# CHAPTER 3. A SINGLE IMAGE GENERATIVE MODEL FOR TILEABLE TEXTURE SYNTHESIS

Figure 3.1: Results obtained with SeamlessGAN for tileable texture synthesis. We show 2×2 repetitions of the outputs on the squared images, and the input exemplars by their side.

In this chapter, we present *SeamlessGAN*, a method capable of automatically generating tileable texture maps from a single input exemplar. In contrast to most existing methods, focused solely on solving the synthesis problem, our work tackles both problems, *synthesis* and *tileability*, simultaneously. Our key idea is to realize that tiling a latent space within a generative network trained using adversarial expansion techniques produces outputs with continuity at the seam intersection that can then be turned into tileable images by cropping the central area. Since not every value of the latent space is valid to produce high-quality outputs, we leverage the discriminator as a perceptual error metric capable of identifying artifact-free textures during a sampling process. Further, in contrast to previous work on deep texture synthesis, our model is designed and optimized to work with multi-layered texture representations, enabling textures composed of multiple maps such as albedo, normals, etc. We extensively test our design choices for the network architecture, loss function, and sampling parameters. We show qualitatively and quantitatively that our approach outperforms previous methods and works for textures of different types. The contributions presented in this chapter have led to the following publication[1]:

[1] Additional publication details and results are included on the project website.


SeamlessGAN: Self-Supervised Synthesis of Tileable Texture Maps

"SeamlessGAN: Self-Supervised Synthesis of Tileable Texture Maps" Carlos Rodriguez-Pardo, Elena Garces

*IEEE Transactions on Visualization and Computer Graphics (TVCG)*

(2022)

## 3.1 Introduction

Realistic and high-quality textures are important elements to convey realism in virtual environments. These can be procedurally generated [Tu+20; HDR19; Gue+20; Gal+12; Gil+14; Gui+17; HN18], captured [Guo+20b; LSC18] or synthesized from real images [EF01; Kwa+05; Zho+18; Mor+17]. Frequently, textures are used to efficiently reproduce elements with repetitive patterns (for example, facades, surfaces, or materials) by means of spatially concatenating -or *tiling*- multiple copies of themselves. Creating tileable textures is a very challenging problem, as it requires a semantic understanding of the repetitive elements, often at multiple scales. For this reason, such a process is frequently done manually by artists in 3D digitization pipelines.

Recent advances in Convolutional Neural Networks (CNNs) and Generative Adversarial Networks (GANs) have been applied to texture synthesis problems [Zho+18; FAW19; Liu+20; BJV17; JBV16; Mar+20; Her+20] showing unprecedented levels of realism and quality, however, the output of these methods is not tileable. Despite recent methods [DH19; BJV17; Rod+19; Mor+17; Li+20; Nik+21] addressing the problem of tileable texture synthesis, we show that they either assume a particular level of regularity or the generated textures lose a significant amount of visual fidelity with respect to the input exemplars. Further, most of these methods have only focused on synthesizing single images. Rendering realistic materials requires more information about their optical properties beyond what represents a single RGB pixel. To this end, it is common to use spatially-varying BRDFs [Des+19], which are optical appearance models parameterized by stacks of images, each one representing a different property, such as albedo, normals, or transparency. As the number of methods that generate texture stacks from physical samples grows [Des+19; DDB20; Guo+20b] so does the need to turn them into *tileable texture stacks*.

In this chapter, we propose a deep generative model, *SeamlessGAN*, capable of synthesizing tileable texture stacks from inputs of arbitrary content. In contrast

to *Wang Tiles* [Coh+03], by which a single large texture region is created by concatenating multiple different tiles with matching borders, we aim to automatically obtain a seamless single-tile, which allows for reduced memory consumption and enhanced usability in casual scenarios when the user might lack the necessary artistic skills. Our key idea is to realize that tiling a latent space within a generative network produces outputs with continuity at the seam intersection [FAW19], which can then be turned into tileable images by cropping the central area. Since not every value of the latent space is valid to produce high-quality outputs, we follow a double strategy: First, we train the generative network using an adversarial expansion technique [Zho+18], which provides a latent encoding of the input texture, which can then be decoded into high-quality outputs that double the spatial extent of the input. Second, we use the trained discriminator as a local quality metric in a sampling algorithm. This allows us to find the input of the generative process that produces tileable textures similar to the original, as well as multiple candidates per input exemplar. As opposed to previous work, which focused on maximizing stationarity [Mor+17] and thus might remove important high-level texture features in regular or near-regular textures, our method is focused on maximizing tileability while preserving the original texture as intact as possible, in terms of its stylistic and semantic properties. To allow for the synthesis of stacks of textures, we propose a neural architecture composed of various decoder networks. Despite not explicitly imposing inter-map consistency, we show that it is implicitly guaranteed by how the generative network is trained. Without losing generality, we show texture stacks synthesis of two maps: an albedo and a normal map. We demonstrate that our method outperforms state-of-the-art solutions on tileable texture synthesis of single images and show several examples for synthesizing tileable texture stacks. We validate our design choices through several ablations studies and off-the-shelf perceptual quality metrics.

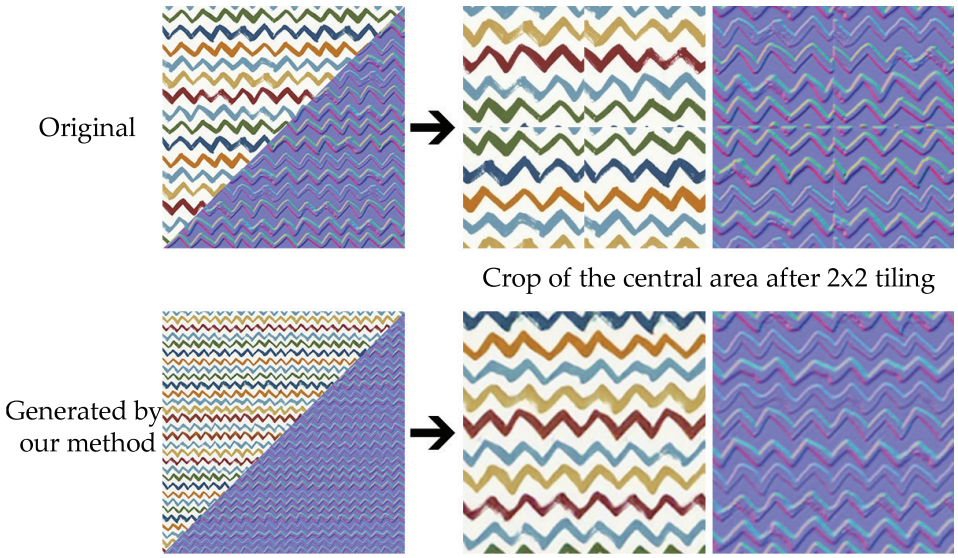
## 3.2 Related work

We review the texture synthesis methods most closely related to our work. For a more comprehensive review, please refer to the surveys in [Akl+18; Raa+18]. We will also mention other related work regarding tileable texture synthesis, and deep internal learning.

### 3.2.1 Texture Synthesis

Traditionally, **non-parametric texture synthesis** algorithms worked by ensuring that every patch in the output textures approximates a patch in the input texture. Earlier methods included image quilting [EF01; EL99], GraphCuts [Kwa+03], genetic algorithms [DZP07], and optimization [Kwa+05; PS00]. More recent approaches use variations of PatchMatch [Bar+09; Bar+15] as a way of

Figure 3.2: Naïvely tiling the original texture causes discontinuities at the seam intersections as shown in the top row. Our method automatically generates a tileable texture stack from an input exemplar which double the size of the input.




finding correspondences between generated and input images [Kas+15; Dar+12; Zho+17].

Despite those methods showing high-quality results for textures of different characteristics, recent work on deep parametric texture synthesis shows better generality and scalability, requiring less manual input. Our approach belongs to the category of **Parametric texture synthesis**. These methods learn statistics from the example textures, which can then be used for the generation of new images that match those statistics. While traditional methods used hand-crafted features [De 97; HB95], recent parametric methods rely on deep neural networks as their parameterization. Activations within latent spaces in pre-trained CNNs have been shown to capture relevant statistics of the style and texture of images [GEB16a; Gat+17; JAF16; Ren+17; RB19]. Textures can be synthesized through this approach by gradient-descent optimization [Sne17; GEB15a] or by training a neural network that learns those features [DB16; Nik+21]. Finding generic patterns that precisely describe the example textures is one of the main challenges in parametric texture synthesis. Features that describe textured images in a generic way are hard to find and they typically require hand-tuning. Generative Adversarial Networks (GANs), which have shown remarkable capabilities in image generation in multiple domains [Kar+18; KLA19; Kar+20b], can learn those features from data. Specifically, in texture synthesis, they have proven successful at generating new samples

of textures from a single input image [Zho+18] or from a dataset of images [BJV17; FAW19; JBV16; Liu+20; Mar+20]. We build upon the method of Zhou *et al.* [Zho+18] which shows good performance on the synthesis of non-stationary single image textures, and extend it to synthesize texture stacks, as well as generate tileable outputs.

