PolyPublie
Polytechnique Montréal

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

| | |
|---|---|
| **Titre:** Title: | Example-Based Synthesis of 2D and Solid Textures |
| **Auteur:** Author: | Jorge Alberto Gutierrez Ortega |
| **Date:** | 2020 |
| **Type:** | Mémoire ou thèse / Dissertation or Thesis |
| **Référence:** Citation: | Gutierrez Ortega, J. A. (2020). Example-Based Synthesis of 2D and Solid Textures [Thèse de doctorat, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/4224/ |

**Document en libre accès dans PolyPublie**
Open Access document in PolyPublie

| | |
|---|---|
| **URL de PolyPublie:** PolyPublie URL: | https://publications.polymtl.ca/4224/ |
| **Directeurs de recherche:** Advisors: | Thomas Hurtut, Julien Rabin, & Bruno Galerne |
| **Programme:** Program: | Génie informatique |

**Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal**
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal

https://publications.polymtl.ca

**POLYTECHNIQUE MONTRÉAL**
affiliée à l'Université de Montréal

# Example-Based Synthesis of 2D and Solid Textures

**JORGE ALBERTO GUTIERREZ ORTEGA**
Département de génie informatique et génie logiciel

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Génie informatique

Février 2020

© Jorge Alberto Gutierrez Ortega, 2020.

**POLYTECHNIQUE MONTRÉAL**
affiliée à l'Université de Montréal

Cette thèse intitulée :

**Example-Based Synthesis of 2D and Solid Textures**

présentée par **Jorge Alberto GUTIERREZ ORTEGA**
en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de :

**Christopher J. PAL**, président
**Thomas HURTUT**, membre et directeur de recherche
**Julien RABIN**, membre et codirecteur de recherche
**Bruno GALERNE**, membre et codirecteur de recherche
**Pierre POULIN**, membre
**Jean-François LALONDE**, membre externe

# ACKNOWLEDGEMENTS

I gratefully acknowledge the funding I received from the Mexican National Council of Science and Technology (CONACYT), the Natural Sciences and Engineering Research Council of Canada (NSERC), and the French National Center for Scientific Research (CNRS).

My deep gratitude goes to my supervisors, Thomas Hurtut, Bruno Galerne, and Julien Rabin for their guidance, trust, and support. I am always inspired by their intelligence and curiosity.

I wish to acknowledge the members of my thesis jury, Christopher Pal, Pierre Poulin, and Jean-François Lalonde for the time and effort spent reading and evaluating my work, and for their valuable insights.

I would like to thank Farida Cheriet, Philippe Debanné, and Christopher Pal for accommodating me in their labs. I also greatly value the assistance and amiability of Brigitte Hayeur, and the technical support of Louis Malo, Jonathan Brien, Jean-Marc Chevalier.

I am also grateful with the professors at Polytechnique Montréal, University of Montréal, and University of Bordeaux, whose classes enabled me to complete this project.

I am indebted to the team and community of *Thèsez-vous* which certainly reduced the time (and agony) to write my dissertation. Working there reminded me the value of community and empathy. In the same spirit, I thank the staff of *Oui mais non* for providing a refreshing spot to write.

During this time I had the chance to meet some amazing people. I would like to acknowledge the colleagues and researchers I met on my classes at Polytechnique and during my stays in the labs GREYC and MAP5.

If I was able to complete this project it is in great extent thanks to the people that surrounded me outside my studies. I whole-heartedly thank my parents and sisters for their support. I thank Greg for his friendship, and Daniel for his advice. My girlfriend Anna deserves all my gratitude. Thank you for all the support, advice, patience, and the endless proofreading.

I am convinced that the sport contributed to my work in many ways, this is why I would like to thank my coach Dorys and all my running companions.

# RÉSUMÉ

L'objectif des algorithmes de synthèse de textures par l'exemple est d'automatiser la génération de textures. L'exemple est une image 2D qui définit les caractéristiques visuelles à reproduire (couleurs, motifs, arrangements, etc.). Les méthodes dites "2D" génèrent de nouvelles images, typiquement plus grandes, semblables à l'exemple sans être des copies évidentes. Les méthodes de synthèse de textures "solides" quant à elles doivent inférer une structure tridimensionnelle des motifs présents dans l'exemple 2D. Il est attendu que les coupes du solide généré ressemblent à l'exemple 2D correspondant. Dans les deux cas, les textures synthétisées peuvent contenir des artefacts indésirés : manque de structures locales pour les motifs générés, régions floues ou uniformes, ou encore répétition de parties de texture identiques. Dans ce travail nous proposons deux méthodes pour la synthèse de textures 2D et solides. Ces méthodes ont pour objectif d'améliorer l'état de l'art concernant la qualité visuelle et la complexité algorithmique.

En comparaison des textures solides, les images sont légères à stocker et manipuler, donc l'aspect le plus important pour la synthèse de textures 2D est la qualité visuelle. Les méthodes par patchs sont typiquement basées sur une affectation au plus proche voisin et reproduisent généralement la structure des motifs dans l'exemple avec succès. Cependant, ces méthodes ne garantissent pas une synthèse exempte de régions floues ou de répétitions non désirées. Dans la première partie de cette thèse, nous proposons un algorithme de synthèse 2D par patchs basé sur l'affectation optimale. L'algorithme proposé reproduit la structure locale et garantit dans une certaine mesure l'absence de régions uniformes ou floues. Nos expériences montrent la capacité du modèle à générer des résultats de haute qualité visuelle pour différents types de textures. De plus, nous proposons des variantes pour réduire la répétition causée typiquement par la formulation par patchs. Nous obtenons des résultats satisfaisants en utilisant un échantillonnage aléatoire des patchs de l'exemple à la place de l'échantillonnage régulier. Finalement nous étudions deux mécanismes de relaxation pour contrôler la répétition des motifs et accélérer les calculs.

L'utilisation de textures solides générées hors ligne exigeant une quantité excessive de mémoire, l'objectif des méthodes de synthèse solide se centre sur le temps de calcul et la génération à la demande. C'est-à-dire, la capacité pour un algorithme de générer de façon indépendante les portions de texture nécessaires à un moment donné. Dans la deuxième partie de cette thèse, nous proposons un modèle de synthèse de textures solides basé sur les réseaux de neurones convolutifs. Ce modèle se compose d'un réseau générateur capable d'effectuer

la synthèse à la demande entraîné à l'aide d'une fonction d'évaluation par coupe. Cette dernière repose sur une *perte perceptuelle* définie par un réseau de neurones pré-entraîné. Elle permet d'entraîner le générateur en produisant seulement des coupes de textures solides, n'ayant qu'un seul voxel d'épaisseur selon une des dimensions spatiales. Les tranches générées utilisent peu de mémoire, ce qui permet de traiter des volumes de plus grandes résolutions que dans la littérature. Une fois entraîné, le générateur exécute la génération de textures à la demande avec un faible coût calculatoire. Dans le cas des motifs 3D isotropes, le modèle produit des résultats de haute qualité visuelle en comparaison des modèles de textures solides existants. Les vues en coupe des textures solides générées ont une qualité comparable avec les méthodes de l'état de l'art en synthèse de textures 2D. Finalement, nous étudions les limites de la modélisation 3D à partir d'exemples 2D.

# ABSTRACT

Example-based texture synthesis algorithms aim at automating the generation of textures. An *example* is a 2D image defining the visual characteristics that need to be reproduced in the new samples (colors, patterns, spatial arrangements, etc.). 2D texture synthesis methods generate new samples, usually larger, that *look* like the example without being a noticeable copy. Solid texture synthesis methods, on the other hand, need to infer a plausible three-dimensional structure. In this setting, it is usually required that some cross-sections of the generated solid texture *look* like a corresponding 2D example. In both cases, synthesized textures can suffer from noticeable artifacts that hamper the visual quality: lack of local structure in the generated patterns, blurry or constant regions, and repetitions of identical portions of textures. In this work we propose two new methods for 2D and solid texture synthesis. They aim at improving the current state of the art regarding computational complexity and visual quality.

Because 2D images are light and efficient to store and retrieve, the most important aspect of 2D texture synthesis is visual quality. Patch-based methods are usually based on a nearest-neighbor matching and achieve a good reproduction of the example's patterns structure. However, they do not guarantee a synthesis free of blurry regions or undesirable repetitions. In the first part of this thesis, we propose a 2D patch algorithm based on optimal assignment, that reproduces the local structures while guaranteeing to some degree the absence of blurry/constant regions. Experiments show the capability of the proposed model for reaching high visual quality for different kinds of textures. Additionally, we explore ways to reduce the repetition commonly caused by patch-based formulations. We show successful results using a random sampling of patches from the example instead of a regular one. We also investigate two relaxing mechanisms to control the repetition of patterns and to accelerate computation.

The colossal memory footprint involved when using solid textures generated off-line, shifts the focus of synthesis methods to computation time and on-demand evaluation. The latter refers to the capability of independently generating only the required portions of texture at a given moment. In the second part of this thesis, we propose a solid texture synthesis model based on Convolutional Neural Networks (CNN). It consists of a 3D generator network capable of on-demand evaluation, and a single-slice loss based on a *perceptual loss* defined by a pre-trained CNN. The proposed loss allows us to train the generator by only generating *slices* of solid texture, i.e., having a single voxel of thickness in one of the spatial dimensions. The 2D generated slices have a small memory footprint and allow us to use larger resolutions

than commonly encountered in the state of the art. Once trained, the generator network successfully performs on-demand evaluation with low computation times. The resulting visual quality regarding the generation of isotropic 3D patterns is advantageous compared with the state-of-the-art solid texture synthesis. The cross-sections of the produced solid textures achieve comparable results with state-of-the-art 2D texture synthesis methods. We also investigate the limits of this cross-section based formulation of solid texture synthesis from 2D examples.

# TABLE OF CONTENTS

ACKNOWLEDGEMENTS . . . . . iii

RÉSUMÉ . . . . . iv

ABSTRACT . . . . . vi

TABLE OF CONTENTS . . . . . viii

LIST OF FIGURES . . . . . xi

LIST OF SYMBOLS AND ACRONYMS . . . . . xxii

CHAPTER 1 INTRODUCTION . . . . . 1
1.1 Definitions . . . . . 1
1.1.1 Example-Based Texture Synthesis . . . . . 1
1.1.2 Example-Based Solid Texture Synthesis . . . . . 3
1.2 Challenges . . . . . 4
1.3 Goals and Hypotheses . . . . . 7
1.3.1 2D Texture Synthesis . . . . . 7
1.3.2 3D Texture Synthesis . . . . . 9
1.4 Organization of the Document . . . . . 10

CHAPTER 2 LITERATURE REVIEW . . . . . 11
2.1 2D Texture Synthesis . . . . . 11
2.1.1 Texture Synthesis Using Statistical Constraints . . . . . 12
2.1.2 Neighborhood-Based Texture Synthesis . . . . . 17
2.1.3 Texture Synthesis Using Deep Convolutional Neural Networks . . . . 21
2.2 Solid Texture Synthesis . . . . . 27
2.2.1 Procedural Solid Textures . . . . . 28
2.2.2 Example-Based Synthesis of Off-line Solid Textures . . . . . 30
2.3 Quality of Synthesized Textures . . . . . 35

CHAPTER 3 STATISTICALLY GUIDED PATCH-BASED 2D TEXTURE SYNTHESIS . . . . . 39
3.1 Texture Optimization . . . . . 39

3.1.1 Nearest Neighbor Model . . . 39
3.1.2 Optimization Scheme . . . 40
3.1.3 Robust Energy . . . 42
3.1.4 Multi-scale Scheme . . . 42
3.1.5 Quality of the Results . . . 43
3.2 Uniformity Mechanisms for Texture Optimization . . . 44
3.2.1 Histogram Matching . . . 45
3.2.2 Bidirectional Texture Energy . . . 46
3.2.3 Spatial Uniformity Distance . . . 48
3.3 Optimal Patch Assignment for Statistically Constrained Texture Synthesis . 49
3.3.1 Optimal Assignment Model . . . 49
3.3.2 Optimization Scheme . . . 50
3.3.3 Implementation Details . . . 55
3.3.4 Experimental Results and Comparison . . . 56
3.4 Relaxed Patch Assignment . . . 64
3.4.1 Relaxed Assignment Model . . . 64
3.4.2 Results . . . 65
3.5 Near-Optimal Patch Assignment . . . 66
3.5.1 Auction Algorithm . . . 67
3.5.2 Near-Optimal Patch Assignment Model . . . 70
3.5.3 Results . . . 70
3.6 Conclusion . . . 75

CHAPTER 4 ON-DEMAND SOLID TEXTURE SYNTHESIS USING DEEP CONVOLUTIONAL NETWORKS . . . 76
4.1 2D Texture Networks Overview . . . 76
4.1.1 Perceptual Loss . . . 77
4.1.2 Generator's Architecture and Training . . . 77
4.1.3 Analysis, Results, and Experiments . . . 80
4.2 Overview of the Proposed Method for Solid Texture Synthesis . . . 84
4.3 On-demand CNN Generator . . . 86
4.3.1 Architecture . . . 88
4.3.2 Spatial Dependency . . . 90
4.3.3 Multi-scale Shift Compensation . . . 90
4.4 Training . . . 91
4.4.1 3D Slice-Based Loss . . . 92

4.4.2 Single Slice Training . . . . . 93
4.5 Results . . . . . 94
4.5.1 Experimental Settings . . . . . 94
4.5.2 Experiments . . . . . 95
4.5.3 Comparison with the State of the Art . . . . . 107
4.6 Discussion . . . . . 110
4.7 Conclusion . . . . . 111

CHAPTER 5 CONCLUSION . . . . . 112
5.1 Summary of Work . . . . . 113
5.2 Limitations . . . . . 114
5.3 Future Research . . . . . 115

REFERENCES . . . . . 117

# LIST OF FIGURES

Figure 1.1 **Examples of textures with different degrees of stochasticity used through this work.** . . . . . 2

Figure 1.2 **Schematic representation of example-based 2D texture synthesis.** . . . . . 3

Figure 1.3 **Application of a 2D texture to a 3D model.** On the left, the textured model. On the right the unwrapped planar model and the association of texture to each piece. This configuration would produce an inconsistency in the wood pattern between the top and bottom of the table. Source: Gabbitt Media texturing and UV unwrapping tutorial video `http://gabbitt.co.uk`. . . . . 4

Figure 1.4 **Schematic representation of solid texture synthesis using 2D examples.** Usually, a block of solid texture is synthesized with patterns specified by one or more example 2D textures. This block can be then used to add texture to a 3D model. One alternative is to directly generate the texture required at the visible 3D coordinates on-line while rendering the model. . . . . 5

Figure 1.5 **Stochastic texture vs textures containing structured patterns.** 6

Figure 2.1 **Methods based on statistical constraints typically consist of two stages.** In the analysis stage the example is characterized with a set of statistical descriptors that then are used as parameters in the next stage. In the synthesis stage a new sample that matches these descriptors is generated, generally starting from a random initialization. 13

Figure 2.2 **Examples of texture synthesis using the statistical model of Portilla and Simoncelli [1].** Synthesized using the code provided by the authors with the following parameters: 4 scales, 4 orientations, neighborhood size of $7 \times 7$ pixels, and 50 iterations. The resolution in pixels is indicated at the bottom of each image. . . . . 16

Figure 2.3 **Examples of texture synthesis using the statistical model of Tartavel et al. [2].** Synthesized using the code provided by the authors with the default parameters. The resolution in pixels is indicated at the bottom of each image. . . . . 17

Figure 2.4 **Examples of texture synthesis using the methods of Wei and Levoy [3] and Ashikhmin [4].** Image resolution in pixels indicated at the bottom. Images from [4]. . . . . 19
Figure 2.5 **Examples of texture synthesis using the patch-based method of Efros and Freeman [5].** Images taken from the article [5]. The resolution in pixels is indicated at the bottom of each image. . . . . 20
Figure 2.6 **Examples of texture synthesis using the method of Kwatra et al. [6].** Images from [6]. The resolution in pixels is indicated at the bottom of each image. . . . . 22
Figure 2.7 **Examples of texture synthesis using VGG-19 as descriptor network with the method of Gatys et al. [7].** Images from the authors website `http://bethgelab.org/`. . . . . 24
Figure 2.8 **Comparison of texture synthesis results with the methods of Ulyanov et al. [8] and Gatys et al. [7].** Images from [8]. . . . . 25
Figure 2.9 **Examples of synthetic images generated with the GAN method of Karras et al. [9], trained on natural images**. Image from the article [9], each example has a resolution of $256 \times 256$ pixels. . . . . 27
Figure 2.10 **New textures synthesized with Periodic Spatial GANs [10]**. The training is performed using the *scaly* category of the DTD dataset [11]. 28
Figure 2.11 **Procedural texture from Perlin [12] applied to a 3D model.** . 29
Figure 2.12 **Procedural textures created using *substance designer* (software)**. The textures were designed by Nick Williams (`https://www.artstation.com/artwork/lOyRY`). . . . . 29
Figure 2.13 **Formulation of solid texture synthesis from 2D examples.** A set of 2D images $\{u_d, d \in \{1, \dots, D\}\}$ (in this case $D = 3$) defines the expected visual characteristics of all the cross-sections in direction $d$. Image inspired from [13]. . . . . 30
Figure 2.14 **Effects of the examples configuration.** Figure from [13]. For anisotropic textures it is important to carefully select the number of directions and the orientations of the example textures. (a) Example texture used along all axes. (b) Solid texture resulting from an inconsistent setting of the patterns. (c) Solid texture resulting from a setting with patterns horizontally aligned. (d) Solid texture resulting from a setting with patterns vertically aligned. All samples are synthesized using the method of Wei [14]. . . . . 31

