some procedural methods alleviate this process by automatically estimating their parameters from an example image [GD95, GLLD12, GSV14, GLM17]. However, they only deal with surface textures and most photo realistic textures are still out of the reach of these methods.

Most example-based solid texture synthesis methods aim at generating a full block of voxels whose cross-sections are visually similar to a respective texture example. These methods are in general fully automated and they are able to synthesize a broader set of textures. The texture example being only 2D, a prior model is required to infer 3D data. Some methods [HB95, DGF98, QhY07] employ an iterative slicing strategy where they alternate between independent 2D synthesis of the slices and 3D aggregation. Another strategy starts with a solid noise and then iteratively modifies its voxels by assigning them a color in function of a set of coherent pixels from the example. Wei [Wei03] uses the 2D neighborhood of a pixel also called patch to assess coherence, then, the set of contributing pixels is formed by the closest patch along each axis of the solid. Finally, the assigned color is the average of the contributing pixels. Dong et al. [DLTD08] determine coherence using three overlapping 2D neighborhoods (i.e. forming a 3D neighborhood) around each voxel and find only the best match among a set of precomputed candidates.

The example-based solid texture synthesis methods that achieve the best visual results in a wider category of textures [KFCO07, CW10] involve a patch-based global optimization framework [KEBK05] governed by a statistical matching strategy. This improves the robustness to failure. These methods, however, require high computation times, are limited to low resolution textures (typically $256^2$ input examples), and are incapable of on demand evaluation which limits their usability. Regarding speed, the patch-based method proposed by Dong et al. [DLTD08] allows for fast on demand evaluation, thus allowing for visual versatility and practicality. Here the patch-matching strategy is accelerated via the pre-processing of compatible 3D neighborhoods accordingly to the examples given along some axis. This preprocessing is a trade-off between visual quality and speed as it reduces the richness of the synthesized textures. Thus, their overall visual quality is less satisfactory than the one of the optimization methods previously mentioned.

2.2. Neural networks on texture synthesis

Our method builds upon previous work on example-based 2D texture synthesis using convolutional neural networks. We distinguish two types of approaches: image optimization and feed-forward texture networks.

Image optimization Image optimization methods were inspired from previous statistical matching approaches [PS00] and use a variational framework which aims at generating a new image that matches the features of an example image. The role of CNN in this class of methods is to deliver a powerful characterization of the images. It typically comes in the form of feature maps at the internal layers of a pretrained deep CNN [SZ14] namely VGG-19. Gatys et al. [GEB15] pioneered this approach for texture synthesis, by considering the discrepancy between the feature maps of the synthesized image and the example ones. More precisely for texture synthesis where one has to take into account spatial stationarity, the corresponding perceptual loss as coined later by [JAFF16] is the Frobenius distance of the Gram matrices of CNN features at different layers. Starting from a random initialization, the input image is then optimized via a stochastic gradient descent algorithm, where the gradient is computed using back-propagation through the CNN. Since then, several variants have built on this framework to improve the quality of the synthesis: for structured textures by adding a Fourier spectrum discrepancy [LGX16]; for non-local patterns by considering spatial correlation and smoothing [SCO17]; for stability by considering a histogram matching loss and smoothing [WRB17]. These methods deliver good quality and high resolution results as they can process images of resolutions up to $1024^2$ pixels. Their main drawback comes from the optimization process itself, as it requires several minutes to generate one image. Implementing local evaluation on these methods is infeasible.
since they use a global optimization scheme as for patch-based texture optimization methods [KEBK05, KFCO*07]. Some extension to dynamic texture synthesis has also been proposed in [TBD18]. The textured video is optimized using a perceptual loss combined with a loss based on estimated optical flow to take into account the time dimension.

**Feed-forward texture networks** Feed-forward networks approaches were introduced by Johnson et al. [JAFF16] for style transfer and Ulyanov et al. [ULVL16] for texture synthesis. In the latter, they train an actual generative CNN to synthesize texture samples that produce the same visual characteristics as the example. These methods use the loss function in [GEB15] to compare the visual characteristics between the generated and example images. However, instead of optimizing the generated output image, the training aims at tuning the parameters of the generative network. Such optimization can be more demanding since there is no spatial regularity shared across iterations as for the optimization of a single image. However this training phase only needs to be done once for a given input example. This is achieved in practice using back propagation and a gradient-based optimization algorithm using batches of noise inputs. Once trained, the generator is able to quickly generate samples similar to the input example by forward evaluation. Originally these methods train one network per texture sample. Li et al. [LFY*17a] proposed a training scheme to allow one network to have the capacity to generate and mix several different textures. By improving the capacity of the generator network Li et al. [LFY*17a] and Ulyanov et al. [ULVL16] methods reached a modestly higher visual quality but found that the synthesized textures did not change sufficiently for different inputs. In order to prevent the generator from producing identical images they were forced to incorporate a term that encourages diversity in the objective function. Feed-forward texture networks methods generally produce results with slightly lower visual quality than the image optimization methods, however they exhibit faster computation times. Moreover, as we show in Section 4 the texture network framework can be customized to allow on demand evaluation.