### 3.2.2 Texture Tileability

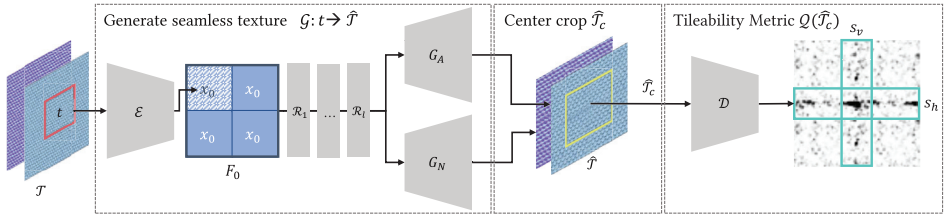
Synthesizing tileable textures received surprisingly little attention in the literature until recent years. Moritz *et al.* [Mor+17] propose a non-parametric approach that is able to synthesize textures from a single example while preserving its *stationarity*, which measures how tileable the texture is. Li *et al.* [Li+20] propose a GraphCuts-based algorithm. They first find a patch that optimally represents the texture, then use graph cuts to transform its borders to improve its tileability. This method allows for synthesizing multiple maps at the same time. Relatedly, Deliot and Heitz [DH19] propose a histogram-preserving blending operation for patch-based synthesis of tileable textures, particularly suited for stochastic textures. The power of deep neural networks for tileable texture synthesis has also been leveraged in the past years. First, Rodriguez-Pardo *et al.* [Rod+19] exploit latent spaces in a pre-trained neural network to find the size of the repeating pattern in the input texture. Then, they use perceptual losses for finding the optimal crop of the image such that, when tiled, looks the most similar to the original image. Also leveraging deep perceptual losses and using Neural Cellular Automata [Mor+20] as an image parameterization, Niklasson *et al.* [Nik+21] generate self-organizing textures that are seamlessly tileable by design, but are limited in resolution and by the quality of the gram matrix perceptual metric used as a loss function [Hei+21]. Bergmann *et al.* [BJV17] achieve tileability in their output textures by training a multi-image GAN in a periodic spatial manifold. Our proposed method does not follow any of these approaches. Instead, we build upon a state-of-the-art single-image texture synthesis method, which is able to generate high-quality and high-resolution images, and extend it to generate textures which are tileable. To this end, we propose a sampling algorithm that finds the input of the generative process that maximizes a novel tileability metric. Our goal is to preserve the input original texture as intact as possible while imposing artifact-free continuity at the intersection of the seams when the texture is tiled. Furthermore, our method has a reduced computational footprint compared to other deep generative texture synthesis methods, as we show in the results section.

### 3.2.3 Deep Internal Learning

Learning patterns from a single image has been studied in recent years, in contexts different to those of texture synthesis. Ulyanov *et al.* [UVL18] show

that single images can be represented by randomly initialized CNNs, and show applicability on denoising or image inpainting problems. A similar method is proposed by Shocher *et al.* [SCI18] for single-image super-resolution. Additionally, single images have shown to be enough for learning low-level features that generalize to multiple problems [ARV19]. Hypernetworks [HDL16] in *Implicit Neural Representations* [Sit+20; Tan+20] also allow for shift-invariant priors over images, which can then used in generative models [DTD21]. Single-image generative models have been explored for domains other than textures. GANs trained on a single image have been used for image retargeting [Sho+19], deep image analogies [Ben+21], or for learning a single-sample image generative model [SDM19; Hin+21; Sho+19; SGK21; VZH20]. These methods, while powerful for natural images, are not well-behaved for textures, as shown in [Liu+20] and [Mar+20]. By introducing inductive biases specially designed for textured images, characterized by their repeating patterns, deep texture synthesis methods achieve better performance in textured images than generic single-image synthesis approaches, which need to account for more globally coherent semantic patterns.

Figure 3.3: Overview of *SeamlessGAN*. A crop $t$ of the input texture stack $T$, is fed to an encoder $E$, which transforms it into a latent space $x_0 \leftarrow E(t)$. We tile this latent space vertically and horizontally, obtaining a latent field $F_0$. $F_0$ is further processed by several residual blocks $R_i, i \in \{1,l\}$.. The resulting latent variables are transformed by two different decoders, $G_A$ and $G_N$, which output 4 copies of a candidate tileable texture $\hat{T}$. By cropping the center part of this texture, we obtain a single of those copies, with seamless borders, $\hat{T}_c$. Additionally, this texture can be analyzed by a discriminator $D$ which provides local estimations of the quality of the synthesis. We introduce a tileability evaluation function $Q$, which, by analyzing two vertical and horizontal search areas $s_v, s_h$, is able to detect artifacts that may arise when tiling the texture. This gives us an estimation of how *tileable* the texture is. This estimation can then be used by a sampling algorithm for generating high-quality tileable textures.




## 3.3 Overview

Our method takes as input an *untileable texture stack* $T$ , which is a layered representation–or SVBRDF [BS12]–of a material used by render engines to virtually reproduce it. A typical texture stack is composed of several maps such as albedo, normals, or roughness. For simplicity, and without losing

generality, we assume that our texture stacks have two maps: an albedo A, and a normal map N, both are RGB images of the same dimensions. There are several methods that generate texture stacks for any material using, for example, smartphones [Des+19; DDB20; Guo+20b; Shi+20; VPS21; Rod+23a], however, these stacks are not tileable by default. Having this data as input, we propose a two-step automatic pipeline to generate tileable texture stacks from arbitrary material input exemplars.

In the first step, described in Section 3.4, we use crops $t$ of the input texture $\mathbf{T}$ to train a Gen-erative Adversarial Network (GAN) able to synthesize novel *untileable* texture stacks $T'$ using adversarial expansion [Zho+18]. This training framework has been shown to provide state-of-the-art results on single-sample texture synthesis, surpassing previous approaches [Uly+16; BJV17; Hen+21]. Thanks to using a GAN, we learn an implicit representation of the texture parameterized in two neural modules: $G$ and $D$. $G : t \rightarrow T'$ is a generator that outputs new untileable textures which double the spatial resolution of the input. $D$ is a discriminator that, thanks to the adversarial framework used for training, is able to distinguish real from fake textures.

In the second step, described in Section 3.5, we produce tileable stacks by means of two key ideas: first, we tile a latent space $\mathbf{x}_0$ of the trained generator $G$, obtaining a novel texture stack showing continuity at the seams intersection $\hat{T}$. Second, we implement a sampling process using the trained discriminator $D$ used as quality metric $Q$ to find an optimally tileable texture stack $T^*$. An overview of our sampling step is shown in Figure 3.3.

In the following, we first describe our GAN architecture, including our proposal to deal with textures stacks. Then, we describe the sampling process to generate tileable ones.

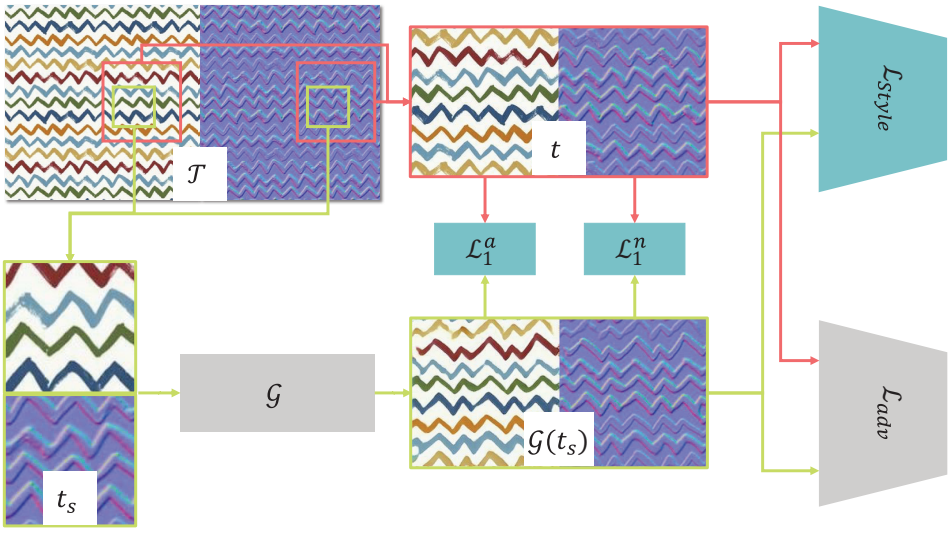
## 3.4 Self-Supervised Texture Synthesis with Adversarial Expansion

For each input texture $T$ we train a GAN, whose generator $G$ is able to synthesize novel examples $T'$. Unlike other GAN frameworks, which take as input a random vector, we use crops $t$ of the original stack as input to guide the generative sampling, such that, $\hat{T}' = G(t)$ for $t \in T$. The training strategy builds upon the work of Zhou *et al.* [Zho+18], which uses *adversarial expansion* to train the network as follows: First, a target crop $t \in T$ of $2k \times 2k$ pixels is selected from the input stack. Then, from that target crop $t$, a source random crop $t_s \in t$ is chosen with a resolution of $k \times k$ pixels. The goal of the generative network will be to synthesize $t$ given $t_s$. This learning approach is fully self-supervised. The generative model is trained alongside a discriminator $D$, which learns to predict whether its inputs are the target crops $t \in T$ or the generated samples $T' = G(t_s)$. Figure 3.4 shows an overview of the training strategy.