Figure 2.15 **Slice-based strategy for solid texture synthesis.** The solid texture evolves iteratively from a random initialization. At each iteration: 1- The solid is decomposed on all its slices, 2- The slices are modified to better match the statistics of the example, 3- The slices are aggregated. 32

Figure 2.16 **Examples of solid texture synthesis using the method of Qin and Yang [15].** The top row shows two cases where the same example is used along three orthogonal directions. The bottom row shows one case where one different example is associated to each orthogonal direction. Images from [15]. . . . . 33

Figure 2.17 **3D neighborhood limited to neighbors in three orthogonal planes centered around one voxel.** Image from the solid texture synthesis survey of Pietroni et al. [16]. . . . . 34

Figure 2.18 **Examples of solid texture synthesis using the method of Kopf et al. [17].** In all the cases the same example is associated to the three orthogonal directions. The spheres are *carved* out of cubic arrays of textures. Images from [17]. . . . . 35

Figure 2.19 **Example of the correspondence map of a synthesized texture.** Large smooth regions indicate the verbatim copy of pieces from the example. Stochastic regions indicate innovation. . . . . 37

Figure 3.1 **Multi-scale synthesis scheme.** At each scale the top two images are the initialization and the example respectively and the bottom image is the synthesized sample. This sample is upsampled (↑) and used as the initialization at the following scale. . . . . 43

Figure 3.2 **Illustration of a nearest-neighbor (NN) affectation between points in 2D.** The two sets are differentiated by color and shape. The black lines indicate the affectation between pairs of points. Even at higher dimensions, using the nearest-neighbor model can result in an extremely uneven affectation between two sets of points. . . . . 44

Figure 3.3 **Examples of artifacts introduced by the method of Kwatra et al. [6].** We use yellow arrows to point at blurry/constant regions and orange arrows for the repeated identical pieces of texture. Images from the website of [6]. . . . . 45

Figure 3.4 **Empirical reweighting for the histogram matching used by Kopf et al. [17].** . . . . . 47

Figure 3.5 **Illustration of a bidirectional nearest neighbor (NN) affectation between two point clouds in 2D.** The two sets are differentiated by color and shape. The black and blue lines indicate the direct and inverse NN affectations between pairs of points. . . . . 48

Figure 3.6 **Illustration of an optimal assignment affectation between two point clouds in 2D.** The two sets are differentiated by color and shape. The black lines indicate the affectation between pairs of points. By definition the optimal assignment makes an even affectation that pairs all of the elements in both sets. In contrast to a simple permutation, the optimal assignment is the one that minimizes the overall distance. . . . . 55

Figure 3.7 **Value of Eq.** (3.13) **for three different textures during the optimization.** We take into account the values at different scales of the synthesis by normalizing by the number of patches. The spikes are due to the initialization at the scale transition. The vertical axis is in logarithmic scale. . . . . 57

Figure 3.8 **Examples synthesized with the proposed patch optimal assignment method.** The correspondence map highlights the piecewise identity nature of the results. . . . . 58

Figure 3.9 **RGB and patch histograms of the synthesized texture in the last row of Figure 3.8.** The absolute differences with the histograms of the example image are displayed in black. . . . . 59

Figure 3.10 **Illustration of the sub-grids of pixels $\Omega^{\downarrow}$ for an image resolution of $64 \times 64$ pixels.** The black dots represent the centers of the patches that are used. Left, regular spaced sampling of patches with stride of two which results in taking one patch out of four. Right, example of a random sampling of patches that takes one out of four patches. . . . . 60

Figure 3.11 **Innovation in a synthesis using a random pixel sub-grid in our patch assignment method.** The noisy correspondence map (CM) shows that the synthesis contains little local copy. All the images have a resolution of $256 \times 256$. . . . . 60

Figure 3.12 **Impact of the type of grids of pixels from the example in the patch assignment synthesis.** The hybrid synthesis refers to using random grids at the two coarse scales and a regular grid at the last, finest scale. The correspondence maps (CM) highlight the innovation/copy in the synthesis. All the images have a resolution of $256 \times 256$ and were synthesized using the same random *seed*. . . . . 61

Figure 3.13 **Comparison with other methods.** We compare our approach with the patch-based method of Kwatra et al. [6], the two statistical matching methods of Portilla et al. [1] and Tartavel et al. [2], and the PatchMatch method of Barnes et al. [18] (with the multi-scale implementation of David Tschumperlé for texture synthesis [19]). These approaches are used with the default parameters described in the respective papers. . . . . 63

Figure 3.14 **Comparison of results for different values of r.** From left to right: $r = 2$, $r = 0.8$, $r = 0.8$ with Gaussian falloff weighting, and $r = 1$ using the same random initialization. Results are quite similar, except for $r = 2$ which is noticeably less sharp. . . . . 64

Figure 3.15 **Illustration of the affectation between two point clouds using the relaxed assignment model.** The two sets are differentiated by color and shape. The black lines indicate the affectation between pairs of points. The value of parameter $\rho$ controls the shift from a Nearest Neighbor affectation and an Optimal Assignment. . . . . 66

Figure 3.16 **Texture synthesis with relaxed assignments for different values of the relaxation parameter $\rho$.** The value $\rho = 0$ corresponds to nearest neighbor matching, which does not respect patch distribution. Increasing $\rho$ gives relaxed assignments that allow for slight variations of the synthesized sample's patch distribution. For large values of $\rho$, the assignment is an optimal permutation resulting in the reproduction of the characteristics of the input. . . . . 67

Figure 3.17 **Illustration of the affectation between two point clouds using the near-optimal assignment model.** The two sets are differentiated by color and shape. The black lines indicate the affectation between pairs of points. The value of the parameter $\epsilon$ controls the optimality of the affectation. In all cases the result is a permutation, but for small values of $\epsilon$ the energy is smaller. . . . . 71

Figure 3.18 **Texture synthesis using the near-optimal patch assignment model**. The top row shows the example and the result of using the optimal assignment model of Section 3.3 ($\epsilon = 0$). The remaining rows show the results using the near-optimal model with varying values of parameter $\epsilon$. All the images have a resolution of $192^2$ pixels and are normalized to values between $[0, 1]$. All the results were synthesized using the same random *seed*. . . . . 73

Figure 3.19 **Optimization curves for the near-optimal patch assignment model.** The curves correspond to the results in Figure 3.18. . . . . 74

Figure 3.20 **Computation times for different values of** $\epsilon$. We use the near-optimal patch assignment model on several examples from the VisTex database. The resolution of the example textures and synthesized samples were $192^2$ pixels for this experiment. . . . . 74

Figure 4.1 **Training framework of Ulyanov et al. [8]**. The CNN Generator network $\mathcal{G}(\cdot|\theta)$ processes a multi-scale noise input $Z$ to produce a new texture sample $v$. The loss $\mathcal{L}$ compares the feature statistics induced by the example $u$ in the layers of the pre-trained Descriptor network $\mathcal{D}(\cdot)$ with those induced by the generated sample $v$. The training iteratively updates the parameters $\theta$ to reduce the loss. . . . . 78

Figure 4.2 **Ulyanov et al. [8] generator's architecture**. It processes a set of noise inputs $Z = \{z_0, \ldots, z_K\}$ at $K+1$ different scales using convolution operations and non-linear activations. The information at different scales is combined using upsampling and channel concatenation. $M_i$ indicates the number of input channels and $M_s$ controls the number of channels at intermediate layers. For simplicity we consider square outputs with spatial size of $N_1 = N_2 = N$. For each intermediate feature the spatial size is indicated above and the number of channels below. . . . . 79

Figure 4.3 **Evolution of the generator in our implementation of Ulyanov et al. [8] method**. Samples generated at different stages of training indicated by the iteration number. We used the same noise input for all the samples. . . . . 81

Figure 4.4 **Results using our implementation of Ulyanov et al. [8].** Left, example ($256^2$ pixels), right, synthesized sample of twice the side size. We ran the training for 5000 iterations with 4 samples per mini-batch and a learning rate starting at 0.1 and decreasing by 0.8 every 1000 iterations. . . . . 83

Figure 4.5 **Value of the empirical loss during the training of the generator for the textures dotted, leafy, and jagnow of Figure 4.4.** . . . 84

Figure 4.6 **Impact of size of training**. We trained one generator for each example at four resolutions (indicated in the left). During training, the generator produces samples of the same size as the example. . . . . 85

Figure 4.7 **Impact of size of training** We trained one generator for each example at four resolutions (indicated to the left). During training, the generator produces samples of the same size as the example. . . . . 86

Figure 4.8 **Results using Max pooling vs Average pooling on the layers of VGG during training.** . . . . . 87

Figure 4.9 **Result of training with and without gradient normalization (using average pooling on VGG)**. . . . . 87

Figure 4.10 **Training framework for the proposed CNN Generator network $\mathcal{G}(\cdot|\theta)$ with parameters $\theta$.** The generator processes a multi-scale noise input $Z$ to produce a solid texture $v$. The loss $\mathcal{L}$ compares, for each direction $d$, the feature statistics induced by the example $u_d$ in the layers of the pre-trained Descriptor network $\mathcal{D}(\cdot)$ to those induced by a each slice of the set $\{v_{d,1}, \ldots, v_{d,N_d}\}$. The Training iteratively updates the parameters $\theta$ to reduce the loss. We show in Section 4.4 that we can perform training by only generating single-slice solids instead of full cubes. . . . . 88

Figure 4.11 **Schematic of the Generator's architecture**. It processes a set of noise inputs $Z = \{z_0, \ldots, z_K\}$ at $K+1$ different scales using convolution operations and non-linear activations. The information at different scales is combined using upsampling and channel concatenation. $M_i$ indicates the number of input channels and $M_s$ controls the number of channels at intermediate layers. For simplicity we consider a cube shaped output with spatial size of $N_1 = N_2 = N_3 = N$. For each intermediate cube the spatial size is indicated above ($\lceil\cdot\rceil$ stands for the ceiling function) and the number of channels below. . . . . 89

Figure 4.12 **Value of the 3D empirical loss, Eq.** (4.4)**, during the training of the generator for the textures *histology*, *cheese*, and *granite* of Figure 4.15.** . . . . . 96

Figure 4.13 **On-demand tiling consistency**. Left: example texture of $128^2$ pixels and 512 cubes of generated texture that tile perfectly, we add a gap between the cubes to show the continuity of the patterns. Right: example texture of $512^2$ pixels with an assembled set of blocks of different sizes generated on-demand. Note that it is possible to generate blocks of size 1 in any direction. . . . . 96

Figure 4.14 **Texture mapping on 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.* . . . . 97

Figure 4.15 **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° angle. The trained models successfully reproduce the visual features of the example in 3D. . . . 98

Figure 4.16 **Synthesis of isotropic textures (same presentation as in Figure 4.15)**. 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 local high frequency artifacts. . . . . 99

Figure 4.17 **2D to 3D isotropic patterns.** The example texture (top-left) depicts a pattern that is approximately isotropic in 2D, but the material it depicts is not. Training the generator using the example along three orthogonal directions results in solid textures that are isotropic in the three dimensions (top-right). Here, the red and green patterns vary isotropically in the third dimension, this creates a bigger variation on their size and makes some slices contain red patterns that lack the green spot. This is a case where the slices of the solid texture (bottom) cannot match exactly the patterns of the 2D example, thus, not complying with the example in the way a 2D algorithm would. . . . . 101

Figure 4.18 **Illustration of a solid texture whose cross sections cannot comply with the example along three directions**. Given a 2D example formed by discs of a fixed diameter (upper left) a direct isotropic 3D extrapolation would be a cube formed by spheres of the same diameter. Slicing that cube would result in images with discs of different diameters. The cube in the upper right is generated after training our network with the 2D example along the three orthogonal axes. The bottom row shows cross-sections along the axes, all of them present discs of varying diameters thus failing to look like the example. . . . 102

Figure 4.19 **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. 103

Figure 4.20 **Importance of the compatibility of examples**. In this experiment, two generators are trained with the same image along three directions, but for two different configurations. 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. Finally, the rightmost column gives the empirical loss value at the last iteration. 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. The lower value of the training loss for this configuration reflects the better reproduction of the patterns in the example. . . . . 104

Figure 4.21 **Diversity among generated samples**. We compare the middle slices along three axes of three generated textures of $256^3$ voxels. The comparison consists in finding the pixel with the most similar neighborhood (size $4^2$) in the other image and constructing a correspondence map given its coordinates. The smooth result in the diagonal occurs when comparing a slice to itself. The stochasticity in the rest of results means that the compared slices do not share a similar arrangement of patterns and colors. . . . . 106

Figure 4.22 **Anisotropic texture synthesis using two examples**. The first three columns show the examples and the training configuration (i.e., how images are oriented for each view) used that lead to the result in the fourth column ($v$). In the last three columns we experiment with preprocessing one of the examples to match the colors of the other (by performing a histogram matching independently on each color channel). The example in the HM column is the one preprocessed and the last column again shows the resulting solid texture. We observe favorable results particularly when the colors of both examples are close. 107

Figure 4.23 **Comparison with the existing methods that produce the best visual quality**. The last row shows the results using the proposed method. Our method is better at reproducing the statistics of the example compared to Kopf et al.'s [17] method and is better at capturing high frequencies compared to both methods. . . . . 108

Figure 4.24 **Comparison of our approach with Kopf et al. [17]**. 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 [17], 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 $640^3$ and $512^3$ for the rest. . . . . 109

# LIST OF SYMBOLS AND ACRONYMS

| | |
|---|---|
| CNN | Convolutional Neural Networks |
| GAN | Generative Adversarial Networks |
| GPU | Graphics Processing Unit |
| IS | Inception Score |
| FID | Fréchet Inception Distance |
| NN | Nearest Neighbor |
| IRLS | Iterative Reweighted Least Squares |
| LSAP | Linear Sum Assignment Problem |
| CM | Correspondence Map |
| ReLU | Rectified Linear Unit |
| PRNG | Pseudo-Random Number Generator |
| LRP | Local Random-Phase |

# CHAPTER 1 INTRODUCTION

3D computer graphics applications play a substantial role in an ever-expanding digital world. Digital artists conceive and bring to life impressive 3D models, which are at the core of applications like animated movies, video games, visual effects, or augmented reality. Textures go hand in hand with this process as they define the color at every point in the space. Their visual complexity makes possible to enrich the 3D digital objects' appearance without further elaborating the geometry. Textures are generally encountered in the form of 2D raster images and they can be the product of an artist—scanning a drawing or created in digital form—or a digital photograph. The geometry of a 3D model is generally represented as a collection of points in the 3D space, associated to form polygons.

In this context, it is important to have a *large enough* texture to cover a given model. On one hand, tiling a low resolution texture can cause unpleasant obvious repetitions and visible seams. On the other hand, scaling the texture can cause unrealistic size proportions between the patterns in the texture and the geometry of the model. Additionally, it might be difficult to manually create a larger version of this texture. Even at the right resolution, texture mapping, i.e., the process of associating each point of a surface in the 3D space to a 2D image, often introduces distortions that aggravate with the complexity of the 3D model.

In this work we address those problems by conceiving a new example-based 2D texture synthesis method to generate textures of an arbitrary size and a new example-based solid texture synthesis method that trivializes texture mapping.

Aside from 3D graphics, texture synthesis models are useful in image editing applications as filling or replacing content (i.e., inpainting) or transferring the visual style, such as color palette, strokes style, etc., from one image to another without altering the remaining information (i.e., style transfer).