Other approaches achieve texture synthesis as a feed-forward evaluation of a generator network using a different training strategy. Bergmann et al. [BJV17] train a generator network without using the perceptual loss. Instead they employ a generative adversarial network framework [GPAM*14] with a purely convolutional architecture to allow for flexibility on the size of the samples to synthesize. This method shares the advantages of feed-forward networks regarding evaluation but is based in a more complex training scheme which can affect the quality on different texture examples. Finally, Li et al. [LFY*17b] proposed another strategy inspired from Auto-Encoder mechanism [Yan87, BK88]. They use truncated versions of VGG-19 network as encoders that map images to a feature space. Then they design decoder networks that generate images from such features. During the training stage, this generator is optimized to invert the encoder by trying to generate images that match encoded images from a large dataset of natural images. During synthesis, a random image and an exemplar image are first encoded; random features are matched to the target ones using first and second order moment matching, and then fed to the decoder to generate a random texture, without requiring a specific training for this example. While very appealing, this approach is difficult to adapt to 3D texture synthesis where such large dataset is not available. Moreover, the quality is not as good as for previously mentioned methods [GEB15, ULVL16].

### 2.3. Neural networks for volumetric object generation

A related problem in computer vision is the generation of binary volumetric objects from 2D images [GCC*17, JREM*16, YYY*16]. Similarly to our setting, these approaches rely on unsupervised training with a loss function comparing 3D data to 2D examples. However these methods do not handle color information and only produce at low resolution volumes (32³).

### 3. Overview

Figure 1 outlines the proposed method. We perform solid texture synthesis using the convolutional neural generator network $G$ detailed in Section 4. The generator with learnable parameters $\theta$, takes a multi-scale volumetric noise input $Z$ and processes it to produce a color solid texture $v = G(Z; \theta)$. The proposed model is able to perform on demand evaluation which is a critical property for solid texture synthesis algorithms. On demand evaluation spares computations and memory usage as it allows the generator to only synthesize the voxels that are visible.

The desired appearance of the samples $v$ is specified in the form of a view dissimilarity term for each direction. The 3D generated samples $v$ are compared to $D \in \{1, 2, 3\}$ exemplar images $\{u_1, \ldots, u_D\}$ that correspond to the desired view along $D$ directions among the 3 canonical directions of the Cartesian grid. The generator learns to sample solid textures from the visual features extracted in the examples via the optimization of its parameters $\theta$. To do so, we formulate a volumetric slice-based loss function $L$. It measures how appropriate the appearance of a solid sample is by comparing its 2D slices $v_{d,n}$ ($n$-th slice in along the $d$-th direction) to each corresponding example $u_d$. The comparison is carried out in the space of features from the descriptor network $D$, based on VGG-19 similarly to previous 2D methods.

The training scheme, detailed in Section 5, involves the generation of batches of solid samples which would $a priori$ require a prohibitive amount of memory if relying on classical optimization approach for CNN. We overcome this limitation thanks to the stationarity properties of the model. We show that training the proposed model only requires the generation of single slice volumes along the specified directions. Section 6 presents experiments and comparative results. Finally, in Section 7 we discuss the current limitations of the model.

### 4. On demand evaluation enabled CNN generator

The architecture of the proposed CNN generator is summarized in Figure 2 and detailed in Subsection 4.1. The generator applies a series of convolutions to a multi-scale noise input to produce a single solid texture. It is inspired by Ulyanov et al. [ULVL16] model, which is based on a multi-scale approach inspired itself from the human visual system that has been successfully used in
4.1. Architecture

The generator produces a solid texture \( v = G(Z|\theta) \) from a set of multi-channel volumetric white noise inputs \( Z = \{z_0, \ldots, z_K\} \). The spatial dimensions of \( Z \) directly control the size of the generated sample. The process of transforming the noise \( Z \) into a solid texture \( v \) is depicted in Figure 2. It follows a multi-scale architecture built upon three main operations: convolution, concatenation, and upsampling. Starting at the coarsest scale, the 3D noise sample is processed with a set of convolutions followed by an upsampling to reach the next scale. It is then concatenated with the independent noise sample from the next scale, itself also processed with a set of convolutions. This process is repeated \( K \) times before a final single convolution layer that maps the number of channels to three to get a color texture. We now detail the three different blocks of operations used in the generator network.