### 3.4.1 Network Architecture

SeamlessGAN is comprised of an encoder-decoder convolutional generator $G$ with residual connections, and a convolutional discriminator $D$. So as to be able to synthesize textures of multiple different sizes, the networks are designed to be fully-convolutional. We follow the residual architectural design in [Zho+18], with two extensions: First, building on recent advances in style transfer algorithms, we use *Instance Normalization* [UVL16] before each ReLU non-linearity in the generator and each Leaky-ReLU [MHN13] operation in the dis-criminator. This allows to use normalization for training the networks without the typical artifacts caused by Batch Normalization [Cho+18; Des+18; NK18; Kur+19]. Second, in order to allow for the synthesis of multiple texture maps at the same time, we propose a variation in the generator architecture. In the following, we describe the details of each component, with a particular focus on the elements that are different from [Zho+18].

Figure 3.4: An overview of our training framework for learning to synthesize texture stacks through adversarial expansion. At each iteration, from an input stack $T$ , we randomly crop a target crop $t$, from which we select a source crop $t_s$. The goal of the generator $G$ is to estimate $t$ given $t_s$. We measure the difference between target and estimated crops using a combination of adversarial, pixel-wise, and perceptual loss functions.




#### Generator

The generative architecture proposed is comprised of three main components: An *encoder* $E$, which compresses the information of the input texture $t$ into a latent space, $\mathrm{x}_0 \leftarrow E(t)$, with half the spatial resolution of the input texture. Then,

a set of *residual blocks*, $R_i, i \in \{1,l\}$,, which learn a compact representation of the input texture $x_i \leftarrow R_i(x_i-1)+x_i-1$. Finally, a stack of *decoders* **G** that transforms the output of the last residual block Rl into an output texture $T' \leftarrow \boldsymbol{G}(x_l)$. Residual learning allows for training deeper models with higher levels of visual abstraction and which generate sharper images [He+16; RFB15; Iso+17]. Similar residual generators have been used in unsupervised image-to-image translation problems [Zhu+17a].

Synthesizing a texture stack of multiple maps with a single generative model poses extra challenges over the single map case. Each map represents different properties of the surface, such as geometry or color, resulting in visually and statistically different images. This suggests that independent generative models per map could be needed. However, the texture maps must share pixel-wise coherence, which is not achievable if multiple generative models are used. Inspired by previous work on intrinsic images [Jan+17; YS19; LS18], we propose the use of a generative model that learns a shared representation of all the texture maps, but has different decoders for each of them. The latent space within the generator will thus encode a high-level representation of the texture, which is then decoded in different ways for each different map in the texture stack. Specifically, we train a stack of decoders, $\boldsymbol{G}=\{G_A,G_N\}$, one for each map of the stack (albedo and normals in our case).

### Discriminator

For the discriminator $D$, we use a *PatchGAN* architecture [Zhu+17a; Iso+17; Led+17; Zho+18], which, instead of providing a single estimation of the probability of the whole image being real, it classifies this probability for small patches of it. This architecture has several advantages for our problem. First, it provides local assessments of the quality of the synthesized textures, which we exploit for obtaining high-quality textures. Second, its architecture design allows to provide some control over what kind of features are learned: by adding more layers to $D$, the generated textures are typically of a higher semantic quality but local details may be lost. A comprehensive study on the impact of the depth of $D$ can be found in [Zho+18].

### Loss Function

We train the networks following a standard GAN framework [Goo+14b]. We iterate between training $D$ with a real sample texture stack $T$ and a generated sample $T'$. The adversarial loss $L_{adv}$ is extended with three extra loss terms: $\mathcal{L}_1^a$, $\mathcal{L}_1^n$, and $L_{style}$; corresponding respectively to the pixel-wise distance between the generated and target albedo, normals, and a perceptual loss. The perceptual loss $L_{style}$ is computed as the sum of the perceptual losses between target and generated albedo maps, and target and generated normal maps. We follow the *Gram Loss* described in [GEB16b] as our perceptual loss. We weight the total

style loss by weighting different layers in the same way described in [Rod+19; Zho+18; GEB16b]. Our global loss function thus is defined as:

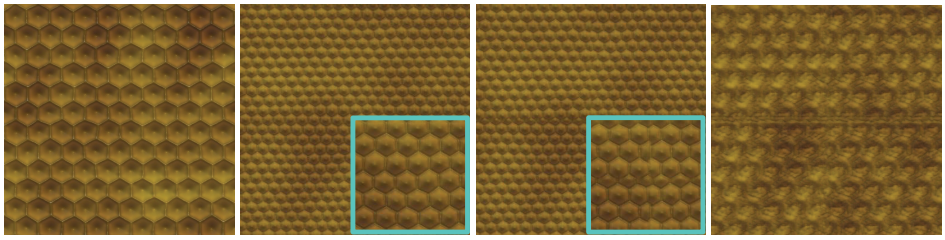
$$\mathcal{L} = \lambda_{adv}\mathcal{L}_{adv} + \lambda_{\mathcal{L}_1^a}\mathcal{L}_1^a + \lambda_{\mathcal{L}_1^n}\mathcal{L}_1^n + \lambda_{style}\mathcal{L}_{style} \tag{3.1}$$

## 3.5 Tileable Texture Stack Sampling

### 3.5.1 Latent Space Tiling

After training, the generator $G$ is able to synthesize novel samples of the texture given a small exemplar of it. Although these novel samples duplicate the size of the input, they are not tileable by default. Previous work [FAW19] showed that by spatially concatenating different latent spaces in a ProGAN [Kar+18] generator, it is possible to generate textures that contain the visual information of those tiles while seamlessly transitioning between the generated tiles. Inspired by this idea, we spatially repeat (horizontally and vertically) the first latent space $\mathbf{x}_0$ within the generative model, obtaining a latent field $\mathbf{F}_0$. This field is passed through the residual layers $R_l$ and the decoders **G** to get a texture stack, $\hat{T}$, that contains four copies of the same texture with seamless transitions between them (see Figure 3.3). At first, the latent field $\mathbf{F}_0$ shows strong discontinuities at the borders between the four copies. Later, as the texture is passed through the network, these artifacts are progressively transformed into seamless borders by the rest of the residual blocks and the decoder of the model. The intuition behind this idea is that, after training the network with the target texture, each of these latent spaces encode low resolution versions of the input in the spatial domain and semantically-rich information in the deeper layers.

Figure 3.5: Impact of the latent field level l on the quality of the output textures $\hat{T} = G(t)$, using the same input $t$. Generating the latent field $\mathbf{F}_l$ early layers ($l$ = 0) generates the best visual results, later layers either create small artifacts ($l$ = 3) or generate unrealistic textures ($l$ = 5). A zoom of the central crop is shown as an inset.



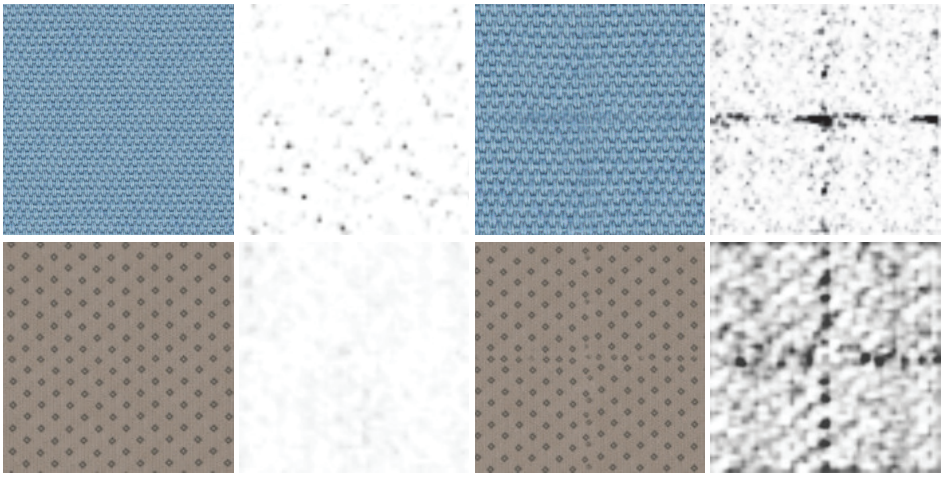
(a) $t$ (b) $\hat{T}, l = 0$ (c) $\hat{T}, l = 3$ (d) $\hat{T}, l = 5$

A seamlessly tileable texture stack $\hat{T}_c$ is obtained by cropping the central region (with an area of 50% of $\hat{T}$) . The predicted texture $\hat{T}$ has 4 × 4 the

resolution of the input crop $t$. Thus, after the cropping operation, $\hat{T}_c$ has twice the resolution of the input $t$.

A key parameter to select is the level $l$ at which to split the generative process. As shown in Figure 3.5, generating the latent field $\mathrm{F}$ by tiling earlier levels of the latent space forces the network to transform $\mathrm{F}$ more times, thus resulting in more seamlessly tileable textures. We confirm the results found in [FAW19] and noticed that generating this latent field at earlier levels of the latent space ($l \in \{0,1\}$) yields the best visual results, by allowing for smoother and more semantically coherent transitions between tiles. We thus tile the output of the encoder $\mathrm{x}_0=\mathrm{E}(t)$, before transforming it by the residual blocks $R_i$.

Figure 3.6: Outputs of $D$ for textures of different qualities. (a) The generated albedos $\hat{T}_1$ are artifact-free on the search areas (b), thus the discriminator $D(\hat{T}_1)$ is fooled to believe $\hat{T}_2$ cannot fool the discriminator, which finds artifacts on the central areas $D(\hat{T}_2)$ (d).