## 1.1 Definitions

### 1.1.1 Example-Based Texture Synthesis

In computer graphics any image can be used as a texture. In this work, however, we focus on a narrower definition of textures commonly used in the field of image processing. We define textures as raster images consisting of repeating patterns with a certain degree of randomness. That randomness influences the variability among the patterns and their overall spatial arrangement. Textures are often classified by this level of randomness that goes from

deterministic to stochastic [13,20]. Figure 1.1 shows some examples of textures used through this work. The patterns in the top-leftmost texture follow the rather deterministic grid arrangement but contain a proportion of stochasticity that causes their variability. Contrarily, in the top-rightmost texture the patterns that form the ground have variable shapes and random arrangements, which makes the color of each pixel rather stochastic.



Figure 1.1 **Examples of textures with different degrees of stochasticity used through this work.**

In this context, it is generally assumed that textures are stationary, meaning that their statistical properties do not change spatially, or in other words, that any regions of the image of the same size have the same visual characteristics. Textures can be derived from digital photography, scanned drawings, or directly generated in a graphics editor software. In any case, textures require human intervention either for the creation or for the manipulation of their scale and resolution. Naturally, for most textures, repeating and tiling verbatim copies of the image is not visually satisfactory.

Example-based texture synthesis is a powerful tool for efficient texture generation. Given an *example* texture with the desired visual characteristics, example-based texture synthesis methods aim at generating new visually similar samples of arbitrary sizes.

It is also required that the synthesis process introduces enough variation so that every synthesized sample looks different from the others. Figure 1.2 illustrates what an example-based

texture synthesis algorithm is expected to accomplish: the synthesized sample $u$ reproduces the visual characteristics of the example $u_0$ without containing any copies of it. Instead, $u$ captures the variability and arrangement of the patterns in $u_0$ so it is perceived as another sample of the same texture. Example-based texture synthesis methods are usually not tailored to a single example texture or one type of texture in particular. Instead, they are expected to perform well with various kinds of textures, as illustrated in Figure 1.1.

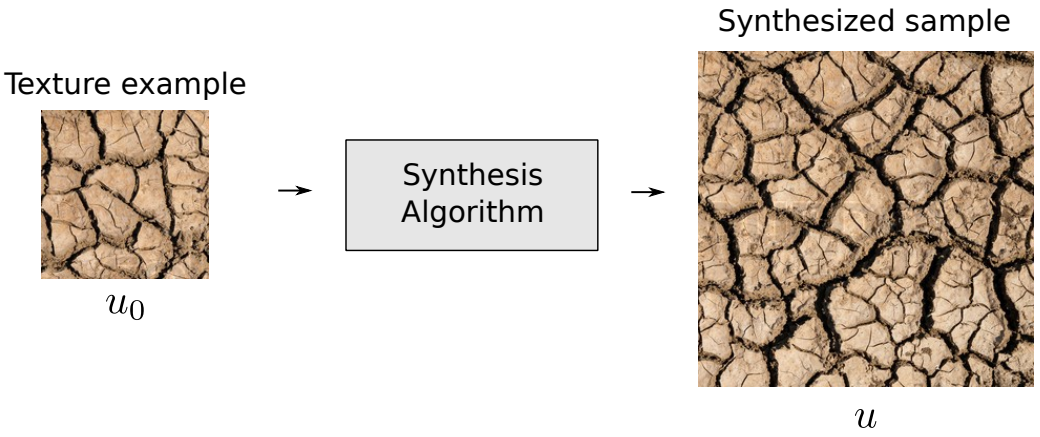



Figure 1.2 **Schematic representation of example-based 2D texture synthesis.**

### 1.1.2 Example-Based Solid Texture Synthesis

An important step for applying a 2D texture to a 3D model is defining the *texture mapping* that associates the 3D coordinates of the model to the pixel grid of the 2D texture. There are several ways to automatically find a texture mapping while trying to minimize distortions [21]. Yet, in practice it is widely manually defined by artists with the help of graphics software. They decompose (unwrap) the 3D geometry into *flattened* pieces and place them on top of the 2D texture. Such positioning defines the mapping, thus, it has to be done carefully in a way that the texture is consistent along the surface and that minimizes visible seams or distortions. Figure 1.3 shows the unwrapping for a simple 3D model of a table. In this example, with a table having mostly flat surfaces, the attention goes to creating a consistent wood alignment. More complex geometries can be very difficult because not only the different pieces need to be consistent but the curved pieces can distort the applied texture. Texture mapping is a complex challenge. Either manual or automatic, it always involves the risk of introducing artifacts and distortions depending on the complexity of the surface. Example-based solid texture synthesis is an elegant alternative to texture mapping. It addresses the generation of solid textures, i.e., volumetric color data with the visual properties of textures.

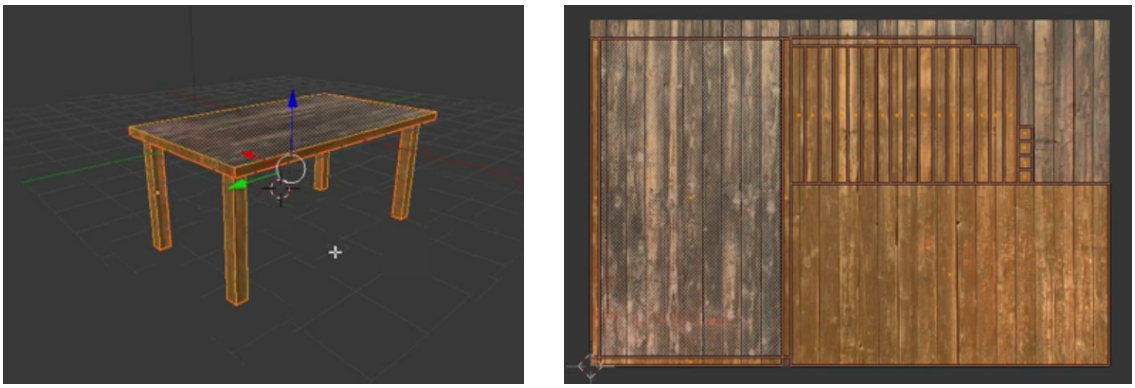


Figure 1.3 **Application of a 2D texture to a 3D model.** On the left, the textured model. On the right the unwrapped planar model and the association of texture to each piece. This configuration would produce an inconsistency in the wood pattern between the top and bottom of the table. Source: Gabbitt Media texturing and UV unwrapping tutorial video `http://gabbitt.co.uk`.

Solid textures can be used on surfaces without further parametrization disregarding the intricacy of the geometry. Oppositely to 2D texture synthesis, the availability of solid texture examples is limited. To cope with this limitation, example-based solid texture synthesis methods commonly focus on generating 3D samples whose cross-sections along a given axis are visually similar to a respective 2D example texture. Once the volumetric information is generated, it is possible to directly evaluate the color at any point given its 3D coordinates. Figure 1.4 illustrates a common setting for solid texture synthesis. In this case, a single example defines the expected visual characteristics of the cross-sections of the solid texture along three orthogonal axes. Usually, a whole volume of voxels is generated, and loaded in memory to add a texture to a 3D mesh model. Solid texture synthesis usually deals with higher computation times and memory footprint, thus making efficiency a crucial requirement for these methods.

## 1.2 Challenges

The main goal of an example-based texture synthesis method is to generate new samples that reproduce the visual characteristics of an example texture as perceived by human observers. The quality of the samples can be easily assessed qualitatively by human visual examination while automatic evaluation is still a difficult challenge by itself. A successful texture synthesis algorithm has to faithful reproduce local patterns, ensure statistical consistency, and be able to generate diverse samples. Additionally, computation time is a crucial aspect for practical applications. Finally, in the case of solid texture synthesis, given the absence of solid texture

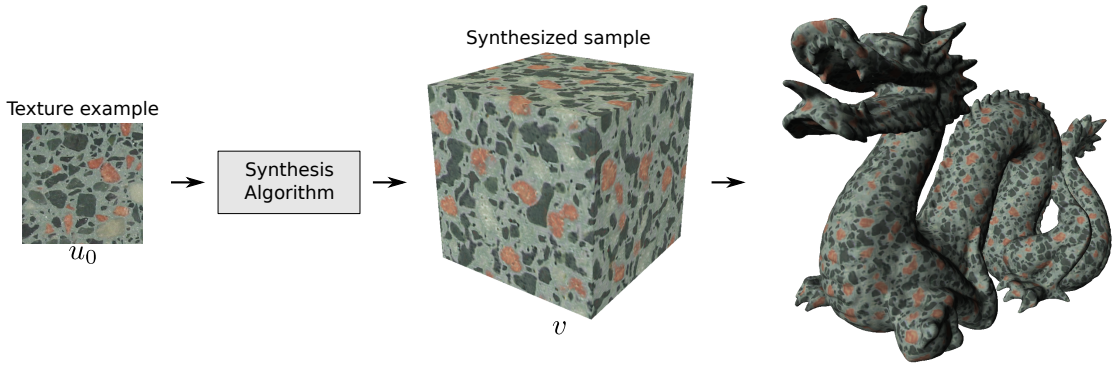



Figure 1.4 **Schematic representation of solid texture synthesis using 2D examples.** Usually, a block of solid texture is synthesized with patterns specified by one or more example 2D textures. This block can be then used to add texture to a 3D model. One alternative is to directly generate the texture required at the visible 3D coordinates on-line while rendering the model.

examples to drive the synthesis, it is uncertain that for a given 2D example, there exists a 3D solution. We discuss these concepts in the following.

### Statistical Consistency

As stated by Julesz et al. [22,23], matching some high order statistics of the occurrence of the pixels should be enough to reproduce the visual characteristics of the image. In practice this might not be sufficient for many photo-realistic textures, e.g., those composed of discernible patterns [24]. Still, there are several aspects for a successful texture synthesis that can be interpreted as being consistent with the example's statistics. First of all, we have the *color's statistics*: identical colors must be present in the example and in the synthesized samples in the same proportion. Second, we have the *variability* of the structural patterns: they must vary in a similar way and amount. Ideally, the model should be able to innovate and create new content, no piece in a synthesized sample should be identical to a piece of the example. Finally, a common obstacle for example-based texture synthesis algorithms is to be able to maintain *long distance relationships* in the example. A typical example is a texture depicting a brickwall, where the bricks are aligned along the image. Misalignments in the synthesized sample become eye-catching artifacts.

### Local Structure

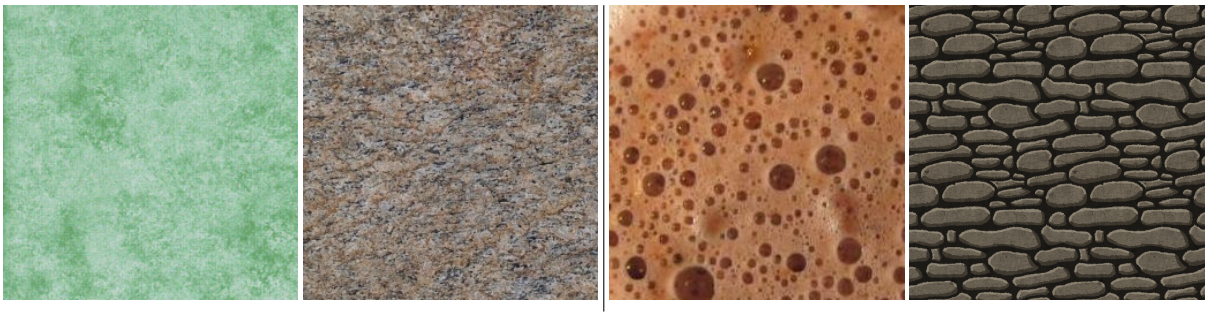
Many textures are composed of complex shapes and structures that are noticeable mainly thanks to the high frequencies at their edges. We refer to those patterns as local structure and

it has been shown to be the most important aspect for assessing the quality of synthesized textures [24]. Figure 1.5 contrasts textures with structured patterns to rather stochastic textures. At the given scale, the pixels in the *structured* textures are organized to form complex patterns e.g., bubble and rock shapes, while there is no visible organization in the stochastic textures.

A successful texture synthesis algorithm needs to reproduce the local structure so that the patterns in the output are similar to the ones in the example. We highlight this characteristic because in practice, the more discernible the structure patterns are in a texture, the harder they are to reproduce by matching the texture's statistics only. Thus the texture synthesis model needs to take into account the local structure.

### Diversity

In the context of texture synthesis, we define diversity as the capability of the method to synthesize different samples while depicting the same visual characteristics. A pair of synthesized samples should be easily perceived as two different images—or two different instances of the same texture. This is of particular importance for the kind of textures where the local structure elements are easily discernible because they make duplication more striking. Most texture synthesis methods instill diversity by including a random input or initialization. In that way, diversity is related to the degree on which the model reacts to this input. If ignored, the algorithm can be stuck in synthesizing the same image or imperceptible variants of it.



Stochastic Structured patterns

Figure 1.5 **Stochastic texture vs textures containing structured patterns.**

#### Existence

Existence arises in the context of solid texture synthesis from 2D examples (as well as in the synthesis of non-stationary textures, not addressed in this work). In the absence of solid texture examples, a solid texture is usually modeled by imposing some example's visual characteristic on its planar cross-sections. It is, however, uncertain that for a given example (or set of examples), a volumetric version of the patterns exists. As it will be detailed in Section 4.5 this uncertainty is caused by the classical way of modeling the example-based solid texture synthesis problem. If no *satisfactory* solution exists, depending on the synthesis model, this can prevent the algorithm to find a visually appealing result.

#### Computation time and on-demand evaluation

While it is common to store 2D textures in memory to be used in interactive applications, the memory footprint of solid textures makes it impossible to simply use them as their 2D counterpart. In practice, replacing 2D texture mapping with solid textures requires efficient solid texture synthesis methods capable of on-demand evaluation. That is, an algorithm capable of computing a color for a given spatial coordinate without synthesizing the full volume of texture. This reduces computation and makes it possible to distribute the synthesis process, and to load in memory only the portions of texture required for the visible parts of the 3D object.

## 1.3 Goals and Hypotheses

The main goal of the proposed work is to conceive and implement example-based 2D texture and solid texture synthesis methods that improve the current state of the art, regarding the aforementioned aspects, such as computational complexity and visual quality.

### 1.3.1 2D Texture Synthesis

#### General goal

Our general goal is to conceive and implement an example-based 2D texture synthesis algorithm that equals the local visual quality of the non-parametric patch-based frameworks and the stability of statistical matching methods.

**Specific goals**

The synthesized samples must:

1. Globally, match some high-order statistics of the example texture in order to produce the same visual impression. The statistics used must help to capture the color distribution and the overall variability of the features present in the example;
2. At a local level, match the features of the example. That is, the pixels must be arranged in patterns similar to those in the example texture;
3. Avoid verbatim copy. Any two synthesized samples must be pixel-wise different to each other and compared to the example.

Additionally, the model must be capable of:

4. Handling diverse types of textures;
5. Controlling the degree of compliance with the example statistics as a way to obtain different visual impressions if desired.

**Hypotheses**

1. The optimal assignment between the distributions of patches of two images can be used as a variational model for texture synthesis. Similar to Kwatra et al. [6], the use of patches of the example to synthesize a new image should help reproduce its local structures. However, the optimal assignment should provide a statistical controlling scheme that guarantees the compliance with the example thus enhancing the visual quality of the results.
2. In a patch-based framework, a random initialization promotes enough diversity for independent samples to be different.
3. The degree of compliance with the patch distribution can be controlled using a relaxation in the assignment model's constraints.
4. Optimal transport is a better metric for patch distribution than nearest-neighbor.

### 1.3.2 3D Texture Synthesis

#### General goal

Our general goal is to conceive and implement a solid texture synthesis model that delivers better than the state-of-the-art visual quality and computation speeds, while being capable of on-demand synthesis, anticipating the integration in a graphics pipeline.

#### Specific goals

The model must:

1. Generate state-of-the-art visual quality results. This involves capturing both the global and local characteristics of the example;
2. Improve the computation time of solid synthesis compared to existing methods, i.e., around one minute for a $128^3$ voxels solid;
3. Manage larger resolution images than the current methods. That is, to use larger images than $256^2$ pixels;
4. Be capable of on-demand synthesis, which saves memory utilization, and therefore promotes the use in practical applications.

#### Hypotheses

1. Convolutional Neural Networks (CNN) can be trained for the task of generating solid textures that present similar visual impression than a given 2D example.
2. The perceptual loss [7, 25], successful in 2D state-of-the-art methods, can be used in a slice-based volumetric setting to achieve state-of-the-art visual quality results for solid textures.
3. The CNN can have a compact multi-scale architecture without hampering visual quality. This should speed up computation and help to manage large images during training.
4. The local dependency between the input and output of a CNN network without padding makes it possible to perform on-demand evaluation.

## 1.4 Organization of the Document

In Chapter 2, we give an overview of the state of the art for example-based 2D and solid texture synthesis. In Chapter 3, we propose a new texture synthesis variational model based on a global assignment of patches. In Chapter 4, we propose a new approach to generate solid textures on-demand using a 3D convolutional neural network. Finally in Chapter 5, we summarize our contributions and state our conclusions.