**Convolution block** A convolution block groups a sequence of three ordinary 3D convolution layers, each of them followed by a batch-normalization and a leaky rectified linear unit function. Considering \( M_{in} \) and \( M_{out} \) channels in the input and output respectively, the first convolution layer carries out the shift from \( M_{in} \) to \( M_{out} \). The following two layers of the block have \( M_{out} \) channels in both the input and the output. The size of the kernels is \( 3 \times 3 \times 3 \) for the first two layers and \( 1 \times 1 \times 1 \) for the last. Contrary to \([ULVL16]\) and in order to enable on demand evaluation (see Subsection 4.2), here the convolutions are computed densely and without padding, thus discarding the edge’s values. Applying one convolution block with these settings to a volume reduces its size by 4 values per spatial dimension.

**Upsampling** An upsampling performs a 3D nearest neighbor upsampling by a factor of 2 on each spatial dimension (i.e. each voxel is replicated 8 times).

**Channel concatenation** This operation first applies a batch normalization operation and then concatenates the channels of two multi-channel volumes having the same spatial size. If different, the biggest volume is cropped to the size of the smallest one.

The learnable parameters \( \theta \) of the generator are: the convolution’s kernels and bias, and the batch normalization layers’ weight, bias, mean and variance. The training of these parameters is discussed in Section 5.

4.2. Spatial dependency

Forward evaluation of the generator is deterministic. By handling the edges correctly, we can feed the network separately with two contiguous portions of noise to synthesize textures that can be tiled seamlessly. Current 2D CNN methods \([ULVL16, LFY^{17a}]\) perform padded convolutions, not addressing on demand evaluation capabilities. Let us notice that a perfect tiling between two different samples can only be achieved by using convolutions without padding. Therefore we discard the values on the edges where the operation with the kernel cannot be carried out completely.

When synthesizing a sample, the number of discarded values is
therefore $c_k$ (where $\lceil \cdot \rceil$ stands for the ceiling function) except for the coarsest scale has to be properly done for each dimension to compensate for the coordinate alignment across scales, i.e. which coordinate $n_k$ at scale $k$ must be associated to a given coordinate $n_0$ at the finest scale $k = 0$. Indeed, after every upsampling operation, observe that each value is repeated twice along each spatial dimension, pushing the rest of the values spatially. Depending on the coordinates of the reference voxel being synthesized, this shift of one position can disrupt the coordinate alignment with the subsequent scale. Therefore, the generator network has to compensate accordingly before each concatenation.

For $K = 5$ upsamplings, one of the $2^K = 32$ combinations of compensation shifts has to be properly done for each dimension to synthesize a given voxel. Given the final coordinate $n_0$ of the voxel (in any spatial dimension), these shifts one are deduced recursively from the following relation $n_{k-1} = 2n_k + s_k$, where $n_k$ is the spatial coordinate used to generate the noise at scale $k \in \{1, \ldots, K\}$, and $s_k \in \{0, 1\}$ is the shift value used after the $k$th upsampling operation.

5. Training

Here we detail our approach to obtain the parameters $\theta_n$ that drive the generator network to synthesize solid textures specified by the example $u$. Like current texture networks methods, we leverage the power of existing training frameworks to optimize the parameters of the generator. Typically an iterative gradient-based algorithm is used to minimize a loss function that measures how different the synthetic and target textures are.

However a first challenge facing the training of the solid texture generator is to devise a discrepancy measure between the solid texture and the 2D examples. In Subsection 5.1 we propose a 3D slice-based loss function that collects the measurements produced by a set of 2D comparisons between 2D slices of the synthetic texture and the 2D examples.
solid and the examples. We conduct the 2D comparisons similarly to the state of the art methods, using the perceptual loss function [GEB15, JAFF16, ULVL17].

The second challenge comes from the memory requirements during training. Typically the optimization algorithm estimates a descent direction by applying backpropagation on the loss function evaluated on a batch of samples. In the case of solid textures, each volumetric sample occupies a large amount of memory, which makes the batch processing impractical. Instead, we show in Subsection 5.2 that thanks to the stationary properties of our generative model we can carry out the training using batches of single slice solid samples.

5.1. 3D slice-based loss

For a color solid \( v \in \mathbb{R}^{N_t \times N_v \times N_s \times 3} \), we denote by \( v_{d,n} \), the \( n \)-th 2D slice of the solid \( v \) orthogonal to the \( d \)-th direction. Given a number \( D \leq 3 \) of slicing directions and the corresponding example images \( \{u_1, \ldots, u_D\} \), we propose the following slice based loss

\[
\mathcal{L}(v|\{u_1, \ldots, u_D\}) = \sum_{d=1}^{D} \frac{1}{N_v} \sum_{n=0}^{N_s-1} \mathcal{L}_2(v_{d,n}, u_d),
\]

where \( \mathcal{L}_2(v,u) \) is a 2D loss that computes the similarity between an image and the example \( u \).