(a) $\hat{T}_1$ (b) $D(\hat{T}_1)$ (c) $\hat{T}_2$ (d) $D(\hat{T}_2)$

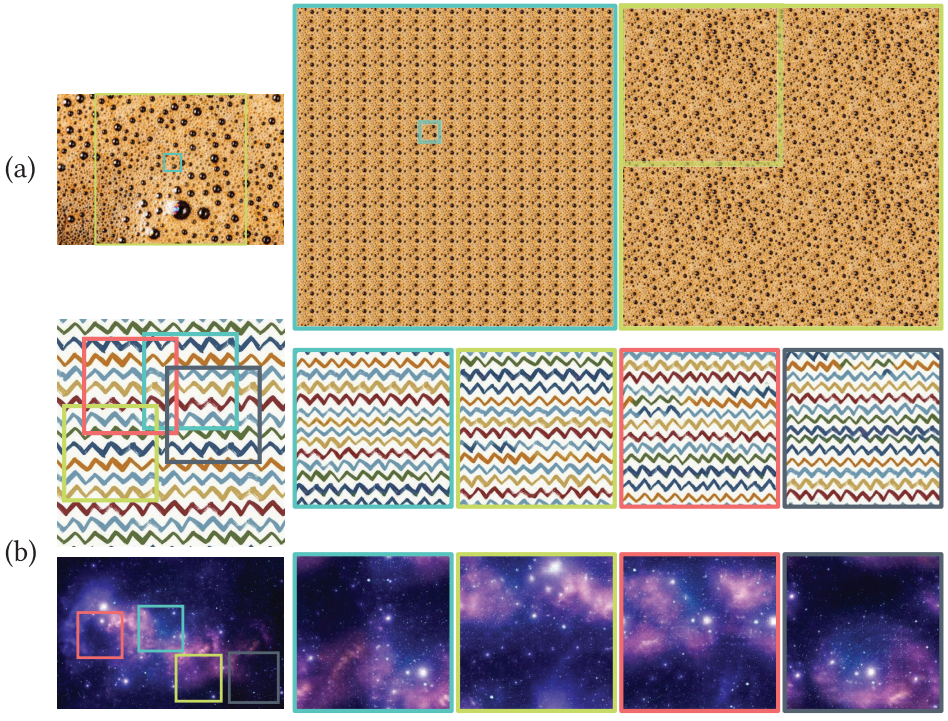
### 3.5.2 Discriminator-guided Sampling

Our strategy to tile the latent space guarantees that the generated texture is a continuous function with smooth transitions between the boundaries of the tiles. However, in contrast to the algorithm in [FAW19], where the latent spaces are drawn from random vectors, ours are encoded representations of input textures. Thus, the selection of the input that the network receives plays an important role on the quality of the output textures. As shown in Figure 3.6, not all the generated textures are equally valid. This selection can be posed as an optimization problem: $t^* = argmax_t Q(G(t))$, where the goal is to find the crop

$t^*$ that maximizes the quality $Q$ of the generated texture $G(t)=\hat{T}_c$. To solve this optimization problem, one option could be to pose it as an optimization in the generative latent space. However, we found that a simpler solution based on sampling already provides satisfactory results. The remainder of this section explains the sampling process and the quality metric.

**Sampling** By using a fully-convolutional GAN, our model can generate textures of any size, hardware being the only limiting factor. This is key for tileable texture synthesis as, even if a given texture is seamlessly tileable, larger textures require fewer repetitions to cover the same spatial area, which ultimately results in fewer repeating artifacts, as illustrated in Figure 3.7 (a). There are two main challenges when finding tileable textures: the input texture needs to contain the distribution of the existing repeating patterns, and the input tile itself must not create strong artifacts when tiling the latent spaces of the generator.

Figure 3.7: *SeamlessGAN* can generate multiple tileable outputs from the same sample $T$. By cropping different parts of T , we can feed the generator G with different inputs $t_c$ (shown on the left), generating tileable textures $\hat{T}_c$ of different sizes. On the example at the top (a), the synthesized images are tiled so they cover the same spatial extent, which shows that, even if the textures are tileable, larger textures generate fewer repeating artifacts. (b) Shows the tile resulting from sampling different parts of the input image.



Our goal is thus to find the largest possible tileable texture stack. To do so, we sample multiple candidate crops for a given crop size $c \in \{c_{min}, c_{max}\}$ is the resolution of the input $T$. We sample crop sizes starting at the largest possible size $c = c_{max}$, and stop when we find a suitable candidate according to the tileability metric $Q$. As shown in Figure 3.7 (b), this sampling mechanism also allows us to generate multiple tileable candidates for a single exemplar if we choose different parts of $T$ as input.

**Discriminator-guided Quality Function,** $Q$ The second component of our sampling strategy is the quality function used to determine whether a stack is tileable or not. We observed that the artifacts appear on vertical and horizontal frames around the center of the textures (Figure 3.6). This is likely caused by strong discontinuities or gradients on the same areas of the tiled latent spaces, which the rest of the generative network fails to transform into realistic textures. Following recent work on generative models [SSK20], we use the discriminator $D$ as a semantic-aware error metric that can be exploited for detecting local artifacts in the generated textures. This can be done in our case because the global loss function contains pixel-wise, *style*, and adversarial losses. The adversarial loss learns the semantics of the texture, whereas the $L_1$ and style losses model color distances or repeated patterns [Rod+19]. This combination of loss functions allows us to balance textural, semantic, and perceptual color properties.

We thus design a quality evaluation function $Q$ that estimates if the generated texture stack $\hat{T}_c$ is *tileable*, looking for artifacts on a central area, $S \in D(\hat{T}c)$ of the discriminator. This area is composed of two regions, $S = s_v \cup s_h$, where $S_v$ is a vertical area, and $S_h$ is an horizontal one, both centered on the output of the discriminator (Figure 3.3). The function $Q$ leverages the fact that $D$ outputs 0 when it believes a patch to be synthetic. However, as the values are sample-dependent, we establish a threshold $\tau \in \mathbb{R}$ using the values of the rest of the image $S$ as a reference $\tau = \gamma \cdot \min(S_r)$, where $S_r = D(\hat{T}_c) \cap S^-$ is the remaining part of the image, and $\gamma \in \mathbb{R}$ is a threshold that allows to control the sensitiveness of $Q$. Consequently, $Q(\hat{T}_c)$ is 1 if $\min(s_v)$ and $\min(s_h)$ are greater or equal than $\tau$, considering the texture as *tileable*, and 0 otherwise. The goal of $Q$ is to estimate whether or not the central areas, where discontinuities may be present, are as realistic as the rest of the texture. It works as a classification function which returns a high value if the texture is identified as real by the discriminator in such areas. This allows us to create a sampling strategy that distinguishes between images with local artifacts from seamless textures. By using minimum values instead of the mean estimation of the discriminator, our quality metric focuses on detecting artifacts which may arise during the latent space tiling operation.

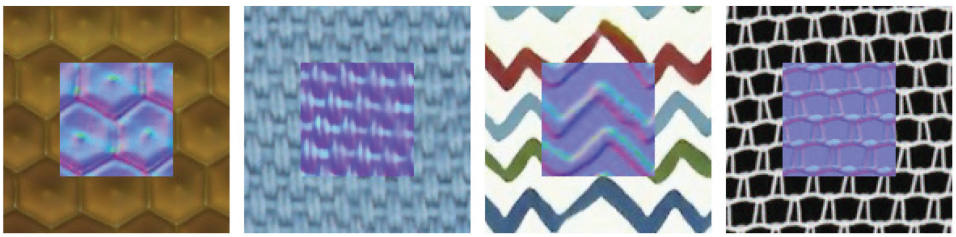
## 3.6 Implementation Details

As described in Section 3.4, we follow a standard GAN training framework, by iterating between training the discriminator and the generator. We use a batch

size of 1, and an input size of $k = 128$. All weights are initialized by sampling a Gaussian distribution $N$ (0, 0.02), following standard practice [Zhu+17a]. $G$ has $l$ = 5 residual blocks with ReLU activations, and $D$ is comprised of 5 convolutional layers with Leaky-ReLU non-linearities, and a Sigmoid operation at the end. We use a stride of 2 for the downsampling operations in $E$ and transposed convolutions [LSD15] for upsampling in the decoders **G**. We weight each part of the loss function as: $\lambda_{adv}$=$\lambda_{style}$=1, and $\lambda_{\mathcal{L}_1^a} = \lambda_{\mathcal{L}_1^n} = 10$ .

The networks are trained for 50000 iterations using Adam [KB15], with an initial learning rate of 0.0002, which is divided by 5, after iterations 30000 and 40000. Aside from random cropping, we do not use any other type of data augmentation method. The models are trained and evaluated using a single NVIDIA GeForce GTX 1080Ti. Even if training takes around 40 minutes for each texture stack, once trained, the generator can generate individual new samples in milliseconds, before checking for tileability. We use PyTorch [Pas+19] as our learning framework. To accelerate the training process, we leverage mixed precision training and automatic gradient scaling [Mic+18]. Every operation in the training pipeline is done natively in GPU using Torchvision [MR10]. These optimizations allow us to train the networks one order of magnitude faster than previous methods [Zho+18]. The input textures $T$ tested in this chapter have, on average, 500 pixels in their larger dimension, for which our method can generate tiles of at most 1000 pixels. We use Photometric Stereo [Ike81] for computing the normals of fabric textures, and artist-generated normals for the other samples.

Figure 3.8: Crops of synthesized textures using our method. We observe that pixel-wise coherence between maps is preserved.