# CHAPTER 2 LITERATURE REVIEW

Example-based 2D texture synthesis is a popular area of study. There is a vast variety of methods that attain different levels of visual quality and efficiency. Because images generally occupy a small amount in memory, they can be stored and used efficiently. Currently, the main focus of this area of study is to improve visual quality. In Section 2.1 we briefly describe the state of the art for example-based 2D texture synthesis. This provides an overview of the recent advances and challenges in the area before presenting our proposed model in Chapter 3. We propose a new example-based 2D texture synthesis method that delivers high visual quality results by matching the patch statistics of the example.

Solid texture synthesis methods are scarce in comparison. They address the problem of texturing 3D models in computer graphics where they compete with geometry-based approaches like surface texturing or automatic texture mapping [26]. Solid textures offer a significant advantage, they can be used to texture any surface without requiring parametrization, thus avoiding distortions regardless of the surface's geometry. As for 2D, many example-based solid texture synthesis methods generate samples off-line for further use. These models achieve good visual quality but are impractical because of the large memory requirements of solid textures. Therefore, on-demand evaluation is critical. In Chapter 4 we propose an example-based solid texture method that uses convolutional neural networks and is capable of on-demand evaluation. Section 2.2 is devoted to the existing methods for solid texture synthesis.

2D and solid texture methods progress together. Most solid texture methods are designed as an extrapolation of 2D methods. However, some quality improvements proposed directly for solid texture methods can be used to improve the quality of 2D methods.

Finally, in Section 2.3 we discuss several methods that attempt to measure the quality of the synthesized textures and other strategies that assist human visual evaluation.

## 2.1 2D Texture Synthesis

We can sort example-based texture synthesis methods into three categories:

- Statistical constraining methods describe a given texture using a set of statistical measurements from the example. They produce a new sample by shaping a random image to match these measurements.

- Neighborhood-based methods consider that a texture is totally defined by the local arrangement of patterns in the example. They produce new samples by copying and stitching together coherent pieces from the example as in a jigsaw puzzle.

- Methods based on deep neural networks commonly use a *descriptor* network to characterize a texture. They produce new samples either by shaping a random image or by training a *generator* network given an error signal from the descriptor. In the latter setting, both networks can be trained separately or simultaneously.

### 2.1.1 Texture Synthesis Using Statistical Constraints

Texture synthesis using statistical constraints was originally inspired by human perception theory [22, 23]. These methods follow the hypothesis that there exists a set of statistical measurements such that, two images having the same set of values will produce the same visual impression on human observers.

This hypothesis is easy to verify for simple synthetic images with constant or random values for all of their pixels. In natural images, however, the relationships between pixels can be highly complex. This complexity makes natural images difficult to be described using a small set of statistical measurements. It seems, however, reasonable to assume that textures, given their homogeneous spatial characteristics, can be described by such a set of statistical measurements. Another common assumption of these methods is that the example image contains enough information for the statistical measurements to be representative of the given texture.

Methods in this category usually consist of a proposed set of statistical measurements and a mechanism to generate new samples constrained by the values produced by the example. Texture synthesis is usually performed in two stages, as depicted in Figure 2.1: an analysis stage where the example texture is characterized by the statistical measurements and a synthesis stage where a new image that exhibits the same statistical characteristics is generated. This is often done by shaping an initial noise image which encourages the random arrangement of patterns and diversity among synthesized samples.

The method proposed by Portilla and Simoncelli [1] is an influential example on how to define a set of meaningful descriptors (statistical measurements) for modeling textures. They propose the following intuitive procedure which ultimately relies on the qualitative assessment made by human observers. It consists of the following steps:

1. Start with a set of basic descriptors and use it to synthesize a large set of textures.

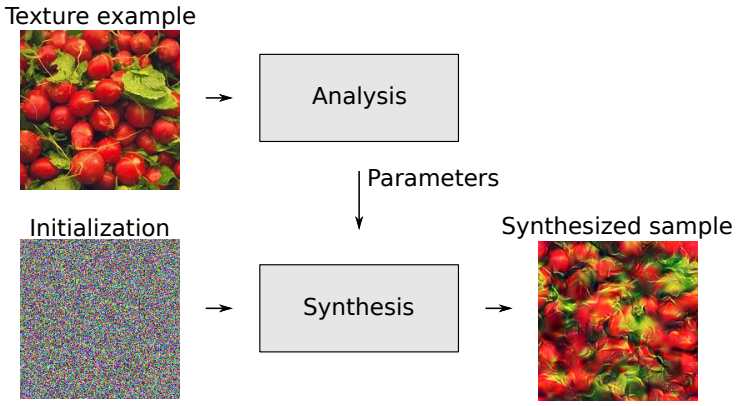



Figure 2.1 **Methods based on statistical constraints typically consist of two stages.** In the analysis stage the example is characterized with a set of statistical descriptors that then are used as parameters in the next stage. In the synthesis stage a new sample that matches these descriptors is generated, generally starting from a random initialization.

Ideally, this set should represent all the kinds of textures on which the model is expected to perform well.

2. Classify poor quality results given the type of failure, then identify the group that produces the poorest results.

3. Chose a new statistical descriptor that captures the missing relationships between pixels in the poor quality group.

4. Verify that the new descriptor corrects the intended problem by re-synthesizing the textures where the model previously failed.

5. Verify that the original descriptors are still necessary. For each descriptor, find a texture that fails if it is not used for the synthesis.

This process is followed, at least indirectly, every time one designs a new texture model. The initial set of descriptors are often the ones used in an existing method and the first set of failure cases is those of that given method. Among existing methods, the set of descriptors often involves the example image's response to several filters whose distribution is summarized using either moments (mean value, correlation matrices, ...) or empirical distributions. A broad range of such descriptors have been investigated such as color histograms, Fourier modulus [27], or steerable filter response histograms [1,28–30]. Recently, Tartavel et al. [2,31] use occurrences in an adapted patch dictionary.

The mechanism used to generate new samples depends on the set of descriptors. The synthesis can usually be formulated as an optimization problem (although this is not always explicit) where, for some descriptors, the goal is to minimize the difference between the value produced by the new image and the value produced by the example, while other descriptor values constrain the optimization, i.e., the new image has to match exactly the value produced by the example. That problem is generally non-convex and often non-smooth, so most methods rely on strategies to obtain a local optimum. Those strategies include: histogram matching [29], alternating projections [32], averaged projections [2]. An alternative approach is to use maximal entropy models [33,34], which was recently extended for CNN features [35].

In the following paragraphs we detail the texture model based on steerable filter statistics [1] and the texture model based on occurrences in a patch dictionary [2].

### Texture Model Based on Steerable Filter Statistics

The simplest measurements we can use to describe an image are the ones computed in the pixel domain. Some examples are: histogram, mean, variance, or range. They summarize some meaningful information but do not capture spatial relationships between pixels, which is necessary to accurately model textures consisting of discernible patterns. Instead, images are often analyzed using a different representation such as the response coefficients to a set of filters.

The *Steerable Pyramid Decomposition* [32] is a powerful representation for textures. It is a multi-scale and multi-orientation decomposition of an image that facilitates the extraction of the patterns formed by the pixels at different spatial ranges. See [36] for a detailed description of the steerable pyramid decomposition.

Heeger and Bergen [29] describe an example texture using marginal histograms of the coefficients in the pixel and in the steerable pyramid domains. The mechanism they use for texture synthesis is *histogram matching*, which consists of modifying the coefficients in the image being synthesized to match the histogram of the example. They perform texture synthesis iteratively, starting from a random image initialization. Each iteration consists of decomposing the image, followed by matching the histograms at the different levels of the pyramid and the pixel domain, and finally reconstructing the image.

The model of Portilla and Simoncelli [1] uses a complex steerable pyramid. It describes textures using joint statistics at each scale and across scales along with some statistics from the pixel domain. Those descriptors are used as constraints to iteratively shape a random input, drawn independent and identically distributed (iid) from a Gaussian distribution, in

order for it to exhibit the same statistics. The iterative algorithm consists of sequential projections onto the sets of images that match each constraint. The convergence is not theoretically guaranteed but experiments show that the process is stable. The empirical convergence's speed depends on the number of scales and orientations and the size of the neighborhood, which are parameters set by the user.

Figure 2.2 shows synthesis examples synthesized using the method of Portilla and Simoncelli [1] with the code provided by the authors. For all the cases, the synthesized texture matches the statistical descriptors of the example texture which results in a global similarity and a good reproduction of the colors. The local structure of the patterns is however only partially reproduced and it aggravates with more complex structures. For instance, the top-left texture contains rather stochastic patterns which are reproduced satisfactorily; in contrast, the radishes and leaves in the top-right example are poorly reproduced. This is a classical result of statistical matching methods. They usually perform well on stochastic textures but often fail to faithfully reproduce the local elements found in textures with discernible patterns. This shows the difficulty to capture the local relationship between pixels using a statistical measurement.

### Texture Optimization Model Based on Occurrences on a Patch Dictionary

Despite the complexity of the filters in the common representations used for images, complex local structure is often poorly captured. Inspired by the initial neighborhood-based methods (detailed in the next sub-section), Peyré [37] and Tartavel et al. [2, 31] integrate local information of the image in a texture model using a patch (i.e., small spatial subsets of pixels) dictionary for sparse representations.

They use a redundant set of patches to create an adapted dictionary, which is a collection of simpler patches or *atoms* that approximate the original set via sparse combinations. Thus, forming an efficient representation of the local content of the image. Texture synthesis can be performed by finding the coefficients on the patch dictionary that produce patches similar to those in the example and then constructing an image given those generated patches.

The use of a dictionary for the patch's representation helps to maintain coherence with the example's local structure, without performing exact copies and repetition. Thus helping the model generalize by allowing a small reconstruction error for patches.

In the work of Tartavel et al. [2, 31] texture synthesis is formulated as an optimization problem where the target descriptor parameters are divided into three sets of measurements. The pixel's histogram captures the color distribution of the example texture. The Fourier

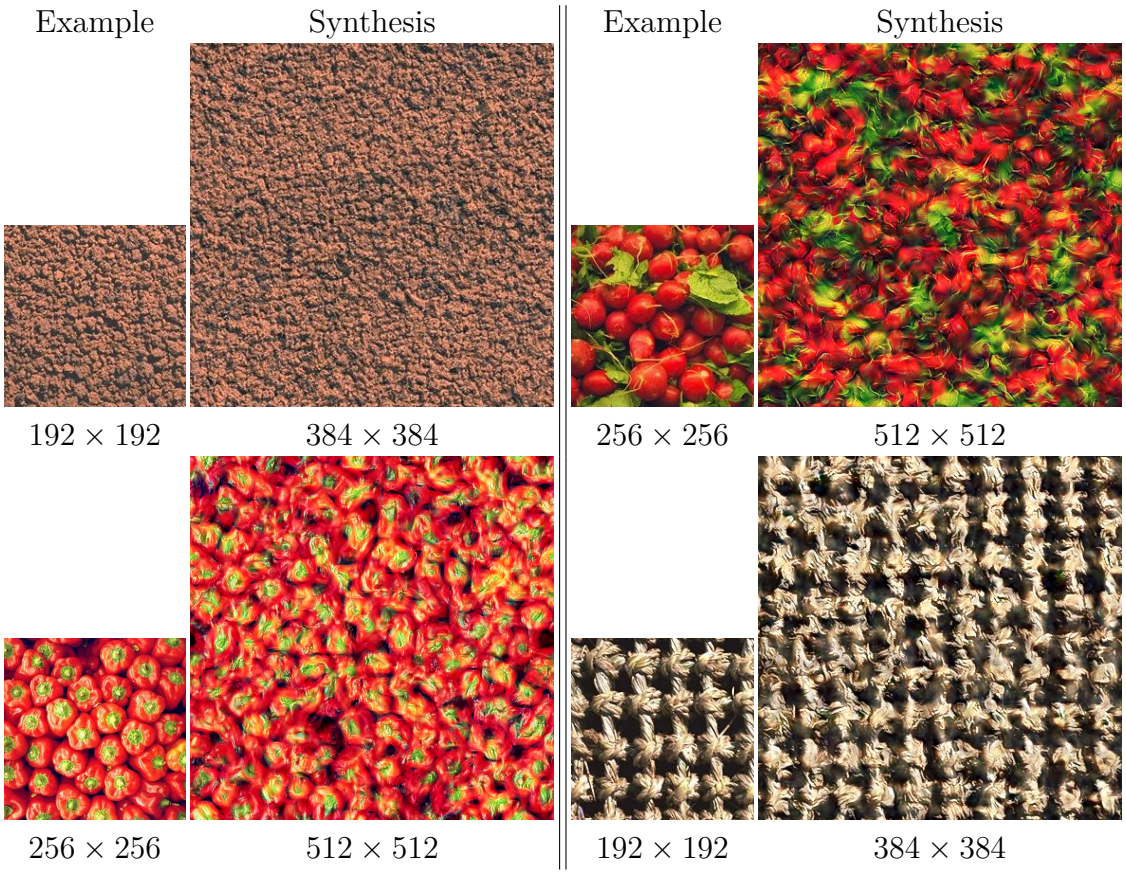



Figure 2.2 **Examples of texture synthesis using the statistical model of Portilla and Simoncelli [1].** Synthesized using the code provided by the authors with the following parameters: 4 scales, 4 orientations, neighborhood size of $7 \times 7$ pixels, and 50 iterations. The resolution in pixels is indicated at the bottom of each image.

spectrum accounts for the high frequency components of the texture and the overall regularity. Finally, the structure elements are characterized by the occurrences in an adapted patch dictionary. These parameters are integrated into an energy functional that asserts the quality of the synthesis. Texture synthesis is accomplished by finding a local minima of the non-convex energy via an averaged projections method.

Figure 2.3 shows examples of texture synthesis using the method of Tartavel et al. [2] with the code provided by the authors. The quality of the synthesized textures is similar to that of Portilla and Simoncelli [1] for stochastic textures (top-left example) and slightly better for textures with more structured patterns.

As shown in the last two methods detailed here, a major drawback of statistical matching texture synthesis is the difficulty to properly reproduce the *macro* structure elements, i.e., the elements that are recognizable by human inspection.

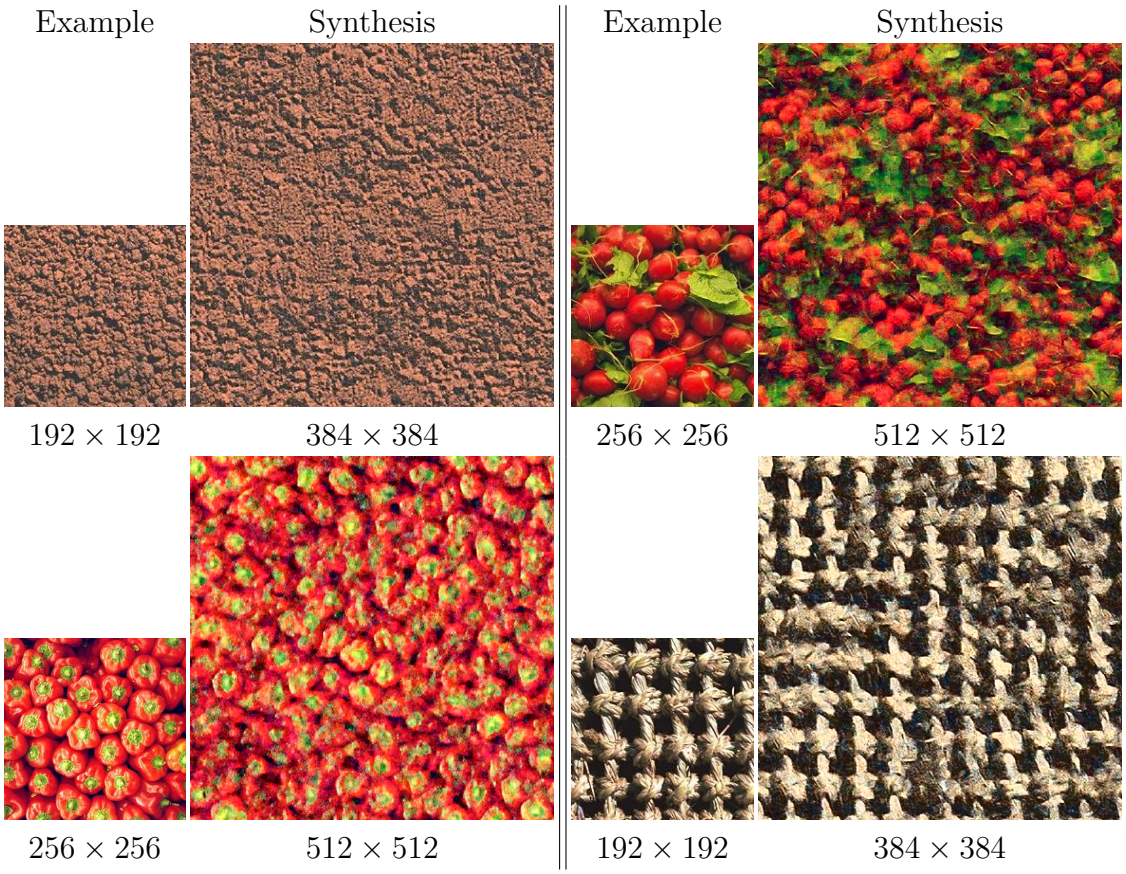



Figure 2.3 **Examples of texture synthesis using the statistical model of Tartavel et al. [2].** Synthesized using the code provided by the authors with the default parameters. The resolution in pixels is indicated at the bottom of each image.