We use the 2D perceptual loss \( \mathcal{L}_2 \) from [GEB15], which proved to be successful for training CNN [JAFF16, ULVL16]. It compares the Gram matrices of the VGG-19 feature maps of the synthesized and example images. The feature maps result from the evaluation of the descriptor network \( D \) on an image, i.e. \( D : x \in \mathbb{R}^{N_t \times N_v \times 3} \mapsto \{F^l(x) \in \mathbb{R}^{N^l \times M^l}\}_{l \in L} \), where \( L \) is the set of considered VGG-19 layers, each layer \( l \) having \( N^l \) spatial values and \( M^l \) channels. For each layer \( l \), the Gram matrix \( G^l \in \mathbb{R}^{M^l \times M^l} \) is computed from the feature maps as

\[ G^l(x) = \frac{1}{N^l} F^l(x)^T F^l(x), \]

where \( T \) is the matrix transpose. The 2D loss between the input example \( u_d \) and a slice \( v_{d,n} \) is then defined as

\[
\mathcal{L}_2(v_{d,n}, u_d) = \sum_{l \in L} \frac{1}{(M^l)} \left\| G^l(v_{d,n}) - G^l(u_d) \right\|_F^2,
\]

where \( \| \cdot \|_F \) is the Frobenius norm. Observe that the Gram matrices are computed along spatial dimensions to take into account the stationarity of the texture. Those Gram matrices encode both first and second order information of the feature distribution (covariance and mean).

5.2. Single slice training scheme

Formally, training the generator \( G(Z, \theta) \) with parameters \( \theta \) corresponds to minimizing the expectation of the loss in Equation 1 given the example \( u \)

\[
\theta_{\theta} \in \text{argmin}_{\theta} \mathbb{E}_Z [\mathcal{L}(G(Z, \theta), \{u_1, \ldots, u_D\})],
\]

where \( Z \) is a multi-scale white noise sample. Note that \( Z \) is thus translation invariant. The generator on the other hand induces a non stationary behavior on the output due to upsampling operations. Indeed when upsampling a stationary signal by a factor 2 with a nearest neighbor interpolation the resulting process is only invariant to translations of multiples of two. Because our model contains \( K \) volumetric upsampling operations, the process is translation invariant by multiples of \( 2^K \) values on each axis. Considering the \( d \)-th axis, for any coordinate at the scale of the generated sample \( n = 2^K p + q \) with \( q \in \{0, \ldots, 2^K - 1\} \), the statistics of the slice \( G(Z, \theta)_{d,n} \) only depend on the value of \( q \), therefore

\[
\mathbb{E}_Z [\mathcal{L}_2(G(Z, \theta)_{d,n}, u_d)] = \mathbb{E}_Z [\mathcal{L}_2(G(Z, \theta)_{d,q}, u_d)].
\]

(5)

Assuming \( N_d \) is a multiple of \( 2^K \), we have

\[
\mathbb{E}_Z [\mathcal{L}(G(Z, \theta), \{u_1, \ldots, u_D\})] = \sum_{d=1}^{D} \frac{1}{N_d} \sum_{n=0}^{2^K-1} \mathbb{E}_Z [\mathcal{L}_2(G(Z, \theta)_{d,n}, u_d)]
\]

\[
= \sum_{d=1}^{D} \frac{1}{2^K} \sum_{q=0}^{2^K-1} \mathbb{E}_Z [\mathcal{L}_2(G(Z, \theta)_{d,q}, u_d)].
\]

(6)

As a consequence, instead of using \( N_d \) slices per direction the generator network could be trained using only a set of \( 2^K \) contiguous slices on each constrained direction.

The GPU memory is a limiting factor during the training process, even cutting down the size of the samples to \( 2^K \) slices restricts the training slice resolution. For example, training a network for a texture output of size \( 512 \times 512 \times 32 \) with \( K = 5 \) and VGG-19 would require more than 12GB of memory. For that reason we propose to stochastically approximate the inner sum in Equation (6).

Considering the slice \( n = 2^K p + q_d \) in the \( d \)-th axis with a fixed \( p \in \{0, \ldots, \frac{N_d}{2^K} - 1\} \) and with \( Q_d \in \{0, \ldots, 2^K - 1\} \) randomly drawn from a discrete uniform distribution,

\[
\mathbb{E}_{Z,Q_d} [\mathcal{L}_2(G(Z, \theta)_{d,Q_d}, u_d)] = \frac{1}{2^K} \sum_{q=0}^{2^K-1} \mathbb{E}_Z [\mathcal{L}_2(G(Z, \theta)_{d,q}, u_d)].
\]

(7)

Then using doubly stochastic sampling (noise input values and output coordinates) we have

\[
\mathbb{E}_Z [\mathcal{L}(G(Z, \theta), \{u_1, \ldots, u_D\})] = \sum_{d=1}^{D} \mathbb{E}_{Z,Q_d} [\mathcal{L}_2(G(Z, \theta)_{d,Q_d}, u_d)],
\]

(8)

which means that we can train the generator using only \( D \) single-slice volumes oriented according to the constrained directions. Note that the whole volume model is impacted since the convolution weights are shared by all slices.