We tile the latent space at its earliest level ($F_l$,$l = 0$), as it provides the best quality results, as shown in Section 3.5.1. For identifying the tileability of textures, we use a search area *S* that spans a 20% of each spatial dimension of the textures. For all the results shown, $\gamma$ = 1. We choose $c_{min}$ = 100 pixels, and $c_{max}$ equal to the resolution of the whole input texture. This means that for the level of $c_{max}$, only one texture is sampled. For each $c < c_{max}$, we sample 3 different random crops. Typically, for the results shown in this chapter, a tileable texture stack is found at high-resolution crop sizes, $c \geq c_{max}$ −10, thus needing to sample and evaluate less than thirty random crops before finding a satisfactory solution. This entire sampling procedure takes less than five minutes.

## 3.7 Experiments

In this section, we evaluate two of our main contributions: first, in Section 3.7.1 we study the impact of the design choices of the generator G to synthesize a multi-layer texture stack, and second, the quality of the tileable texture synthesis through several ablation studies. We study the inter-map consistency in Section 3.7.2, and the impact of the loss function in Section 3.7.3. Finally, we compare our method with methods on tileable texture synthesis (Section 3.8).

### 3.7.1 Generator *G* Design

One of the main challenges when synthesizing texture stacks using a single generator is to preserve the low-level details of each of the texture maps whilst maintaining the local spatial coherence between them; if this coherence is lost, renders that use the synthesized maps will show artifacts or look unrealistic. We propose two different variations of the single-map generative architecture $G$ presented in [Zho+18], each of which makes different assumptions on how the synthesis should be learned taking into account the particular semantics and purpose of each map. A diagram of each proposed architecture is shown in Figure 3.10. For a fair evaluation, we follow the criteria that both networks must have approximately the same number of trainable parameters.

Our baseline, $G_1$, treats the texture stack as a multiple-channel input image, and entangles every texture map in the same layers. It assumes that the maps in the stack share most of the structural information and, as such, there is no need to generate them separately. Thus, the last layer in the decoder outputs every texture map. Our proposed alternative architecture, $G_2$ finds a shared representation of each texture map, but has a separate decoder for each of them. As such, the residual blocks are shared for all the texture stack, but each decoder can be optimized for the semantics and statistics of each particular map.

To quantitatively evaluate which architecture produces the highest quality output we compare the original texture with the generated one using standard metrics: SSIM, Si-FID, and LPIPS. The *Structural Similarity Index Measure (SSIM)* [Wan+04] is a perceptual-aware metric, working on the pixel space, that measures the similarity in structural information, and may be appropriate to evaluate synthesized textures. The Si-FID [SDM19] is a single image extension of the *Fréchet Inception Score (FID)* [Heu+17], which measures the difference in deep latent statistics between natural and artificially generated images. Finally, we use the *Learned Perceptual Image Patch Similarity (LPIPS)* [Zha+18] as a perceptual distance metric in deep image spaces. This metric is widely used for evaluating generative models [Kar+20b; Hua+18b; Cha+19; Alm+18; Mar+20].

The baseline $G_1$ shows more artifacts than $G_2$, most likely due to the fact that the generation parts of the network are not fully separated. A quantitative evaluation is shown in Table 3.1, showing that $G_2$ outperforms $G_1$ from perceptual and statistical standpoints. Those metrics require the input and target images to have the same spatial dimensions. To obtain this, we crop the 50% center area of each generated stack, which doubles the dimensions of the inputs due to the adversarial expansion approach we follow, and compare them with the input textures. In summary, $G_2$ allows to generate textures that better preserve the properties of the input images without any additional computational cost and without requiring to modify the loss function, or the discriminator design.

Table 3.1: Quantitative comparison between our variations of the generator $G$. We show the average results on different metrics across different texture stacks, separated by maps. As shown, $G_2$ outperforms its baseline across metrics and texture maps. $G_1$ only yields better scores at the Si-FID metric on the normal map. Higher is better for SSIM, while lower is better for Si-FID and LPIPS

| | | SSIM [Wan+04] ↑ | Si-FID [SDM19] ↓ | LPIPS [Zha+18] ↓ |
|---|---|---|---|---|
| G1 | Albedo | 0.2774 | 0.3462 | 0.4826 |
| | Normals | 0.2364 | **0.3636** | 0.4870 |
| G2 | Albedo | **0.3123** | **0.3275** | **0.4377** |
| | Normals | **0.2921** | 0.4019 | **0.4340** |

### 3.7.2 Inter-map Consistency

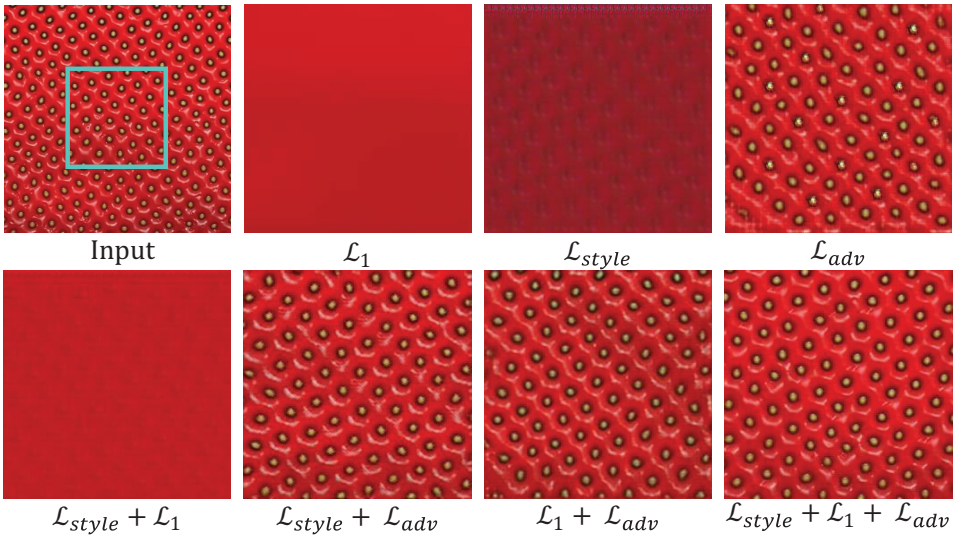
Whilst our generators $G$ can generate high-quality tileable pairs of albedo and normal maps, there is no guarantee that those maps are pixel-wise coherent, as no part in the loss function explicitly accounts for this relationship. The $L_{style}$ loss is computed separately for each map in the texture stack, which may generate non-coherent gradients. Furthermore, our architecture of choice $G_2$ separates the decoding of each map in different parts of its architecture, which may hinder the generation of spatially-coherent maps. Nevertheless, we show that this is not a problem in practice. Figure 3.8 shows crops of our synthesized texture maps. It can be seen that pixel-wise coherency between maps is preserved even for challenging geometric structures. We argue that the role of the discriminator is key to detect inter-layer inconsistencies by yielding lower probabilities for non-coherent maps. Even if separating the decoders may increase the risk of incoherent texture maps, this coherence is forced by the discriminator during training. To test this empirically, for the textures in Figure 3.8, we fed the discriminator with a crop of the original stack, a crop with the normals translated 5

px, and translated 100 px, for which we obtain average values of 0.99, 0.35, and 0.32, respectively. This suggests that, since during training, the discriminator receives the entire texture stack at once, it learns to identify spatial inconsistencies between maps.

### 3.7.3 Loss Function Ablation Study

As described in Section 3.4, the generator $G$ is trained to minimize a loss function, which is comprised of adversarial, perceptual and pixel-wise components. Each of these have different impacts on the output synthesis. We hereby study the impact of each of these components.

Figure 3.9: Ablation study on the impact of the loss function on the quality of the synthesized textures. We evaluate each network using the same input, marked using a blue box. As shown, training $G$ using the full loss function yields the best results. Each component is weighted by a $\lambda$, as specified in Section 3.6.

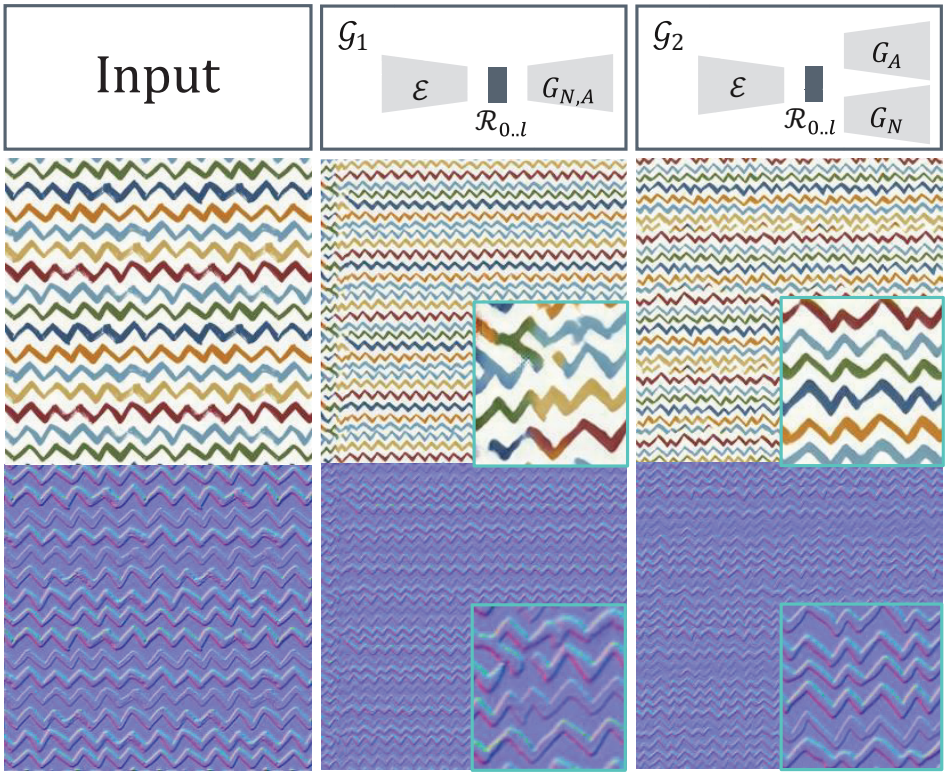



To better isolate the impact of each metric, we perform this study using a limited generator that only outputs albedo maps. As we show in Figure 3.9, the adversarial loss $L_{adv}$ provides high-level semantic consistency, the $L_1$ norm acts as a regularization method which removes artifacts, while the perceptual loss $L_{style}$ adds additional details to fully represent the texture. Similar findings are reported in [Zho+18]. None of these loss functions yield compelling or artifact-free results on isolation, being the adversarial loss the most important factor of the global function.