### 2.1.2 Neighborhood-Based Texture Synthesis

The foundation of neighborhood-based methods is the hypothesis that each pixel in a texture has a Markovian property, i.e., each pixel's value can be determined given the values of the pixels in a small neighborhood around it. In practice this means that neighborhood-based methods focus on replicating the relationships between neighboring pixels in the example. The example serves as a catalog of all these possible local relationships for that texture, and new samples are synthesized by copying or combining pixels following that catalog.

We can classify neighborhood-based methods given the unit of synthesis they use. Seminal work synthesize the output one pixel at a time, subsequent methods use patches, and finally, texture optimization methods iteratively evolve all the output's pixels simultaneously. In the following paragraphs we discuss each of these categories. Additionally, we detail texture optimization [6] in Chapter 3 as it lays the base for our proposed 2D texture synthesis method.

### Pixel as Unit of Synthesis

Former neighborhood-based methods perform texture synthesis sequentially one pixel at a time. Using single pixels as the unit of synthesis gives these methods the flexibility to capture high frequency details from the example texture. When synthesizing a new sample, the initial point is either random values or a *seeding* copy of a small piece of the example. Then new pixels are synthesized by copying the value of a pixel from the example that has a similar neighborhood.

In the method of Efros and Leung [38] the pixels are synthesized in a circling outwards order from a *seed* in the middle. This strategy requires that only known (i.e., previously synthesized) neighboring pixels be considered in the comparison when looking for a similar neighborhood. In the method of Wei and Levoy [3] the synthesis order follows a scan line which results in the known pixels in a neighborhood to always be at the same positions. This simplifies the neighborhood comparison, allowing the model to use fixed L-shaped neighborhoods that only contain previously synthesized pixels. In order to handle large scale patterns, they propose a multi-scale strategy that first synthesize coarser resolution samples given a sub-sampled version of the example. Then each higher resolution neighborhood takes into account the *neighboring* pixels across scales.

The size of the neighborhood is a user specified parameter and it is usually set in a way that it fits the biggest pattern or regularity in the example. The comparison between neighborhoods generally uses the Euclidean distance and it is sometimes complemented with a Gaussian falloff weighting in order to give more importance to the closest pixels [38]. The search phase (i.e., looking for a similar neighborhood) involves most of the computation. The method of Efros and Leung [38] first finds several good neighborhood matches, then chooses one of them randomly. Subsequent methods focus on more efficient ways to find those matches. Wei and Levoy [3] accelerate the search using approximative methods and selecting a good match deterministically. Finally, several methods [4, 39, 40] exploit spatial regularity to partially bypass the search, simply using the neighboring values of previous matches.

Figure 2.4 shows examples of texture synthesis using the methods of Wei and Levoy [3] and Ashikhmin [4]. The results from [3] show the garbage regions—blurry or constant patches from the example that are enlarged during the synthesis—common to pixel-based methods. This is alleviated in [4] but with the downside of visible sharp transitioning edges where the large copied regions connect.

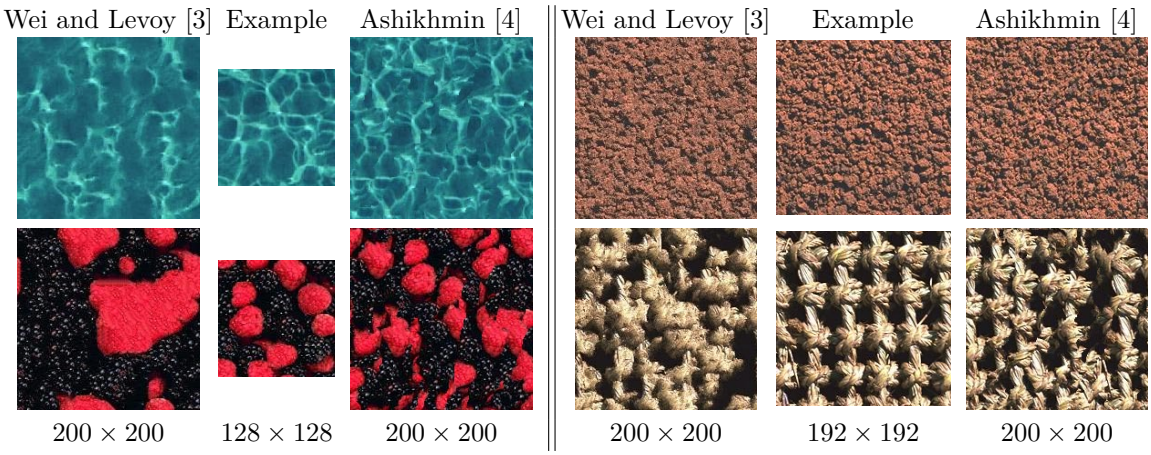



Figure 2.4 **Examples of texture synthesis using the methods of Wei and Levoy [3] and Ashikhmin [4].** Image resolution in pixels indicated at the bottom. Images from [4].

### Patch as Unit of Synthesis

Typically, a successful texture synthesized using a per-pixel method grows regions where neighboring pixels are also neighbors in the example. This indicates that modeling textures using a Markovian property assumption implicitly encourages spatial regularity in the source of the copied pixels. This characteristic motivates the use of patches as building blocks for a new texture. Intuitively, this strategy saves computation and guarantees that most of the local arrangements are well preserved.

Stitching patches together, however, requires a careful handling of the edges in order to avoid visible seams. The method of Xu et al. [41] produces a new texture sample by simply tiling and randomizing squared patches. It partially hides the seams using a smoothing filtering. This method is impressively fast but still suffers from visible seams.

More successful methods place patches sequentially [5, 42, 43] in a similar way to per-pixel methods, guided by the similarity in some overlapping regions. Liang et al. [42] propose to hide the possible remaining seams using a smoothing operation. The methods of Efros and Freeman [5] and Kwatra et al. [43] construct a boundary between the overlapping patches that properly hides seams.

These methods are usually faster than per-pixel methods, but have the downside of severe repetition due to the larger copied regions. This is illustrated in Figure 2.5 which shows examples of texture synthesis using the method of Efros and Freeman [5].

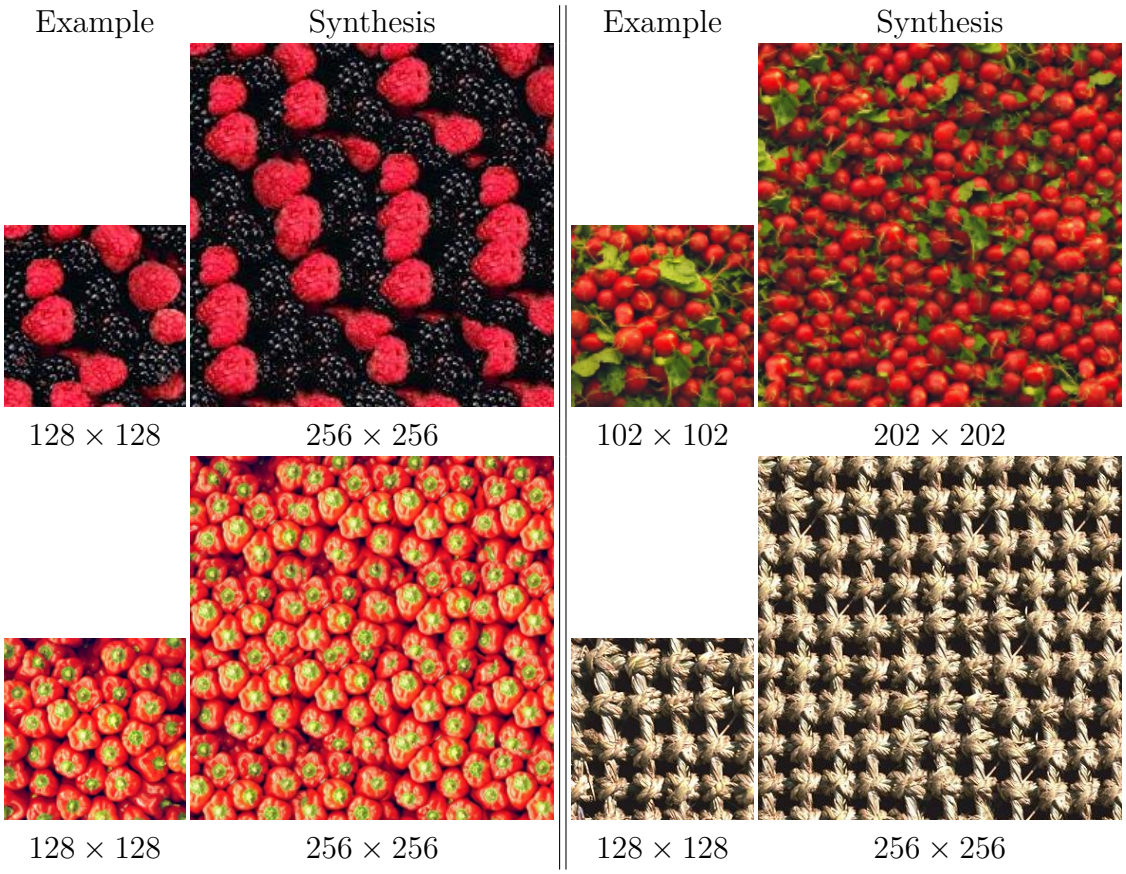



Figure 2.5 **Examples of texture synthesis using the patch-based method of Efros and Freeman [5].** Images taken from the article [5]. The resolution in pixels is indicated at the bottom of each image.

### Texture Optimization

Texture Optimization methods, initially proposed by Kwatra et al. [6], formulate example-based texture synthesis as an optimization problem. They integrate the Markovian assumption in the model using a dissimilarity measure that accounts for the difference between each patch in the sample being synthesized and its most similar patch from the example texture.

The optimal solution of this optimization problem is the example image itself but it is not the desired result for texture synthesis. Texture optimization methods focus instead on finding a local minimum, which is the typically reached solution due to the non-convex nature of the dissimilarity measure. An image that reaches a low value for the dissimilarity measure is formed by patches that are all similar to some patch in the example. By doing so, the image is expected to reproduce the visual characteristics of the example texture. Although this is an elegant formulation, it has some shortcomings. As an example, a typical defect is obtained when the example texture contains *flat* or constant patches. Because flat patches account for

a small dissimilarity value, these constant regions can be enlarged in the synthesized samples.

In order to obtain a solution, these methods usually employ an iterative alternate minimization scheme that consists of two steps. In the search step, they find the most similar patch in the example for each patch in the sample being synthesized. In the aggregation step the value of each pixel is computed as a weighted average of the overlapping values at each location. Those *candidate* values come from the patches from neighboring pixels. Starting from a random initialization, this procedure iteratively evolves the whole sample until convergence in a multi-scale way using the output at a coarse scale as the initialization for a subsequent finer scale.

Figure 2.6 shows some examples of synthesis from the method of Kwatra et al. [6]. This original formulation reaches good quality results thanks to the use of large size of patches at low resolutions, which guides the synthesis away from a constant solution. The downside of this implementation is the severe repetition similar to methods that stitch patches together.

Texture Optimization is a widely used framework for texture synthesis methods. Han et al. [44] use it to synthesize textures on meshes and accelerate it by taking the best candidate in the aggregation step instead of averaging all the overlapping candidates and looking only for pre-computed good matches for each pixel in the search step. Kaspar et al. [45] complement Texture Optimization with several techniques to improve the visual quality. Kopf et al. [17] and Chen et al. [46] record the utilization of pixels from the example in order to guide the synthesis towards a more uniform utilization of its richness. They also perform solid texture synthesis as will be discussed in the next section. Elad and Milanfar [47] also use a Texture Optimization formulation for the problem of style transfer, i.e., transferring some artistic technique of an image to another.

### 2.1.3 Texture Synthesis Using Deep Convolutional Neural Networks

The popularity of Convolutional Neural Networks [48] (CNN) on many tasks involving images shows that they are capable of learning meaningful representations [49]. Models for example-based texture synthesis mainly leverage that trait for the task of evaluating the quality of a sample being synthesized, i.e., to evaluate the similarity with the example, and to produce an error signal that can be used to improve it. In contrast to statistical matching methods, where the representation of textures is manually crafted, methods that use CNN rely on a learnt representation, usually derived from training on a large number of images.

A second contribution of CNNs to example-based texture synthesis is the use of a *generator* network that can be trained to convert a noise input into a new sample texture. This replaces

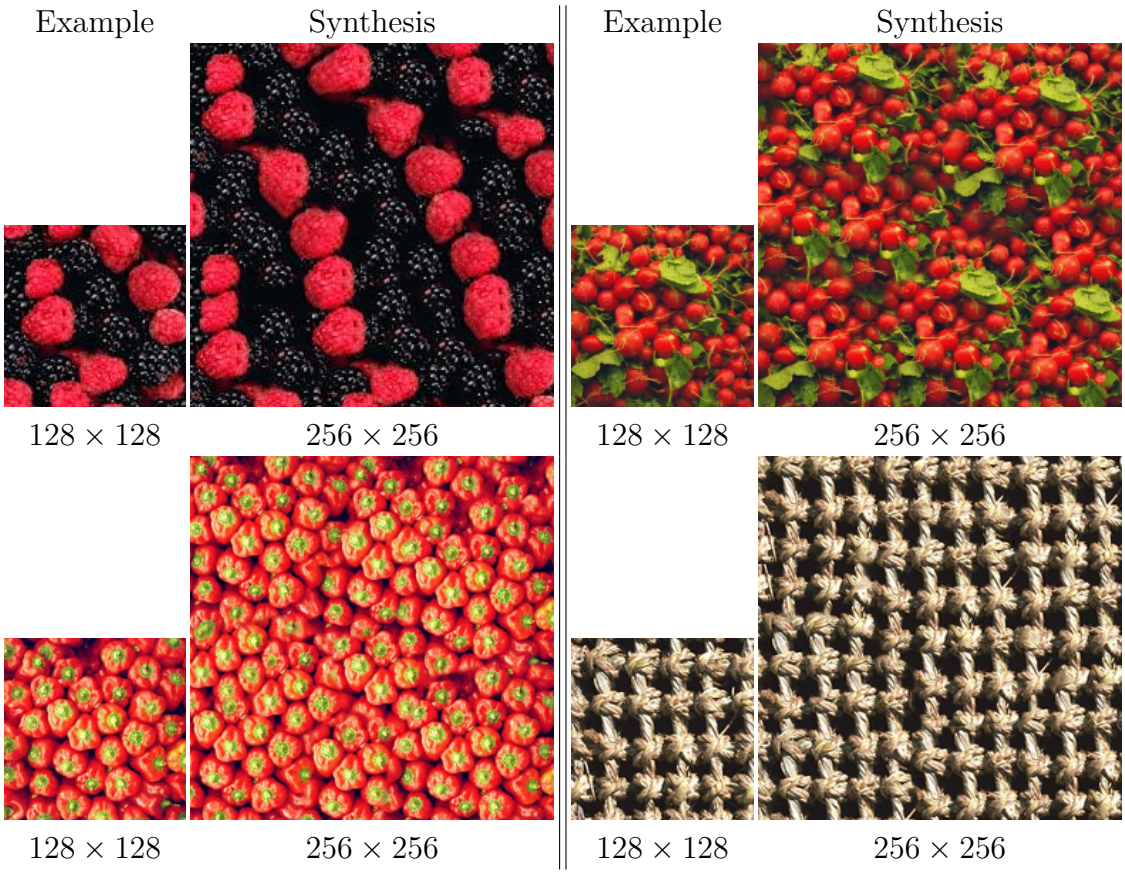



Figure 2.6 **Examples of texture synthesis using the method of Kwatra et al. [6].** Images from [6]. The resolution in pixels is indicated at the bottom of each image.

the synthesis algorithm by a series of mathematical operations (usually convolutions, up-sampling, and non-linear functions) represented by the network. The training is done only once, after that, the generator can synthesize new samples at low computational cost.