The proposed scheme saves computation time during the training, and more importantly, it also reduces the required amount of memory. In this setting we can use resolutions of up to 1024 values per slice during training (examples and solid samples), a resolution significantly larger than the ones reached in the literature regarding solid texture synthesis by example which are usually limited to \( 256 \times 256 \).
6. Results

6.1. Experimental settings

Unless otherwise specified all the results in this section were generated using the following settings.

Generator network We set the number of scales to 6, i.e. $K = 5$, which means that each voxel of the input noise at the coarsest scale impacts nearly 300 voxels at the finest scale. We use $M_i = 3$ input channels and $M_s = 4$ (number of channels after the first convolution block and channel step across scales) which results in the last layer being quite narrow ($6M_s = 24$) and the whole network compact, with $\sim 8.5 \times 10^3$ parameters. We include a batch normalization operation after every convolution layer and before the concatenations. As mentioned in previous methods [ULVL16] we noticed that such a strategy helps stabilizing the training process.

Descriptor network Following Gatys et al. [GEB15], we use a truncated VGG-19 [SZ14] as our descriptor network, with padded convolutions and average pooling. The considered layers for the loss are: relu1_1, relu2_1, relu3_1, relu4_1 and relu5_1.

Training We implemented our approach using pytorch and we use the pre-trained parameters for VGG-19 available from the BETHGE LAB [GEB15, GEB16]. We optimize the parameters of the generator network using Adam algorithm [KB15] with a learning rate of 0.1 during 3000 iterations. Figure 3 shows the value of the empirical estimation of $L$ during the training of three of the examples shown below in Figure 6 (histology, cheese and red marble). We use batches of 10 samples per slicing direction. We compute the gradients individually for each sample in the batch which slows down the training process but allows us to concentrate the available memory on the resolution of the samples. With these settings and using 3 slicing directions, the training takes around 1 hour for a 128$^3$ training resolution (i.e. size of the example(s) and generated slices), 3.5 hours for 256$^3$ and 12.5 hours for 512$^3$ using one GPU Nvidia GeForce GTX 1080 Ti.

Synthesis All the generated samples are built by assembling blocks of 32$^3$ voxels using on demand evaluation. Figure 4 depicts how the small pieces of texture tile perfectly to form a bigger texture. It takes nearly 12 milliseconds to generate a block of texture of 32$^3$ voxels on a Nvidia GeForce RTX 2080 GPU. However we can use bigger elemental blocks, e.g. a cube of 64$^3$ voxels takes $\sim 24$ milliseconds and one of 128$^3$ voxels takes $\sim 128$ milliseconds. For reference, using the method of Dong et al. [DLTD08] it takes 220 milliseconds to synthesize a 64$^3$ volume and 1.7 seconds to synthesize a 128$^3$ volume.

6.2. Experiments

In this section we highlight the various properties of the proposed method and we compare them with state of the art approaches.

Texturing mesh surfaces Figure 5 exhibits the application of textures generated with our model to add texture to 3D mesh models. In this case we generate a solid texture with a fixed size and load it on OpenGL as a regular 3D texture with bilinear filtering.

Single example setting Figures 6 and 7 show synthesized samples using our method on a set of examples depicting some physical material. Considering their isotropic structure, we train the generator network using a single example to designate the appearance along the three orthogonal directions, i.e. $u_1 = u_2 = u_3$. On the first column we show a generated sample of size 512$^3$ built by assembling blocks of 32$^3$ voxels generated using on demand evaluation. The second column is the example image of size 512$^2$, columns 3-5 show the middle slices of the generated cube across the three considered directions and the last column shows a slice extracted in an oblique direction with a 45° angle. These examples illustrate the capacity of the model to infer a plausible 3D structure from the 2D features present in isotropic example images. Observe that a slice across an oblique direction still displays a conceivable structure given the examples. They also demonstrate the spatial consistency while using on demand evaluation.