## 3.8 Results and Comparisons

Most tileable texture synthesis methods have not shown results in synthesizing texture stacks. While adding this capability is reasonably easy for some *non parametric* methods [Li+20; Rod+19], others require major changes in their design. In particular, as we have shown in Section 3.7.1, expanding the architecture of deep learning methods for generating more than one map requires special attention to balance the map's interdependence with the network's capacity and expected quality. Therefore, in this section, in order to be able to compare our method with other works, we use a reduced version of our generator that only outputs a single albedo map.

Figure 3.10: A comparison of the results of our proposed architectures. Separating the decoders of the network for each map ($G_2$) outperforms the joint-decoder baseline ($G_1$) for both texture maps in the stacks. It generally allows for more varied albedo maps, with a more correct structure and style; as well as more accurate and sharper normal maps. The outputs in this figure are not tiled, for improved visibility of the artifacts created by $G_1$. In blue, we show an up-sampled crop of the output textures.



**Qualitative Analysis** First, we compare with the Texture Stationarization algorithm proposed in [Mor+17] using examples taken from their own dataset, as their code is not publicly available. A key difference between both methods is that, while their method aims to maxi-mize the stationarity properties of the textures using external metrics, our method learns them from the texture itself, using a self-supervised approach. This results in our method better preserving the content of the original input. As shown in Figure 3.11, our method shows compelling results for the kind of textures shown in their paper. In the *fence* example, our methods better preserves the vertical straight lines. For the *wall* example, both methods shows compelling results, capturing a different repetition pattern.

As Moritz *et al.*'s dataset contains mainly textures of human-made environments, we gather a different set of images of greater variety in their regularity and content. Figure 3.12 shows a comprehensive comparison with the other methods. The work by Li *et al.* [Li+20], based on *Graph-Cuts*, works reasonably well if the transformation can be done locally at the seams but fails when the required changes are global, as happens in the *basket*. The *Repeated Pattern Detection* algorithm proposed by Rodriguez-Pardo *et al.* [Rod+19] is not able to handle many of these challenging cases, which do not represent grid-like textures, as in the *bananas* or *stars*. The Periodic Spatial GAN (PSGAN) proposed by Bermann *et al.* [BJV17] generates artifacts, not maintaining the content of the texture, as in the *daisies* or the *hieroglyph* examples. A similar effect is observable in the method of [Nik+21], which uses self-organizing representations through neural cellular automata. The output of the method is seamless but the integrity of the original texture is not preserved in many cases, for example, in the *hieroglyph* or the *stars*.

Our method favors keeping high-level and bigger semantic structures of the textures, resulting in larger samples with more variety. The modification applied to the texture is the minimum required to make it seamlessly tileable. It provides high-quality outputs for regular textures with uneven illumination and perspective distortion, such as the *basket*, textures which are greatly irregular, such as the *bananas*, and stochastic ones, like the *strawberry*. While non-parametric synthesis methods are typically computationally cheap, obtaining results in less than one minute [Li+20; Rod+19; DH19], at the cost of quality and generality, parametric methods are more expensive and slower. Our method is competitive in computational cost when compared to other parametric synthesis methods. Our entire training and sampling process takes less than 45 minutes. In comparison, PSGAN [BJV17] needs 4 hours to train, the work by Zhou *et al.* [Zho+18] requires 6 hours, and the method of Niklasson *et al.* [Nik+21] takes 35 minutes to generate textures limited to 200 × 200 pixels. On the contrary, our method is not limited by the size of the input texture and can synthesize tiles of any size thanks to the fully-convolutional architecture, hardware being the only limiting factor.

Figure 3.11: A comparison with the work on Texture Stationarization by Moritz *et al.* [Mor+17], using textures from their dataset. The results are tiled 2 × 2 times to help visualization.

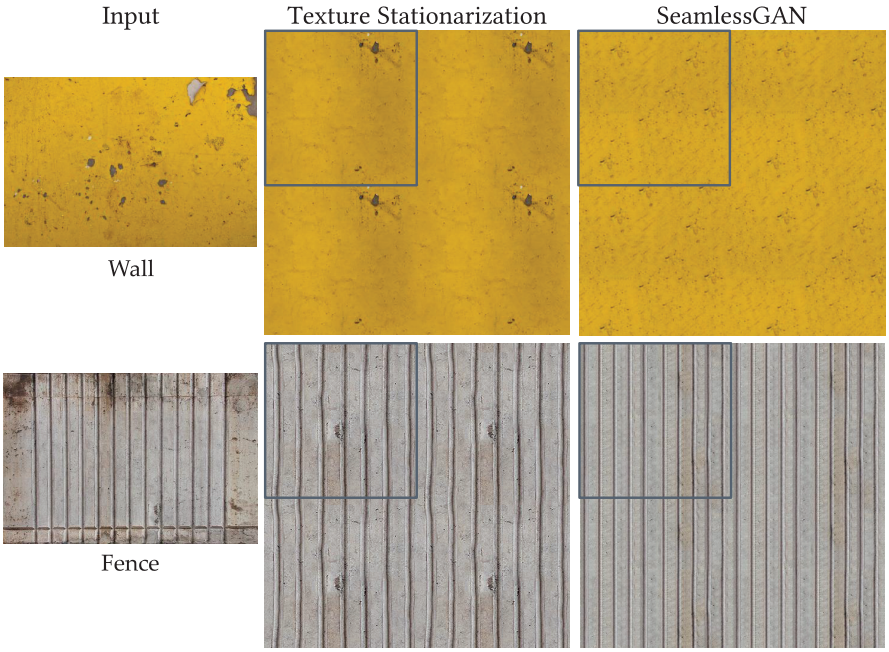



Table 3.2: Quantitative comparison between different methods. We show the average results on dif-ferent perceptual metrics across a variety of textures.As shown, SeamlessGAN consistently outperforms its counterparts in every studied metric. We tile the outputs until they match the spatial resolution of the input examples. Higher is better for SSIM, while lower is better for Si-FID and LPIPS. We use a color code to highlight **best** and **worst** cases.

| | SSIM [Wan+04] ↑ | Si-FID [SDM19] ↓ | LPIPS [Zha+18] ↓ |
|---|---|---|---|
| Deloit *et al.* [DH19] | **0.1424** | 1.3471 | **0.6207** |
| Li *et al.* [Li+20] | 0.2086 | 0.9529 | 0.5818 |
| Moritz *et al.* [Mor+17] | 0.1968 | 0.7620 | 0.5171 |
| Rodriguez-Pardo *et al.* [Rod+19] | 0.2144 | 1.2958 | 0.5137 |
| Bergmann *et al.* [BJV17] | 0.1723 | **1.4355** | 0.5624 |
| Niklasson *et al.* [Nik+21] | 0.1753 | 1.3171 | 0.5328 |
| **SeamlessGAN** | **0.2341** | **0.6311** | **0.4792** |

**Quantitative Comparison** Following a similar evaluation scheme as proposed in Section 3.7.1 for measuring the differences between input and synthesized images, a quantitative evaluation is shown in Table 3.2. We use Moritz's dataset for a fair comparison with every method. The metrics we use for this experiment require that the input and synthesized images have the same resolution. To achieve this, and, in order to account for artifacts in the borders of the generated textures, we tile the synthesized images until they cover the same resolution as their corresponding inputs.

Interestingly, methods based on patches [RB19; Mor+17; Li+20] obtain better SSIM and Si-FID scores than previous deep learning-based methods [BJV17; Nik+21]. This difference is not seen in the LPIPS metric. This suggests that patch-based methods preserve better the structure of the input textures than previous neural parametric models. Our method, by combining a variety of loss functions, allows for better preservation of the style and semantic content of the generated textures. Furthermore, our latent space manipulation algorithm allows for seamless borders between tiles, outperforming both previous non-parametric and parametric methods in perceptual and structural similarity. The magnitude of the Si-FID metric varies significantly between the values obtained in Table 3.1 and Table 3.2, which indicates that this metric may be overly sensitive to the global statistics of the dataset.