We classify the methods that use CNN for texture synthesis into three main settings: *Texture Optimization Using CNN* uses a fixed *descriptor* network and formulates texture synthesis as an optimization problem, *Texture Networks* train a *generator* using a similar fixed descriptor, and finally *Generative Adversarial Networks* (GAN) use a generator and a *discriminator* networks, both of them trainable. We summarize these settings in the following paragraphs.

Additionally, in the beginning of Chapter 4 we analyze in detail the texture networks method of Ulyanov et al. [8] as it serves as the foundation for our proposed solid texture synthesis method detailed in the same chapter.

### Texture Optimization Using Convolutional Neural Networks

Gatys et al. [7] first showed that the feature responses in the internal layers of a trained *descriptor* CNN serve as a powerful representation for textures. Inspired by the work of Portilla and Simoncelli [1], they characterized a given texture using the *Gram matrices* (detailed in Chapter 4) of the feature responses induced by the example on the descriptor network. They formulate texture synthesis as an optimization problem with the goal of finding an image that produces Gram matrices as close as possible to the target produced by the example. Deep learning software makes it easy to use a gradient-based algorithm for this task. This way of measuring the similarity between two textures was later coined as *perceptual loss* by Johnson et al. [25] and we formally define it in Chapter 4.

Gatys et al. [7] obtained outstanding visual quality textures. Figure 2.7 shows some texture synthesis examples. This method outperforms most of the state of the art, particularly in photo-realistic textures. Similarly to previous methods, a common flaw is in the long distance correlations which are not fully captured by the model.

The descriptor network used by Gatys et al. [7] is VGG-19 [50]. It was originally created for the task of image classification. It consists mainly of convolution layers with non-linear activations, it uses pooling to reduce the spatial size of the image and three fully connected layers to perform the classification. Overall it has around 144 Million parameters which makes it one of the largest popular classification networks. The version commonly used for texture synthesis was trained on the ImageNet database [51] with $\sim 500,000$ natural images from 200 categories. Theoretically the descriptor can be any neural network that produces a good characterization of textures but the exclusive and extensive use of VGG-19 in subsequent works shows that this model is exceptional at this task. Gatys et al. [7] also experimented with the caffe reference network [52] reporting lesser quality results. For studies of alternative discriminators, see Ustyuzhaninov et al. [53] and Zhang et al. [54].

The method of Gatys et al. [7] was extended by adding some classical statistical descriptors in order to better preserve long distance correlations [55, 56]. Overall, texture optimization using CNN methods produce impressive results but have the drawback of requiring to solve a computationally expensive optimization problem to produce each new sample—taking minutes on a Graphics Processing Unit (GPU).

### Texture Networks

Texture Networks methods, first proposed by Johnson et al. [25] for style transfer and Ulyanov et al. [8] for texture synthesis, pair a discriminator and a generator networks with

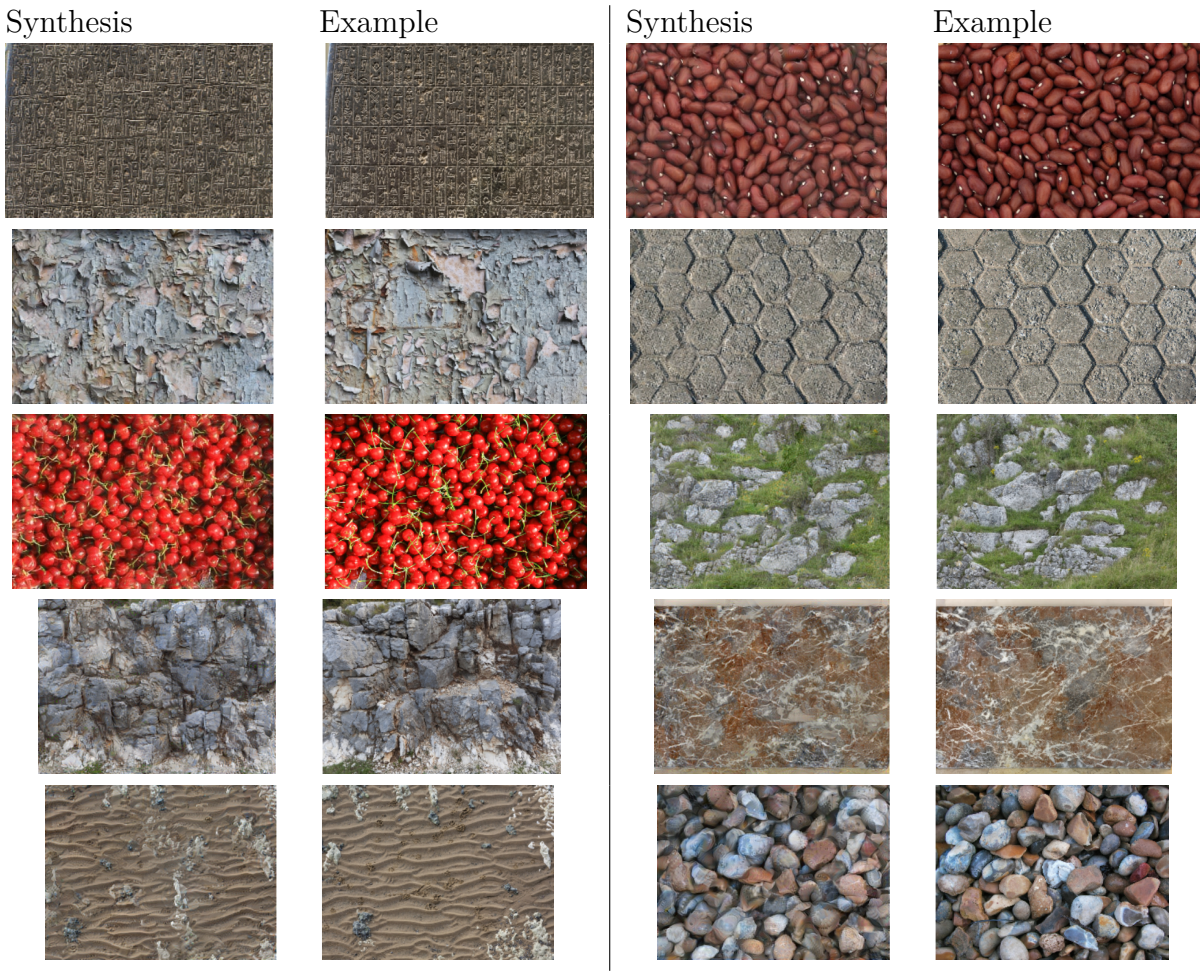



Figure 2.7 **Examples of texture synthesis using VGG-19 as descriptor network with the method of Gatys et al. [7].** Images from the authors website `http://bethgelab.org/`.

the goal of synthesizing new samples more efficiently than Texture Optimization Using CNN methods. Instead of solving an optimization problem for each sample, these methods *train* the generator network to synthesize images that are similar to the example texture. Once the training procedure is completed, the generator can synthesize new samples of the texture with a small computational cost.

The generator usually consists of a series of convolution and non-linear operations that processes a noise input to produce a color image. The discriminator network is usually VGG-19 and serves to evaluate the similarity between the generator's outputs and the example texture. Training consists of generating batches of images and measure the similarity with the example using the perceptual loss and then using a gradient-based optimization method to update the parameters of the generator towards producing better samples.

Figure 2.8 shows a comparison of the visual quality between the method of Gatys et al. [7], which directly optimizes an image using VGG-19, and the method of Ulyanov et al. [8], which trains a generator network using VGG-19. Texture Networks methods usually reach a lower visual quality that Texture Optimization Using CNN methods, but they perform texture synthesis at a much lower computational cost.




Figure 2.8 **Comparison of texture synthesis results with the methods of Ulyanov et al. [8] and Gatys et al. [7].** Images from [8].

The Texture Networks strategy consisting of training a generator network using VGG-19 as descriptor network is now a common framework for several texture synthesis methods. Different architectures for the generator network and training strategies have been proposed to improve the visual quality [57] or to make it possible to synthesize multiple textures with the same generator network [58].

Our method proposed in Chapter 4 extends Texture Networks for example-based solid texture synthesis. Before detailing the proposed method in Chapter 4 we detail the foundations of the method of Ulyanov et al. [8] and study some of its characteristics.

### Texture Synthesis Using Generative Adversarial Networks (GANs)

Generative Adversarial Networks [59] (GANs) are one of the most prominent generative models for natural images. They standout because they bypass the task of manually designing

a training loss function that measures the visual quality of a generated image. This is a difficult task because the quality depends on the type of images at hand and it usually needs to capture the content and relationships between the samples in a large reference set. In GANs, a deep network (almost always a CNN) called *discriminator* performs this evaluation by giving a probability of the image being real, i.e., belongs to the reference set.

Similarly to Texture Networks methods, the discriminator's output is used to train a generator network to synthesize images with the right content. In this case, however, the discriminator is trained along the generator. The training process allows the discriminator to learn a meaningful representation for the images in the training set and helps it better recognize the fake samples. At the same time, the better the discriminator performs, the more it pushes the generator to produce high quality images.

GANs are extremely popular for image generation and recent algorithms are capable of producing some unnoticeable fake images. Figure 2.9 shows some images from the GANs method of Karras et al. [9]. The proportion of GANs models for the task of texture synthesis is, however, small. A reason for this is that in example-based texture synthesis, usually a single image defines the whole visual characteristics to reproduce. A typical GANs model would most likely learn to produce a simple copy of the example, which is not a satisfactory result.

Jetchev et al. [10,60] proposed GANs models for texture generation and texture mixing where the generator learns to synthesize textures that reproduce the patterns of one or more examples. They use random sub-images (large patches) of the examples to introduce variability to the set of real images. In this work the discriminator's goal is to give a probability for each patch in an image to be real, which allows the texture mixing. In periodic spatial GANs [10] (PSGAN) they added a *global* component to the random input that serves to control the texture mixing and interpolation after training. Figure 2.10 shows some novel textures generated with the PSGAN model by using a spatially piece-wise constant random input. The visual quality of the results is inferior to methods using VGG-19 for the description of textures, but this model stands out for its capabilities for mixing and interpolating textures.

Zhou et al. [61] use a combination of VGG-19 and adversarial discrimination and achieve impressive results for the synthesis of non-stationary textures. This is a more general problem seldom addressed in the literature and it requires extra assumptions about the behavior of the texture at hand.

For the particular task of (stationary) texture synthesis, where the whole distribution has to be defined by a single example, purely GAN-based models produce inferior results than Texture Networks methods.



Figure 2.9 **Examples of synthetic images generated with the GAN method of Karras et al. [9], trained on natural images**. Image from the article [9], each example has a resolution of 256 × 256 pixels.

## 2.2 Solid Texture Synthesis

The process of applying a 2D texture to a 3D model requires a complex parametrization —usually produced manually—to associate each vertex to a 2D coordinate. Once this parametrization is established, the on-line application of the texture is efficient because 2D textures occupy a small amount of memory and graphics hardware facilitates the fast retrieval of the color at each point in the 3D surface.

In this context, solid textures have a significant geometric advantage over 2D textures. They avoid the need for a parametrization and its accompanying distortions, regardless of the 3D model's geometry. The downside of solid textures is that current models usually make a trade-off between visual quality and on-line application efficiency. Particularly, *procedural* textures are efficient but limited in visual quality and require user's time and skills, and example-



Figure 2.10 **New textures synthesized with Periodic Spatial GANs [10]**. The training is performed using the *scaly* category of the DTD dataset [11].

based solid textures offer automation and high visual quality at the cost of a large memory footprint which makes them inefficient. In this section we summarize these two different instances of solid textures, and explain how several principles of 2D texture synthesis have been extrapolated for the solid case. The proposed model in Chapter 4 closes the gap by using a deep CNN generator that efficiently generates high quality solid textures on-demand.

### 2.2.1 Procedural Solid Textures

Procedural solid textures were first proposed by Peachey [62] and Perlin [12] as mathematical models that produce a color value given a set of 3D coordinates. They designed procedural models for textures resembling different materials like marble, wood, clouds, etc. These models are fast and memory efficient. They integrate easily in the rendering process and allow the synthesis of textures on the visible points of a surface on-demand. Figure 2.11 shows a characteristic example from [12]. This procedural texture sufficiently resembles marble—one of the first successful usage of procedural textures.

The principles of procedural solid textures are widely used in computer graphics. Yet, creating the procedural model for a given texture is a manual experimental process that requires time and skills. Figure 2.12 shows some more marble-like procedural textures created by a



Figure 2.11 **Procedural texture from Perlin [12] applied to a 3D model.**

skilled artist using a recent sophisticated graphics software called substance designer. An alternative to the manual design is to use an example image to guide the expected visual result. In this context, the goal is not to precisely match the visual characteristics of the example but to guide a noise function to produce similar colors or patterns. Dischler et al. [63] used spectral information from a 2D example to parametrize a solid texture procedural model.

The visual quality of procedural textures is generally limited. They are an efficient choice for highly stochastic textures but they usually fail at reproducing textures with distinguishable patterns.

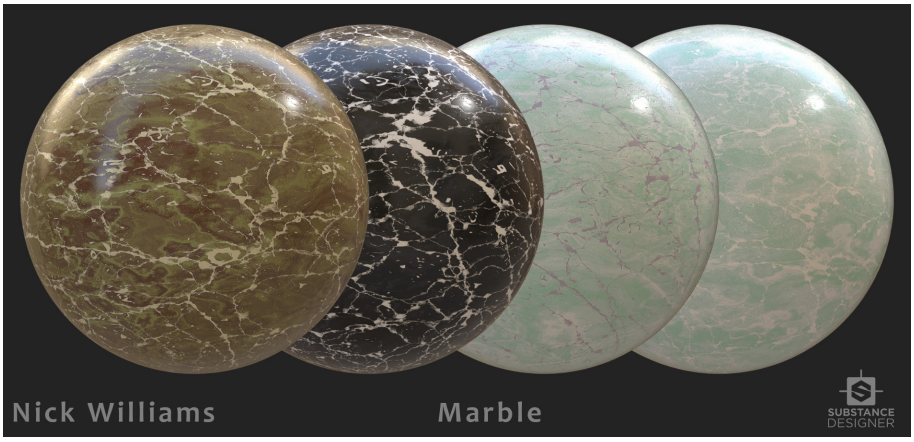



Figure 2.12 **Procedural textures created using *substance designer* (software)**. The textures were designed by Nick Williams (`https://www.artstation.com/artwork/lOyRY`).

### 2.2.2 Example-Based Synthesis of Off-line Solid Textures

Solid texture methods that achieve better visual quality than procedural approaches on a wider set of types of textures generally focus on example-based synthesis of samples off-line. These solid textures take the form of dense arrays of color values associated to 3D coordinates, similar to raster images. This results in a large memory footprint that makes this form of solid textures difficult to integrate in a graphics pipeline. It is, however, interesting to study these methods as some of them could be implemented for on-demand evaluation, as the method of Dong et al. [64] that we will describe later in this section.

Example-based solid texture synthesis methods usually extend a 2D model, so we can also classify them as either statistical matching or neighborhood-based strategies (Section 2.1).

Because *real* solid texture examples are difficult to obtain, these methods use 2D examples to define the visual characteristics of the solid texture. Additionally, few 2D methods can be trivially extended to generate solid textures from 2D examples. Heeger and Bergen's [29] method can only partially transfer some 2D visual characteristics trivially to 3D. Thus, the usual definition of the problem is: given a set of 2D examples $\{u_d, d \in \{1, \ldots, D\}\}$, produce a solid texture $v$ whose cross-sections $v_d$ along slicing direction $d \in \{1, \ldots, D\}$ reproduce the visual characteristics of $u_d$. Generally $D = 3$ for three orthogonal axes. Figure 2.13 shows the conceptual description of the solid textures via a set of examples for three orthogonal axes. This slice-based definition allows us to better transfer visual characteristics from the

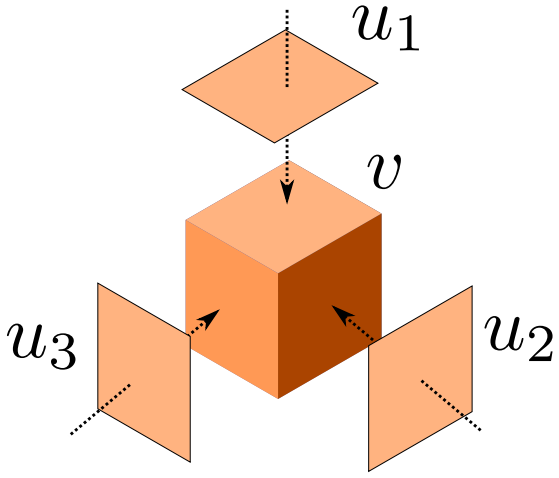



Figure 2.13 **Formulation of solid texture synthesis from 2D examples.** A set of 2D images $\{u_d, d \in \{1, \ldots, D\}\}$ (in this case $D = 3$) defines the expected visual characteristics of all the cross-sections in direction $d$. Image inspired from [13].

example to the solid texture. Yet, it is still a difficult setting because 3D patterns have to be inferred from 2D examples. For isotropic textures, it seems fair to assume that the features propagate equally in the 3D space, therefore, we can use the same example for the three slicing directions ($u_1 = u_2 = u_3$). In order to model anisotropic textures, we often need to carefully choose the example's orientation, so the patterns are compatible. To obtain a successful anisotropic texture, current methods often use only two directions ($D = 2$). Figure 2.14 shows some examples from Wei's work [13] that exemplify different outcomes of modeling an anisotropic texture. A complication with example-based solid texture synthesis is that there is no guarantee of the existence of a 3D *version* of the patterns defined by the examples. This complication will be discussed in Chapter 4.