Existence of a solution For a given texture example $u$ and a number of directions $D > 1$ it is possible that a corresponding 3D texture
Figure 5: We use textures generated with our method to add texture to 3D mesh models. The example texture used to train the generator is shown in the upper left corner of each object. When using solid textures the mapping does not require parametrization as they are defined in the whole 3D space. This prevents any mapping induced artifacts. Sources: the ‘duck’ model comes from Keenan’s 3D Model Repository, the ‘mountain’ and ‘hand’ models from free3d.com and the tower and the vase from turbosquid.com.

does not exist, i.e. not all the slices in the chosen directions will respect the structure defined by the 2D example. This existence issue is vital for example-based solid texture synthesis as it delineates the limits of this slice based formulation used by many methods in the literature. It is briefly mentioned in [KFCO’07, DLTD08] and here we aim to extend the discussion. Let us consider for instance the isotropic example shown in Figure 8, where the input image contains only discs at a given scale (e.g. a few pixels diameter). It follows that when slicing a volume containing spheres with the same diameter $\delta$, the obtained image will necessarily contain objects with various diameters, ranging from 0 to $\delta$. This seems to be a paradigm natural to the 2D-3D extrapolation. It demonstrates that for some 2D textures an isotropic solid version might be impossible and conversely, that the example texture and the imposed directions must be chosen carefully given the goal 3D structure.

This also might have dramatic consequences regarding convergence issues during optimization. In global optimization methods [KFCO’07, CW10], where patches from the example are sequentially copied view per view, progressing the synthesis in one direction can create new features in the other directions thus potentially preventing convergence. In contrast, in our method, we seek for a solid texture whose statistics are as close as possible from the examples without requiring a perfect match. This always ensured convergence during training in all of our experiments. An example of this is illustrated in the first row of Figure 10, where the
Figure 6: Synthesis of isotropic textures. We train the generator network using the example in the second column of size $512^2$ along $D = 3$ directions. The cubes on the first column are generated samples of size $512^3$ built by assembling blocks of $32^3$ voxels generated using on demand evaluation. Subsequent columns show the middle slices of the generated cube across the three considered directions and a slice extracted in an oblique direction with a $45^\circ$ angle. The trained models successfully reproduce the visual features of the example in 3D.
**Figure 7**: Synthesis of isotropic textures (same presentation as in Figure 6). While generally satisfactory, the examples in the first and third rows have a slightly degraded quality. In the first row the features have more rounded shapes than in the example and in the third row we observe high frequency artifacts.

Example views are incompatible given the proposed 3D configuration: the optimization procedure converges during training and the trained generator is able to synthesize a solid texture, however, the result is clearly a tradeoff which mixes the different contradictory orientations. This contrasts with methods in 2D texture synthesis where constrained optimization is generally beneficial, for instance by projecting patches [GRGH17], histogram matching [RPDB12], or moment matching [PS00].

**Constraining directions** While for isotropic textures, constraining $D = 3$ directions gives good visuals results, it may be more interesting to consider only two views for some textures. Indeed using $D < 3$ might be essential to obtain acceptable results, at least along the considered directions. This acts in accordance to the question of existence of a solution discussed in the previous paragraph. We exemplify this in the top two rows of Figure 9, where considering only two training directions (second row) results in a solid texture that more closely resembles the example along those directions, compared to using three training directions (first row) which generates a more consistent volume using isotropic shapes that are not present in the example texture. The brick and cobblestone textures in Figure 9 highlight the fact that, when depicting an object...
where the top view is not crucial to the desired appearance, such as a wall, it can be left to the algorithm to infer a coherent structure. Of course, not considering a direction during training will generate cross-sections that do not necessarily contain the same patterns as the example texture (see column \( v \), \( N \)), but rather color structures that fulfill the visual requirements for the other considered directions.

Additionally, pattern compatibility across different directions is essential to obtain coherent results. In the examples of Figure 10 the generator was trained with the same image but in a different orientation configuration. In the top row example no 3D arrangement of the patterns can comply with the orientations of the three examples. Conversely the configuration on the bottom row can be reproduced in 3D thus generating more meaningful results. All this has to be taken into account when choosing the set of training directions given the expected 3D texture structure.

**Diversity** A required feature of texture synthesis models is that they are able to generate diverse samples. Ideally the generated samples are different from each other and from the example itself while still sharing its visual characteristics. Additionally, depending on the texture it might be desired that the patterns vary spatially inside each single sample. Yet, observe that many methods in the literature for 2D texture synthesis generate images that are local copies of the input image [WL00, KEBK05], which strongly limits the diversity. As reported in Gutierrez et al. [GRGH17], the (unwanted) optimal solution of methods based on patch optimization is the input image itself. For most of these methods though, the local copies are sufficiently randomized to deliver enough diversity. Variability issues have also been reported in the literature for texture generation based on CNN, and [UVL17, LFY+17a] have proposed to use a diversity term in the training loss to fix it. Without such diversity term to promote variability, the generated samples are nearly identical from each other, although sufficiently different from the example. In these cases it seems that the generative networks learn to synthesize a single sample that induces a low value of the perceptual loss while disregarding the random inputs.