**Limitations and Discussion** Our method is inherently limited by the capabilities of the adversarial expansion technique to learn the implicit structure of the given texture. That is, if the input does not show enough regularity, the adversarial expansion fails, as shown in Figure 3.13. It is interesting to see that the synthesis of the *dotted* texture fails to reproduce the larger dots, as they are very scarce. However, the remaining structure is very well represented.

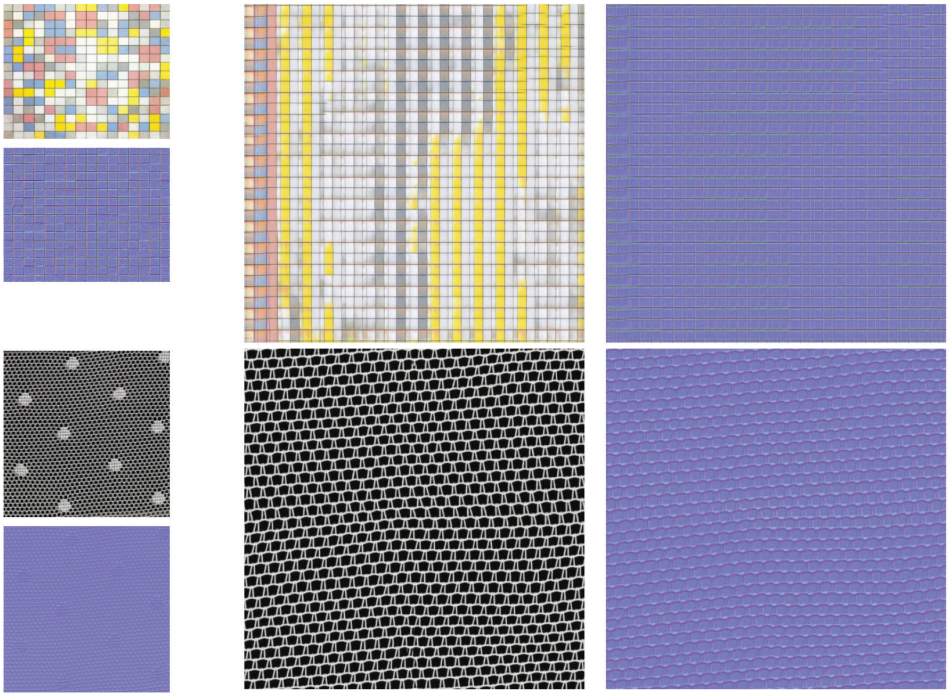
Tiling single texture maps, even if they contain seamless borders, may generate perceptible repetitions. This is an intrinsic limitation of any single sample tileable texture synthesis. Approaches such as *Wang Tiles* [Wan+04] can tackle these limitations, but have additional disadvantages like increased memory and run-time consumption, rendering them less practical for real-time applications. Alternatively, procedural methods [Gue+20] are the most effective way to generate material samples preserving textural properties at multiple scales; however, present several limitations in the range of materials that can be generated to make them fully usable.

## 3.9 Conclusions and Future Work

In this chapter, we have proposed a deep parametric texture synthesis framework capable of synthesizing textures into tileable single-tiles, by combining recent advances in deep texture synthesis, adversarial neural networks, and latent spaces manipulation. Our results show that our method can

Figure 3.12: Comparison with previous methods. On the left column, we show the input textures. From left to right, the synthesized results of Histogram Blending, by Deloit *et al.* [DH19], the GraphCuts algorithm by Li *et al.* [Li+20], Repeated Pattern Detection by Rodriguez-Pardo *et al.* [Rod+19], PSGAN by Bergmann *et al.* [BJV17], Self-Organizing Textures by Niklasson *et al.* [Nik+21]; and ours. Outputs are tiled a similar number of times (at least twice in each dimension) for better visualization. Our method generally captures better the overall structure of the texture, while providing seamless and semantically coherent borders, for enhancing tileability. From top to bottom, we sample $c$ = 4, 12, 5, 17, 13, 1, 19 and 43 input crops before obtaining a tileable texture.

Figure 3.13: A limitation of our texture synthesis algorithm: Left: Input texture stacks; right: synthe-sized stacks. The network fails to replicate the pattern if the occurrence is not frequent enough, as we can see in the base color of these two examples. On the other hand, the synthesized normal map is consistent as its repetitive structure is seen frequently by the network.



generate visually pleasing results for images with different levels of regularity and homogeneity. This work is the first method capable of exploiting properties of deep latent spaces within neural networks for generating seamless textures, and opens the opportunity for end-to-end tileable texture synthesis methods without the need for manual input. Comparisons with previous state-of-the-art methods show that our method provides results which better maintain the semantic properties of the textures, while being able to synthesize multiple maps at the same time.

Our method can be improved in several ways. First, the adversarial expansion framework, while powerful, it has some potential pitfalls that hinder its widespread applicability. The same neural architecture is used for every texture but, as discussed, different choices on the architecture make different assumptions on the nature of the textures. We have proposed a generic architecture that works well for many examples, but recent advances in Neural Architecture Search [Mel+21]

may provide better priors on the optimal neural architecture to use for each sample. Furthermore, each texture synthesis network is trained from scratch. This is not only computationally costly, but learning to synthesize one texture may help in the synthesis of other textures, as shown by [Liu+20]. Fine-tuning pre-trained models for generating new textures may provide cheaper syntheses. Besides, our discriminator, while capable of detecting local artifacts, provides little control for separating such artifacts and global semantic errors. Recent work on image synthesis may provide guidance onto designing better discriminative models [SSK20], training procedures [Yun+19; Sin+21], image parametrizations [Wan+20a; HZ21; Nik+21] or perceptual loss functions [Hei+21].

Second, we proposed a synthesis solution based on manipulating latent spaces within the generative model, but explicitly training the network to generate tileable textures may provide better results than our approach. Besides, our sampling procedure could be extended for better selection of textures, by comparing histograms of the activations of the discriminator on the selected central area, instead of simply comparing minimum values.

Finally, our method has the advantage of being fully automatic, however, pre-processing the texture images so they are more easily tileable can help the synthesis process. For example, automatically rotating the textures so their repeating patterns are aligned with the axes was studied by [Rod+19]. Powerful methods for artifact [Dek+15] and distortion [Li+19] removal could be applied as a pre-processing operation to the input textures before training the generative models, or as additional components to their loss function for improving tileability or homogeneity.

### 3.A Additional Details

**Training details:** We use PyTorch [Pas+19] as our learning framework, and Adam [KB15] as our optimization algorithm. For it, we use an initial learning rate of $\alpha$ = 0.0002, which is divided by 5, after iterations 30000 and 40000. The networks are trained for a total of 50000 iterations, using a batch size of 1.The momentum parameters are kept to the default values of $\beta_1$=0.9 and $\beta_2$=0.999, and $\epsilon$=$10^{-8}$. We additionally apply an $L_2$ regularization of $10^{-5}$ to the optimization of both networks, which may help obtain better synthesized images [Kur+19]. This configuration is the same for the training of both the discriminator and the generator. In contrast to other work in unsupervised image generation [Zhu+17a], both networks are updated after each iteration. The models are trained and evaluated using a single NVIDIA GeForce GTX 1080Ti, leveraging half-precision training and automatic gradient scaling [Mic+18]. Evaluation is done in half precision for faster inference and reduced memory consumption. Using reduced numeric precision does not yield worse synthesized textures.

**Network architecture:** We follow closely the architectures defined in [Zho+18]. For the **Generator** G, we use replication padding as the padding method for the entire architecture, ReLU non-linearities, and Instance Normalization [UVL16] before each non-linearity. This network receives textures of 3 × k × k pixels, which are first processed by an encoder $E$, which transforms it to a $256 \times \frac{k}{4} \times \frac{k}{4}$ latent vector $x_0$. Following standard practice on residual convolutional networks [He+16], the first layer of $E$ is comprised by 64 7 × 7 convolutional filters. The rest of the network, unless stated otherwise, uses standard 3 × 3 kernels. This latent vector is further processed by 5 residual blocks $\mathbf{x}_i \leftarrow R_i(\mathbf{x}_{i-1}) + \mathbf{x}_{i-1}$, which maintain the latent vector dimensions. Then, a decoder **G** takes $x_5$ and, using three convolutional layers with transposed convolutions [LSD15], followed by ReLU operations, it returns the estimated texture, with 2k × 2k resolution. As in the encoder, the last layer of **G** contains 64 7 × 7 convolutional filters. There are as many decoders **G** as individual maps in the texture stack. When tiling the latent space $x_0$, this doubles it spatial resolution, thus becoming a $256 \times \frac{k}{2} \times \frac{k}{2}$ latent vector. As such, once processed by the residual blocks and the decoders, the network outputs a 4k × 4k texture stack.

For the **discriminator** $D$, we simply follow a *PatchGAN* architecture [Zhu+17a; Iso+17; Led+17; Zho+18], which outputs local estimations of the probability of the input image being real or generated. We use 4 blocks of layers, each comprised of, in order, a convolutional layer with 4 × 4 kernels, an Instance Normalization [UVL16] layer, and a Leaky ReLU non-linearity [MHN13]. We use a stride of 2 in the first and last layers to decrease the dimensions of the latent representations. The first block of layers contains 64 trainable kernels, which are doubled after each block. The last layer thus contains 512 trainable filters. In the last layer, we remove the normalization operation and substitute the non-linearity by a Sigmoid operation, to transform the output into the (0, 1) range. The discriminator receives $2k \times 2k$ resolution stacks, and outputs a $1 \times \frac{k}{2} \times \frac{k}{2}$ resolution estimation map.