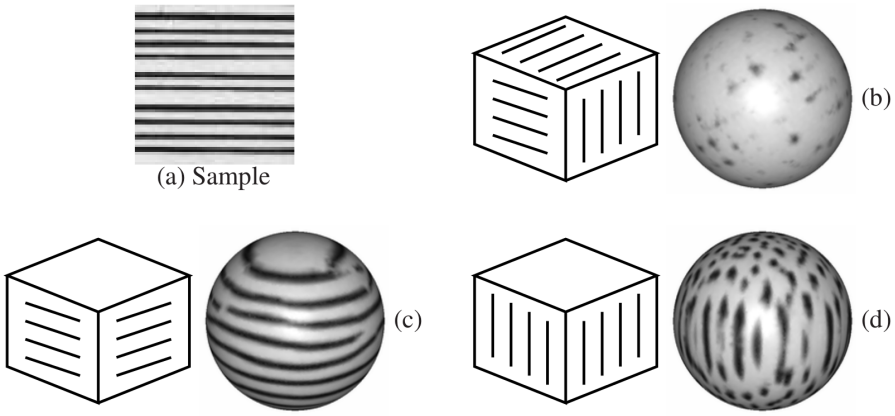



Figure 2.14 **Effects of the examples configuration.** Figure from [13]. For anisotropic textures it is important to carefully select the number of directions and the orientations of the example textures. (a) Example texture used along all axes. (b) Solid texture resulting from an inconsistent setting of the patterns. (c) Solid texture resulting from a setting with patterns horizontally aligned. (d) Solid texture resulting from a setting with patterns vertically aligned. All samples are synthesized using the method of Wei [14].

Thus, contrary to procedural models, example-based methods require little manipulation and expertise from the user, usually only requiring the selection of compatible examples.

Because example-based solid texture synthesis methods usually share some principles with 2D methods, here we explain the two main mechanisms to produce solid textures using those principles: slice-based synthesis and 3D neighborhoods.

### Slice-Based Synthesis

Statistical methods usually can be adapted to perform solid textures using a slice-based strategy. Figure 2.15 shows an overview of this strategy. It consists of iteratively evolving

the sample by independently synthesizing its slices. Starting from a random initialization, at each iteration all the 1-voxel thick slices along all the slicing directions are synthesized independently and then aggregated. The information in the slices is propagated thanks to the aggregation, and the sample is expected to converge. However, this requires that the examples along different directions are compatible.

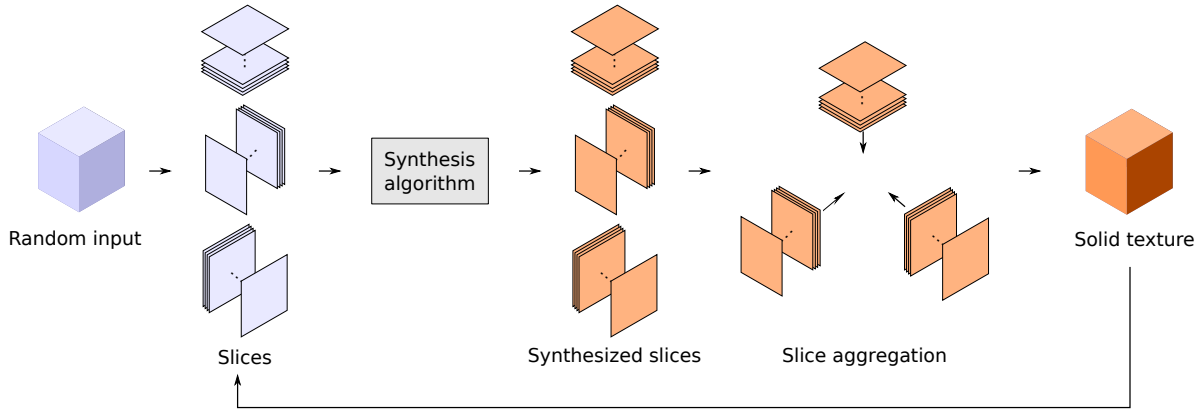



Figure 2.15 **Slice-based strategy for solid texture synthesis.** The solid texture evolves iteratively from a random initialization. At each iteration: 1- The solid is decomposed on all its slices, 2- The slices are modified to better match the statistics of the example, 3- The slices are aggregated.

Heeger and Bergen [29] use *histogram matching* directly in the Laplacian pyramid along with a slice-based synthesis in the steerable pyramid representation. This method averages the colors of the four solids to generate the output texture. This method is limited to isotropic solids. Qin and Yang [15] extended their work on 2D texture synthesis [65] in a similar slicing scheme. Their method iteratively updates the voxels' colors so that each slice's co-occurrence matrices match those of the correspondent example texture. This method can use many slicing directions and a different example for each. In the work of Dischler et al. [66,67] all the possible 2D slices along three directions are iteratively modified with spectral and phase processing to be similar to the given example, and then they are aggregated by averaging the overlapping voxel's colors. In [66] the authors address the anisotropic solid texture generation by carefully choosing the number of directions used and the orientation of the anisotropic example. Generally, these methods do not produce the best visual quality among example-based solid texture methods but they require less manipulation compared to procedural textures.

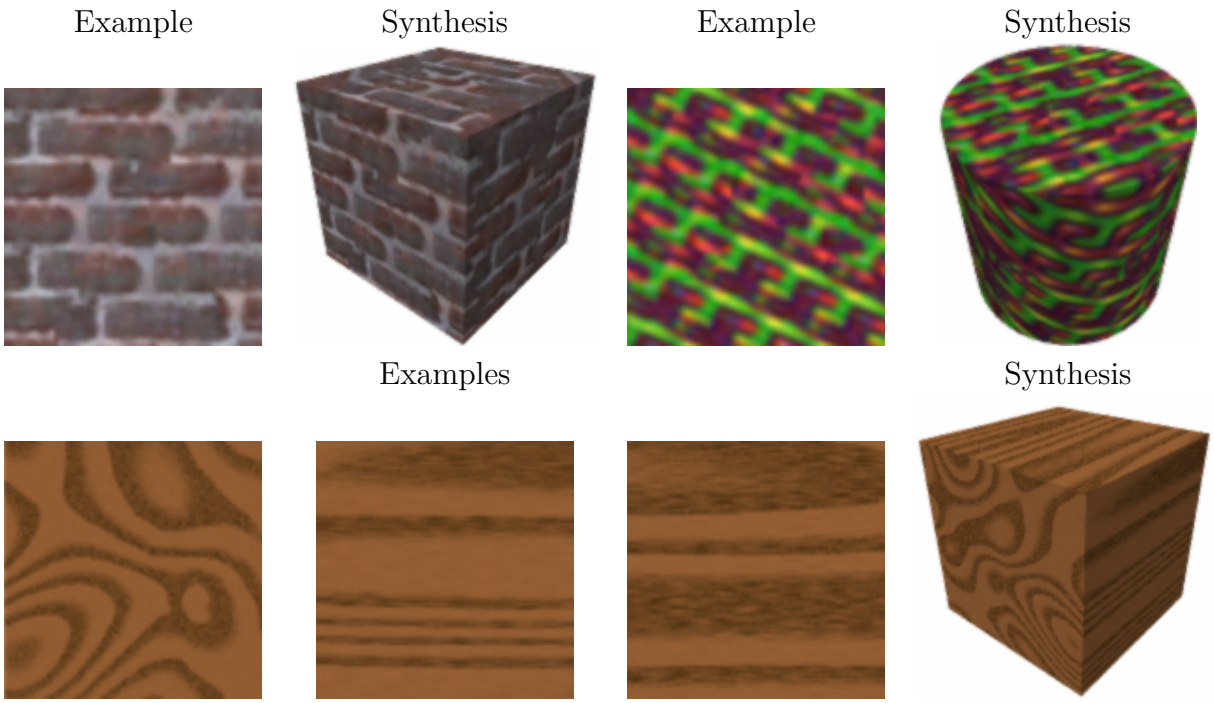



Figure 2.16 **Examples of solid texture synthesis using the method of Qin and Yang [15].** The top row shows two cases where the same example is used along three orthogonal directions. The bottom row shows one case where one different example is associated to each orthogonal direction. Images from [15].

### 3D Neighborhood

Wei [14] proposed a per-voxel strategy for synthesizing solid textures from multiple sources. The strategy is similar to the 2D method [68] in that each of the voxels is synthesized sequentially by comparing its neighborhood with the examples. For solid textures, he uses a simplified *3D neighborhood* where only three orthogonal oriented neighborhoods are considered, see Figure 2.17. Each oriented neighborhood is associated to one example texture. In order to synthesize a new voxel, the algorithm needs to find a matching pixel for each oriented neighborhood. If only one example is used, the voxel takes the value of the best matching pixel. If more than one example is used, the value of the voxel is selected by alternating between finding the set of best matches and aggregating them using a weighted average. Kopf et al. [17] extended the Texture Optimization [6] strategy to perform solid texture synthesis using similar 3D neighborhoods. They iteratively evolve a solid texture from a random initialization. Each iteration consists of two steps. In the search step, for each voxel in the solid texture, the algorithm finds the best match for each oriented neighborhood in its associated example texture. In the aggregation step, at each pixel location it performs a

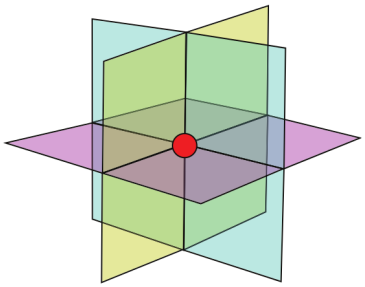


Figure 2.17 **3D neighborhood limited to neighbors in three orthogonal planes centered around one voxel.** Image from the solid texture synthesis survey of Pietroni et al. [16].

weighted average between all the overlapping values at all the orientations. Chen et al. [46] complemented the method of Kopf et al. [17] by adding some strategies to more evenly copy the patterns and colors from the examples.

Among example-based solid texture synthesis, neighborhood-based methods deliver the best visual quality, specifically the methods that use the Texture Optimization strategy [17, 46]. Figure 2.18 shows some examples of solid texture synthesis using the method of Kopf et al. [17]. These examples show the capability of the model to reproduce isotropic patterns in the 3D setting.

Off-line solid textures techniques are not used in practice given their large memory footprint. Integration in the graphics pipeline is seldom addressed by example-based solid texture synthesis methods. For every method we discussed here, either iterative or sequential, the value of the output at a given 3D coordinate has dependency with values in the whole texture. This prevents these methods to perform of on-demand evaluation. The following method takes a step forward to make example-based solid texture synthesis more efficient by enabling on-demand evaluation.

### On-Demand Example-Based Solid Texture Synthesis

Dong et al. [64] proposed a neighborhood-based method capable of on-demand evaluation. They extended the method of [69] to perform solid texture synthesis using 3D neighborhoods. They obtained fast computation times thanks to the pre-computation of compatible 3D neighborhoods. The on-demand capability allows them to synthesize textures on-line in the graphics pipeline. By synthesizing only the voxels that intersect the surface to be textured. This method, however, makes a trade-off between visual quality and efficiency, limiting its applicability for high resolution textures. Compared to Texture Optimization,

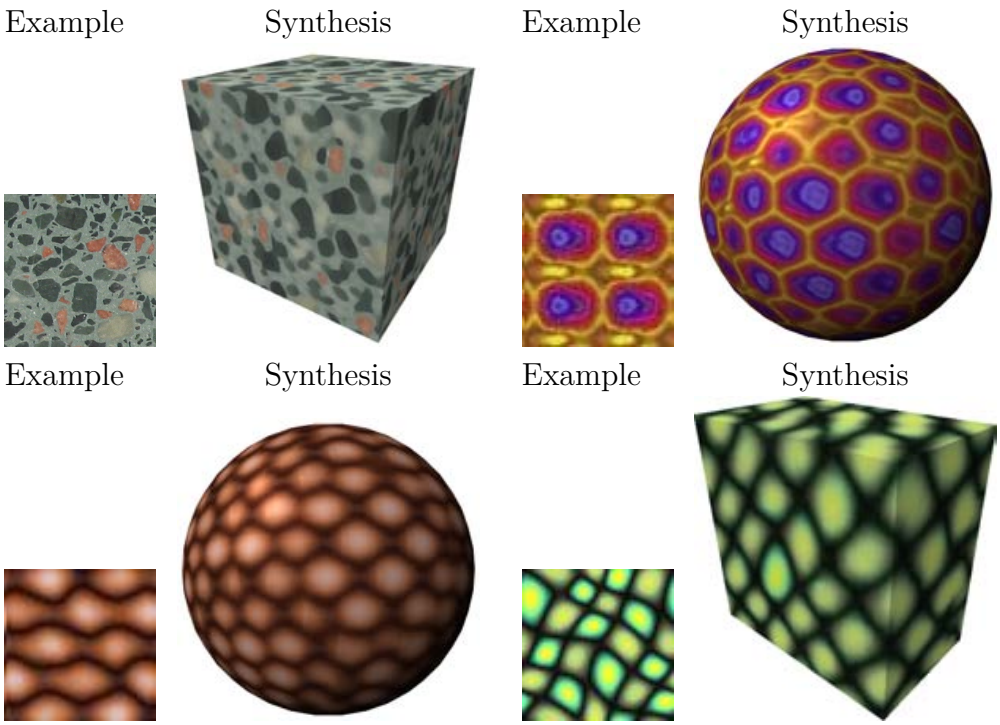



Figure 2.18 **Examples of solid texture synthesis using the method of Kopf et al. [17].** In all the cases the same example is associated to the three orthogonal directions. The spheres are *carved* out of cubic arrays of textures. Images from [17].

this method delivers lesser visual quality and it has higher computation complexity than procedural methods. By contrast, the method we propose in Chapter 4 is capable of on-demand evaluation at a lower computation complexity and without sacrificing visual quality.

## 2.3 Quality of Synthesized Textures

The quality of a synthesized sample texture is typically evaluated by human visual examination. The subjective assessment of whether a synthesized sample is successful or not has naturally evolved as better texture synthesis methods have been proposed. Nowadays, CNN-based methods like the one of Gatys et al. [7] produce such good quality results that it is sometimes difficult to identify the example from the synthesized texture (see Figure 2.7). These results seldom produce eye-catching artifacts for most types of textures.

Rating the quality among different recent methods requires a careful examination of the samples, usually focusing on determining which method introduces less artifacts. Swamy et al. [24] identified four types of undesirable artifacts that human observers find the most salient. They point out that the most important is the lack of structure in the patterns, a

common flaw of statistical constraining methods (Section 2.1.1). Other relevant artifacts are excessive repetition, common in neighborhood-based methods (Section 2.1.2), misalignment of the texture patterns, and blurring.

Automatic evaluation of a synthesized texture is a difficult problem. The challenge is similar to statistical matching texture synthesis methods (Section 2.1.1), that is, to devise a set of measurements that summarizes texture similarity in a way that is coherent with human visual perception. This creates a Goodhart's law case. Each set of features used to evaluate the quality can be used as a goal for a new method, creating textures that perform better but not always caring about better visual quality. Texture classification methods are not capable of performing the comparison between samples related to the same example texture. De et al. [70] propose a relative comparison using measures on the co-occurrence matrices but their method is limited to toroidal examples as it requires an *ideal* synthesized sample created by tiling the example texture. Additionally the method is not successful on textures with visible structure/patterns. Similarly, the parametric assessment method of Swamy et al. [24] uses statistical measurements and fails to take into account the local structure—which they report to be the most salient undesirable artifact for human evaluators. Lin et al. [71] propose a criterion for evaluating the reproduction of the long distance relationships in regular and near-regular textures, i.e., with long, aligned edges of the patterns. This method, however, still requires human assistance with the lattice extraction.