When dealing with solid texture synthesis from 2D examples, such a trivial optimal solution only arises when considering one direction, where the example itself is copied along such direction. Yet, there is no theoretical guarantee that prevents the generator network from copying large pieces of the example as it has been shown that deep generative networks can memorize an image in [LVU18]. However, the compactness or the architecture and the stochastic nature of the proposed model make it very unlikely. In practice, we do not observe repetition among the samples generated with our trained models, even when pursuing the optimization long after the visual convergence (which generally occurs after 1000 iterations, see Figure 3). This is consistent with the results of Ulyanov et al. [ULVL16], the 2D architecture that inspired ours, where diversity is not an issue. One explanation for this difference with other methods may be that the architectures that exhibit a loss of diversity process an input noise that is small compared to the generated output and which is easier to ignore. On the contrary, our generative network receives an input that accounts for almost twice the size of the output. Figure 11 shows the capacity of our model to generate diverse sets of samples from a single trained generator.

**Multiple examples setting** As already discussed in the work of Kopf et al. [KFCO+07] and earlier, it appears that most solid textures can only be modeled from a single example along different directions. In the literature, to the best of our knowledge, only one success case of a 3D texture using two different examples has been proposed [KFCO+07, DLTD08]. This is due to the fact that the two examples have to share similar features such as color distribution and compatible geometry as already shown in Figure 10. Figure 12 illustrates this phenomenon, for each example we experiment with and without performing a histogram matching (independently for each color channel) to the input examples. We observe favorable results particularly when the colors of both examples are close. Although the patterns are not perfectly reproduced, the 3D structure is coherent and close to the examples.

### 6.3. Comparison with state of the art

We compare the visual quality of our results against the two existing methods that seem to produce the best results: Kopf et al. [KFCO+07] and Chen et al. [CW10]. Figure 13 shows some samples obtained from the respective articles or websites side by side with results using our method at two different resolutions. The most salient advantage of our method is the ability to better capture high frequency information, making the structures in the samples sharper. Considering voxels’ statistics, both our method and that of Chen et al. [CW10] seem to obtain a better result than the method of Kopf et al. [KFCO+07].

We do not consider the method of Dong et al. [DLTD08] for a visual quality comparison as their pre-computation of candidates limits the richness of information, which yields lower quality results.
Figure 9: Training the generator using two or three directions for anisotropic textures. The first column shows generated samples of size $512^3$ built by assembling blocks of $32^3$ voxels generated using on demand evaluation. The second column illustrates the training configuration, i.e. which axes are considered and the orientation used. Subsequent columns show the middle slices of the generated cube across the three considered directions. The top two rows show that for some examples considering only two directions allow the model to better match the features along the directions considered. The bottom rows show examples where the appearance along one direction might not be important.

However, thanks to the on-demand evaluation, this model greatly surpasses the computation speeds of the other methods. Yet, as detailed before, our method is faster during synthesis while achieving better visual quality. Besides, our computation time does not depend on the resolution of the examples.

On Figure 14 we show some of our results used for texturing a complex surface and we compare them to the results of Kopf et al. [KFCO*07]. Here the higher frequencies successfully reproduced with our method cause a more realistic impression.

Finally we would like to point out that although deep learning related models are often thought to produce good results only thanks to a colossal amount of parameters, our method (with $\sim 8.5 \times 10^5$ parameters to store) stands close to the memory footprint of a patch-based approach working with a $170^2$ color pixels input (i.e. $8.67 \times 10^5$ parameters if all the patches are used).
Figure 10: Importance of the compatibility of examples. In this experiment, two generators are trained with the same image along three directions, but for two different orientations. The first column shows generated samples of size $512^3$ built by assembling blocks of $32^3$ voxels generated using on demand evaluation. The second column illustrates the training configuration, i.e. for each direction the orientation of the example shown in the third column. Subsequent columns show the middle slices of the generated cube across the three constrained directions. In the first row, no 3D arrangement of the patterns can comply with the orientations of the three examples. Conversely the configuration on the second row can be reproduced in 3D, thus generating more meaningful results.

Figure 11: Diversity among generated samples. Each set of five cubes is generated with the same network, for different random inputs. The synthesized textures share the same characteristics without any repetition.

7. Limitations and future work

Long distance correlation As it can be observed in the brick wall texture on Figure 9 and in the diagonal texture in Figure 10 our model is less successful at preserving the alignment of long patterns in the texture. This limitation is also observed in the second row of Figure 9 where the objects size in the synthesized samples do not match the one in the example, again due to the overlooked long distance correlation. A simple solution for this case could be to use more scales in the generator network, similarly to use larger patches in patch based methods. An other possible improvement to explore is to explicitly construct our 2D loss $L_2$ incorporating those long distance correlations as in [LGX16, SCO17].