## 3.B Additional Comparisons, Results and Ablations

In this section, we outline the implementation details used for our comparison with other tileable texture synthesis methods, which included Histogram-Preserving Blending by Deliot and Heitz [DH19], Graph-Cuts synthesis by Li *et al.* [Li+20], Repeated Pattern Detection by Rodriguez-Pardo *et al.* [Rod+19], Periodic Spatial GAN by Bergmann *et al.* [BJV17] and Self-Organizing Textures by Niklasson *et al.* [Nik+21]. We also provide further results in Figure 3.15 and an ablation study on the size of the model architectures in Figure 3.14.

**Histogram-Preserving Blending** We use the implementation provided by the authors. The only hyperparameter to be changed is the blending border size, which is set to the default value of 33% for all the textures.

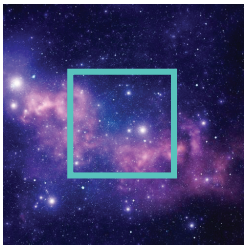
**Graph-Cuts synthesis** To allow for comparison between single-map texture synthesis methods, we run this method to output only the albedo map, using the default values of $\lambda_d = 1$, a ratio of 0.64, a crop ratio of 0.6 and a margin size of 72 pixels. The images are not rescaled to any input or target size, instead, we use their original resolution.

**Repeated Pattern Detection** This paper proposed the use of some preprocessing methods for improving the regularity of the input images, which were not used for our comparison. We use their default hyperparameter values, including $\delta$ = 0.65, $\lambda$ = 0.8, and the deep perceptual loss weighting proposed in [GEB15a].

**Periodic Spatial GAN** This method can handle both single-image and multiple image datasets. We are interested in the former, and, as such, each GAN is trained using only one texture sample. We train their model using Theano [Al-+16], and use their default configuration of a learning rate of 0.0002, a $\beta_1$ = 0.5 as the momentum term for the optimization algorithm, a batch size of 25, and train their network for 10 epochs, each comprised of 25000 training steps. The network parameters and sizes are exactly as defined in the paper.

**Self-Organizing Textures** This method is computationally expensive and, as such, it requires that the input textures are rescaled to 128 × 128 pixels. We train their method using Tensorflow [Aba+16] for 5000 iterations. We use the default training configuration, with a starting learning rate of 0.002, which is then reduced by half at iterations 2000 and 4000. We do not modify the image parametrization design or loss function in any way.

Figure 3.14: An ablation study on the influence of the number of layers in each model in the quality of the generated outputs. We evaluate each network using the same input, marked using a blue box. We compare evaluate the influence of using discriminators with $L \in \{3, 4, 5\}$ layers, marked as $D_L$ and generators with $L \in \{4, 5, 6\}$ residual blocks, marked as $G_L$. We show the configuration we use for every experiment in this chapter using a green box. On the bottom of each image, we show the training time for 50000 iterations. As shown, using a discriminator with 3 layers yields repetitive large-scale patterns, due to its reduced receptive field. Using $D_L = 5$ yields marginal improvements with respect to $D_L = 4$, with the cost of increased runtimes. In terms of $G_L$, we find that there are no significant visual differences between using 5 or 6 residual blocks, but we favor the former due to its reduced training time.



Input

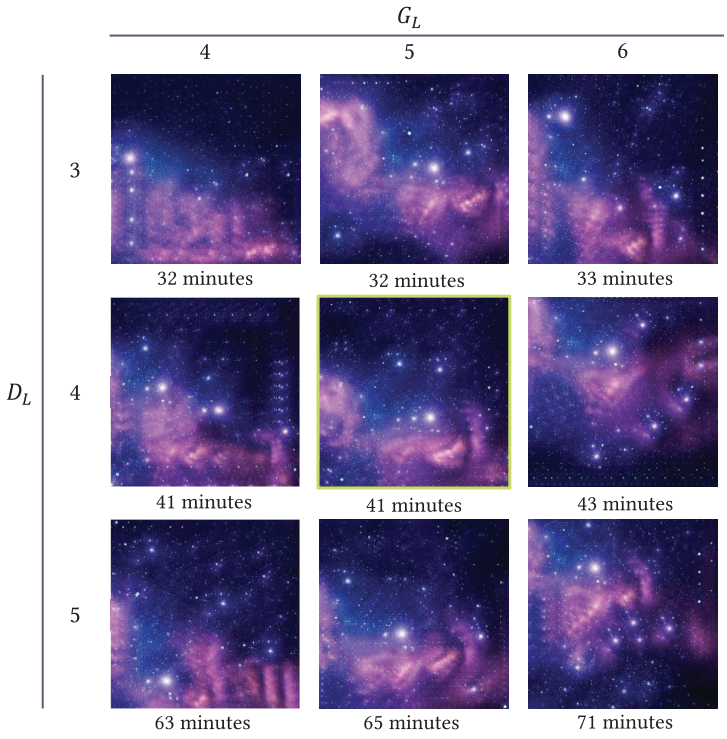


Figure 3.15: More examples on the capabilities of SeamlessGAN on generating multiple tileable textures from a given input. On the left, we show the input sample $T$ and four crops used for generating tileable maps. On the middle column, the outputs of the generative model are shown. On their right, the same textures are tiled 2 × 2 times for visualization purposes.

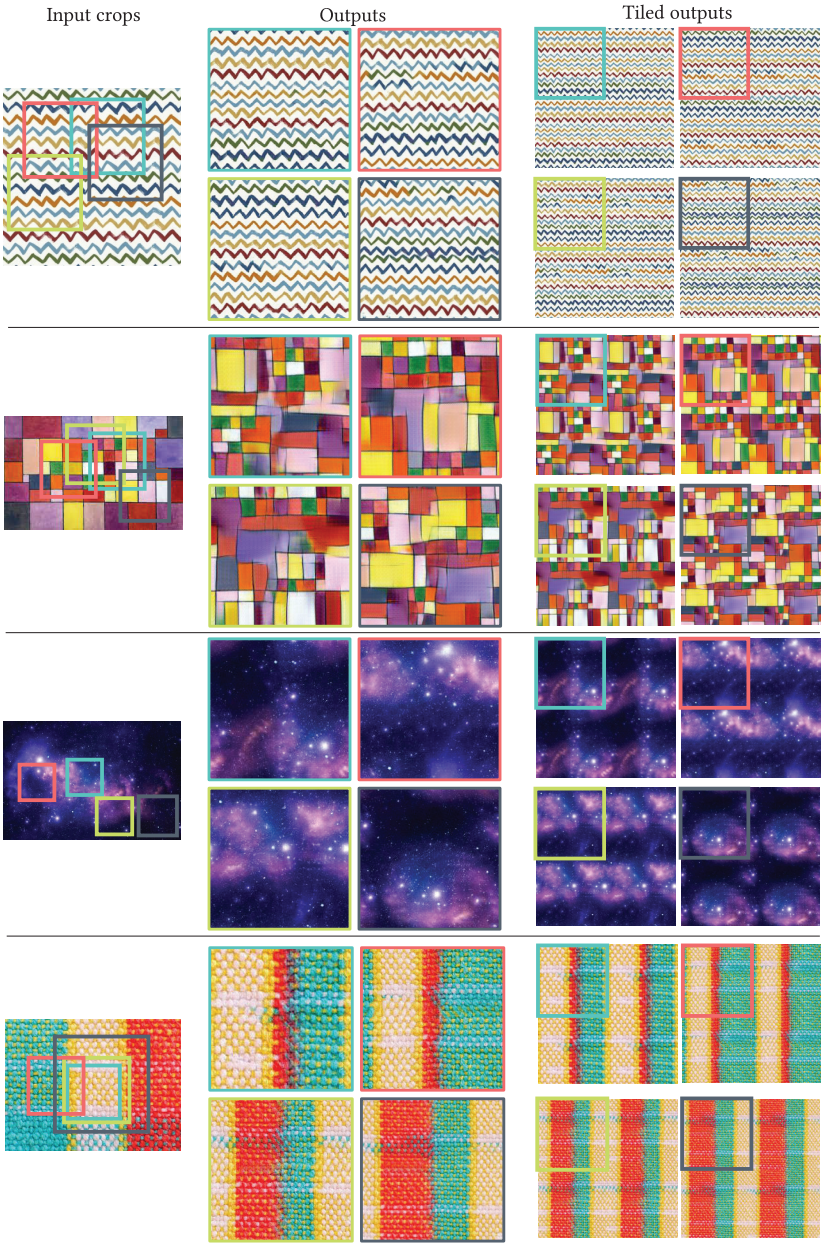


Figure 3.16: Additional comparison with previous methods. From left to right, top to bottom: Input texture, the results of Histogram Blending, by Deloit *et al.* [DH19], the GraphCuts algorithm by Li *et al.* [Li+20], Texture Stationarization by Moritz *et al.* [Mor+17], Repeated Pattern Detection by Rodriguez-Pardo *et al.* [Rod+19], PSGAN by Bergmann *et al.* [BJV17], Self-Organizing Textures by Niklasson *et al.* [Nik+21]; and ours. Outputs are tiled a similar number of times (at least twice in each dimension) for better visualization. Our method generally captures better the overall structure of the texture, while providing seamless and semantically coherent borders, for enhancing tileability. From left to right, we sample $c$ = 27, 36, 29, 1, 17, and 7 input crops before obtaining a tileable texture.