Constrained directions We observed that training the generator with two instead of three constrained directions results in unsatisfying texture along the unconsidered direction, while improving visual quality along the two constrained directions for anisotropic textures (see Figure 9). It would be interesting to explore a middle point between letting the algorithm infer the structure along one direction and constraining it.

Visual quality Although our method delivers high quality results for a varied set of textures, it still presents some visual flaws that we think are independent of the existence issue. In textures like the pebble and grass of Figure 7 the synthesized sample presents oversimplified versions of the example’s features. Although not detailed in the articles, the available codes for [ULVL16, UVL17, LFY∗17a] make use of an empirical normalization of the gradients during training. This technique normalizes the gradient of the generator...
network with respect to the loss at each layer before continuing the back propagation to the generator network’s parameters. In practice it sometimes leads to a slightly closer reproduction of the patterns’ structure of the example. It is however difficult to anticipate which textures can benefit from this technique.

Additionally, our results present some visual artifacts that are typical to generative methods based on deep networks. The most salient are the high frequency checker-board effects, see for instance [JAFF16] where a total variation term is used to try to mitigate the artifact.

**Real time rendering** The trained generator can be integrated in a fragment shader to generate the visible values of a 3D model thanks to its on demand capability. Note however that on-the-fly filtering of the generated solid texture is a challenging problem that is not addressed in this work.

### 8. Conclusion

We addressed the problem of example based solid texture synthesis by the means of a convolutional neural network framework. First we presented a simple and compact generator network capable of synthesizing portions of infinitely extendable solid texture. We train this generator network using a pre-trained 2D descriptor network and a 3D single slice loss. The complete framework is efficient both during training and at evaluation time. The training can be performed at high resolutions, and textures of arbitrary sizes can be synthesized on demand. The method is capable of synthesizing high quality results on a wide set of textures. We showed the outcome on textures with varying levels of structure and on isotropic and anisotropic arrangements. We showed that although solid texture synthesis from a single example image is an intricate problem, our method delivers compelling results given the desired look imposed via the 3D loss construction. The on demand evaluation capability of the generator allows for it to be integrated with a 3D graphics renderer to replace the use of 2D textures on surfaces and thus eliminating the possible accompanying artifacts. We observed some limitations of our method mainly in the lack of control over the directions not considered in the training. Using multiple examples could complement the training by giving information of the desired aspect along different directions. We aim to further study the limits of solid texture synthesis from multiple sources with the goal of obtaining an upgraded framework better capable of simulating real life objects.

### Acknowledgments

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC), RGPIN-2015-06025. This study has been also carried out with financial support from the CNRS for supporting grants PEPS 2018 I3A “3DTextureNets”. In like manner, we acknowledge the support of CONACYT. Bruno Galerne acknowledges the support of NVIDIA Corporation with the donation of a Titan Xp GPU used for this research. The authors would like to thank Loïc Simon for fruitful discussions on 3D rendering.

---

**Table 1: HM examples configuration**

<table>
<thead>
<tr>
<th>HM</th>
<th>examples</th>
<th>configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><img src="image1.png" alt="Example 1" /></td>
<td><img src="image2.png" alt="Configuration 1" /></td>
</tr>
<tr>
<td>✔️</td>
<td><img src="image3.png" alt="Example 2" /></td>
<td><img src="image4.png" alt="Configuration 2" /></td>
</tr>
<tr>
<td></td>
<td><img src="image5.png" alt="Example 3" /></td>
<td><img src="image6.png" alt="Configuration 3" /></td>
</tr>
<tr>
<td></td>
<td><img src="image7.png" alt="Example 4" /></td>
<td><img src="image8.png" alt="Configuration 4" /></td>
</tr>
</tbody>
</table>

**Figure 12:** Anisotropic texture synthesis using two examples. The first columns show the two examples used and the training configuration, i.e. how images are oriented for each view. Last column shows a sample synthesized using the trained generator. For each example, we experiment with (✔️) and without (✖️) preprocessing the example images to match color statistics, by performing a histogram matching (HM) on each color channel independently. We observe favorable results particularly when the colors of both examples are close.
Figure 13: Comparison with the existing methods that produce the best visual quality. The last two columns show the results using the proposed method on two different sizes to simplify comparison. Besides being able to handle higher resolutions, our method is better at reproducing the statistics of the example compared to [KFCO*07] and is better at capturing high frequencies compared to both methods.


Figure 14: Comparison of our approach with Kopf et al. [KFCO*07]. Both methods generate a solid texture that is then simply interpolated and intersected with a surface mesh (without parametrization as required for texture mapping). The first column shows the example texture. The second column shows results from [KFCO*07], some of which obtained with additional information (a feature map for the second and fourth rows, a specularity map for the third one). The last column illustrates that our approach, using only the example for training, is able to produce fine scale details. The resolution of the first three rows are 640x3 and 512x2 for the rest.