In a more general problem of evaluating the quality of GAN's samples two criteria are commonly used. The Inception Score [72] (IS) uses the pre-trained classification network Inception [73, 74] to measure the capability of the model to generate unambiguous samples from different categories. Diversity is a meaningful criteria in texture synthesis, but is different from the category-wise diversity desired for GAN's. The Fréchet Inception Distance (FID) proposed by Heusel et al. [75] directly compares a set of generated samples to a set of real samples using the statistics of the feature maps in the Inception network. It requires a large number of samples to estimate the feature distribution.

There are several automatic tools that can be used to assist human examination in order to perform a more objective evaluation and comparison of synthesized textures. The difference between the color histograms of the synthetic and example textures can tell us if the method is capturing the right proportion of colors, however, it does not give any information about the local relationship between pixels.

The stress test proposed by Kaspar et al. [45] consists in repetitively performing texture synthesis using the last synthesized sample as the example for the next repetition. This quickly highlights if a method introduces artifacts, as they more likely aggravate with rep-

etition. It also informs us about the uniform utilization of the patterns in the example, a good method can sustain more repetitions while delivering acceptable results. Conversely, under-used patterns and colors will tend to disappear.

The correspondence maps, used in some texture synthesis methods [69] to control spatial regularity, can be used to visualize the degree of repetition and innovation on a synthesized sample. A position map is an image, associated with the example, where each pixel has a different color depending on its position and the color varies smoothly along the axis in the image. This makes easy to trace the source position of a given color (see Figure 2.19). Correspondence maps highlight the source of each pixel in a synthesized sample. They are images where each pixel takes the color from the position map associated to the copied pixel from the example. Tartavel et al. [2] use empirical correspondence maps to highlight the innovation capacity of a given method. In this case the source of each synthesized pixel is unknown, but it is estimated by finding the pixel with the most similar neighborhood in the example. Figure 2.19 shows an example of texture synthesis with its position maps and its correspondence map. A smoother correspondence map means that larger pieces of the




Figure 2.19 **Example of the correspondence map of a synthesized texture.** Large smooth regions indicate the verbatim copy of pieces from the example. Stochastic regions indicate innovation.

example are reproduced in the synthetic sample, this naturally means that the patterns are successfully reproduced but it also means that the method has a low capability of innovation. A lack of innovation results in synthetic samples with low variability and often with visible repetition, particularly when synthesizing larger samples than the example. Figure 2.19 is an example of this behavior. The correspondence map in this synthesis example shows a few large regions. The synthesized sample shows a rather good visual quality but with a striking impression of repetition given by the green leaf which is repeated eight times for an image that is only four times larger than the example. In the following chapters we mainly rely on

human visual examination for determining the quality of the textures. We complement visual observation with correspondence maps to highlight innovation capacity and with first-order histograms to confirm our statistical hypothesis.

# CHAPTER 3 STATISTICALLY GUIDED PATCH-BASED 2D TEXTURE SYNTHESIS

In this chapter we propose a patch-based optimization method for 2D texture synthesis that integrates a statistical constraint in the patch distribution. It builds upon the Texture Optimization framework [6] mentioned in the previous chapter (Section 2.1) and further detailed here in Section 3.1. Patch-based 2D texture synthesis methods that deliver the best visual quality integrate a uniformity mechanism to the synthesis process that encourages an even utilization of the patches from the example. We show three of these mechanisms in Section 3.2. The proposed method, detailed in Section 3.3, goes further in that direction. We claim that complementing the spatial coherence with a matching of the patch distribution of the example results in a high visual quality and a robust synthesis. Finally, in Section 3.4 we propose two alternative models that explore relaxations on the distribution matching and optimality respectively.

## 3.1 Texture Optimization

We summarized Texture Optimization in Chapter 2. It formulates example-based texture synthesis as an optimization problem which results in a patch-based method that iteratively evolves the new sample as a whole. This contrasts with the sequential strategy of previous methods. Here we discuss its fundamentals as proposed by Kwatra et al. [6] since it lays the foundation for our proposed method. We detail the variational model and the optimization strategy, and show some results highlighting its common flaws.

### 3.1.1 Nearest Neighbor Model

Similarly to previous neighborhood-based methods, the model of Kwatra et al. [6] aims at synthesizing a new sample whose patches are similar to the patches in the example texture. They formulate this as an optimization problem as follows.

Let $u_0 : \Omega_0 \to [0,1]^3$ be the example texture and $u : \Omega \to [0,1]^3$ be the synthesized sample, both using the normalized RGB color space. Let $\mathcal{P}(x)$ be the set of indexes of pixels in the squared neighborhood of $x$ of radius $w$,

$$\mathcal{P}(x) = \left\{ x + t,\ t \in \{-\tfrac{w}{2}, \dots, \tfrac{w}{2} - 1\}^2 \right\},$$

so that $p(x) = u \circ \mathcal{P}(x)$ is the patch of $u$ in position $x$. In the pioneering work of Kwatra et al. [6] the energy to minimize accounts for the distance between every patch in the texture being synthesized to their nearest neighbor in the input example. Formally:

$$\min_{u} \min_{\sigma} \quad \sum_{x \in \Omega^{\downarrow}} \|u \circ \mathcal{P}(x) - u_0 \circ \mathcal{P} \circ \sigma(x)\|_p^r \tag{3.1}$$

where $\sigma : \Omega^{\downarrow} \to \Omega_0^{\downarrow}$ is a mapping between the indexes of the patches in the synthesized sample and the example. The patches are considered to come from grids $\Omega_0^{\downarrow}$ and $\Omega^{\downarrow}$, subsets of $\Omega_0$ and $\Omega$ respectively, where the indexes are regularly spaced with a stride of $\frac{w}{4}$. Kwatra et al. [6] specifically study values of $r = 2$ and $r = 0.8$, and use the norm $\ell_2$, i.e., $p = 2$.

A synthesized sample that achieves a low value for Eq. (3.1) must consist of patches close to the ones in the example, thus capturing its visual characteristics.

### 3.1.2 Optimization Scheme

Eq. (3.1) is not convex, however, the goal for performing texture synthesis is to find local minimizers for which the local minimum is small enough for the synthesized texture to reproduce the visual characteristics of the example.

For $r = 2$ and $p = 2$, the minimization strategy employed by Kwatra et al. [6] is an iterative alternate minimization with respect to $u$ and $\sigma$ separately. Starting from a random initialization, the problem in Eq. (3.1) is solved in a two steps iterative process.

- **Projection step.** Holding $u$ fixed, we have:

$$\arg\min_{\sigma} \quad \sum_{x \in \Omega^{\downarrow}} \|u \circ \mathcal{P}(x) - u_0 \circ \mathcal{P} \circ \sigma(x)\|_2^2. \tag{3.2}$$

  The solution $\sigma$ corresponds to the nearest neighbor projection of each patch in the image being synthesized $u$ to the set of patches in the example $u_0$. This can be parallelized and solved by a linear search, computing the distance to each patch and comparing them to find the closest. In order to speed up the search, in [6] they build a searching tree structure.

- **Aggregation step.** Next, in the patch aggregation step, the mapping between patches $\sigma$ is fixed. The resulting minimization problem is convex with respect to the image $u$ and can be solved analytically as:

$$\arg\min_{u} \quad \sum_{x \in \Omega^{\downarrow}} \|u \circ \mathcal{P}(x) - u_0 \circ \mathcal{P} \circ \sigma(x)\|_2^2. \tag{3.3}$$

This is a separable continuous convex problem that can be solved for each pixel independently as follows. Using $u \circ \mathcal{P}(x) = \{u(x+t),\ t \in \mathcal{N}\}$, we can rewrite Eq. (3.3)

$$\min_{u} \quad \sum_{x \in \Omega^{\downarrow}} \sum_{t \in \mathcal{N}} \left(u(x+t) - u_0(\sigma(x)+t)\right)^2 . \tag{3.4}$$

For a given pixel $x \in \Omega^{\downarrow}$ on the overlying grid, the overlapping pixels come from the patches centered at its neighbors given by $\{x + t', t' \in \mathcal{N}^{\downarrow}\}$ with:

$$\mathcal{N}^{\downarrow} = \{-\tfrac{w}{2}, -\tfrac{w}{4}, 0, \tfrac{w}{4}\}^2.$$

Otherwise, a pixel that is not on the grid can be defined by $x' = x + k \in \Omega/\Omega^{\downarrow}$ with $k = [k_1, k_2]$ and $k_i = x'_i - \frac{w}{4}\left\lfloor \frac{x'_i}{w/4} \right\rfloor$ for $i \in \{1, 2\}$. The values of $k$ depending on the stride between two pixels on the grid. The patches contributing to the value at $x'$ are centered at the neighbors given by $\{x + t' + k,\ t' \in \mathcal{N}^{\downarrow}\}$. We can rewrite Eq. (3.4) as:

$$\min_{u} \quad \sum_{x \in \Omega^{\downarrow}} \sum_{k \in \{0,\ldots,\frac{w}{4}-1\}} \sum_{t' \in \mathcal{N}^{\downarrow}} \left(u(x+t'+k) - u_0(\sigma(x)+t'+k)\right)^2 .$$

Because $\forall x \in \Omega^{\downarrow}$ and $\forall k \in \{0, ..., \frac{w}{4} - 1\}$ there is only one $x' = x + k \in \Omega$, we can write

$$\min_{u} \quad \sum_{x' \in \Omega} \sum_{t' \in \mathcal{N}^{\downarrow}} \left(u(x'+t') - u_0(\sigma(x'-k)+t'+k)\right)^2 .$$

There are $\frac{w^2}{(w/4)^2} = 16$ combinations of $x'+t'$ that result in the same value $x''$. Regrouping those terms, we can solve for each $x''$ independently, for all $x'' = x + k + t' \in \Omega$:

$$u(x'') = \underset{u(x'')}{\operatorname{argmin}} \sum_{t' \in \mathcal{N}^{\downarrow}} \left(u(x'') - u_0(\sigma(x''-t'-k)+t'+k)\right)^2 ,$$

the value of the each pixel is thus given by:

$$u(x'') = \frac{1}{|\mathcal{N}^{\downarrow}|} \sum_{t' \in \mathcal{N}^{\downarrow}} u_0(\sigma(x''-t'-k)+t'+k). \tag{3.5}$$

Eq. (3.5) shows that the aggregation step boils down to, for each pixel, averaging the values of the set of *candidate* pixels, i.e., the pixels that overlap at that pixel location.

### 3.1.3 Robust Energy

Kwatra et al. [6] explain that the simple average between overlapping pixels in Eq. (3.5), resulting of using $r = 2$ in Eq. (3.1), produces blurry results. Instead, they suggest to use $r < 2$ and they specifically set $r = 0.8$. This setting for Eq. (3.1) requires a different optimization approach. Kwatra et al. [6] use an *Iterative Reweighted Least Squares (IRLS)* algorithm [76]. At each iteration the following weighted least squares problem needs to be solved:

$$\min_{u} \min_{\sigma} \quad \sum_{x \in \Omega^{\downarrow}} \omega(x, \sigma(x)) \quad \|u \circ \mathcal{P}(x) - u_0 \circ \mathcal{P} \circ \sigma(x)\|_2^2, \tag{3.6}$$

with $\omega(x, \sigma(x)) = \|u \circ \mathcal{P}(x) - u_0 \circ \mathcal{P} \circ \sigma(x)\|_2^{r-2}$. The IRLS is incorporated in the patch aggregation step, where for a fixed $\sigma$ the weights $\omega(x, \sigma(x))$ are computed and then the resulting minimization problem is convex and separable with respect to the image $u$. Similar to Eq. (3.5), each pixel value $u(x'')$ is given by:

$$\underset{u(x'')}{\operatorname{argmin}} \sum_{t' \in \mathcal{N}^{\downarrow}} \omega(x'' - t' - k, \sigma(x'' - t' - k)) \ \left(u(x'') - u_0(\sigma(x'' - t' - k) + t' + k)\right)^2 .$$

And results in the weighted average:

$$u(x'') = \frac{\sum_{t' \in \mathcal{N}^{\downarrow}} \omega(x'' - t' - k, \sigma(x'' - t' - k)) \ u_0(\sigma(x'' - t' - k) + t' + k)}{\sum_{t' \in \mathcal{N}^{\downarrow}} \omega(x'' - t' - k, \sigma(x'' - t' - k))}, \tag{3.7}$$

for all $x'' = x + k + t' \in \Omega$ with $x \in \Omega^{\downarrow}$, $k \in \{1, \ldots, \frac{w}{4} - 1\}$ and $t' \in \mathcal{N}^{\downarrow}$.

### 3.1.4 Multi-scale Scheme

The process of synthesizing a texture is accomplished using multiple scales from coarse to fine where the up-sampled output at each scale serves as initialization for the next one. Figure 3.1 illustrates this process. The example at each scale is created using the Gaussian pyramid $\{u_s\}_{s=0}^{S-1}$ of the example image, which is composed of $S$ images $u_s$ computed by filtering $u_0$ with a Gaussian kernel and sub-sampled with a stepsize of $2^s$. The multi-scale system is advantageous because at lower resolutions it is easier to capture long distance information and subsequently the high resolution synthesis is carried out with a convenient initialization. Besides the multi-scale approach, Kwatra et al. [6] propose to use multiple sizes of patches at each scale. The width of patches used in these sub-level optimizations are 32, 16, and 8 pixels, in this specific order. If one width of patches does not fit in that scale of the input, that sub-level is skipped.

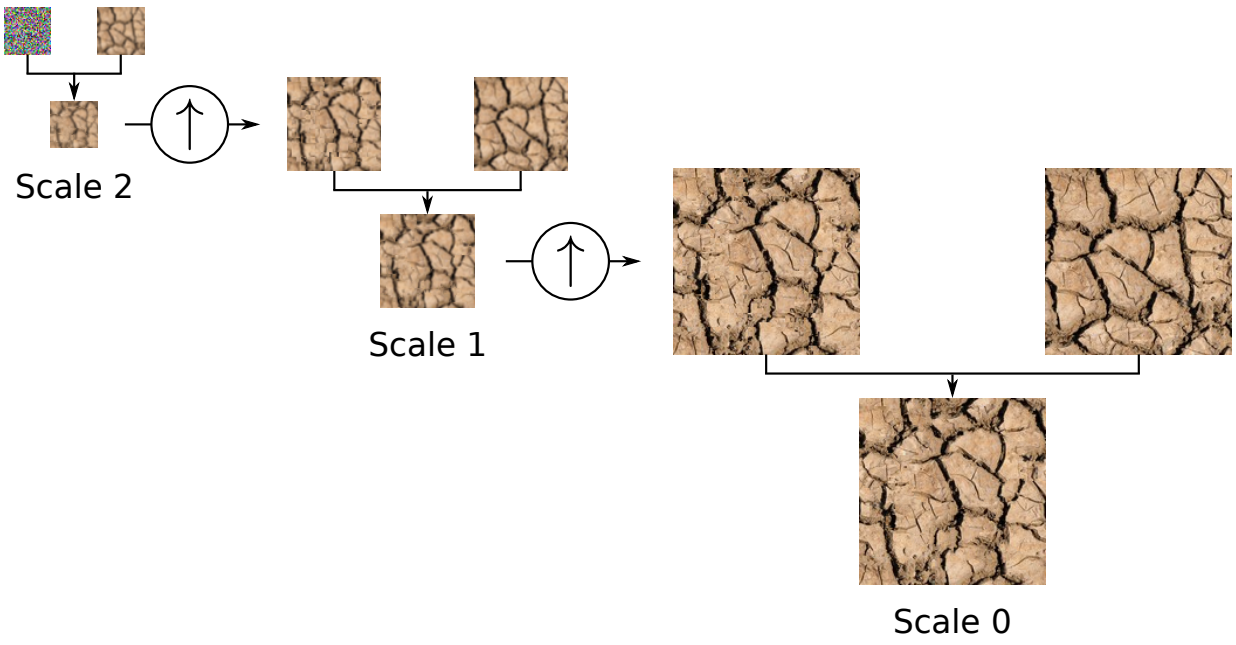



Figure 3.1 **Multi-scale synthesis scheme.** At each scale the top two images are the initialization and the example respectively and the bottom image is the synthesized sample. This sample is upsampled (↑) and used as the initialization at the following scale.

### 3.1.5 Quality of the Results

In Figure 2.6 in the previous chapter we showed how the method of Kwatra et al. [6] is capable of producing good quality results. Thanks to its patch-based strategy it reproduces well the local structures and because of the global formulation it is able to produce long distance coherent patterns.

One disadvantage of this method is the lack of guarantee of obtaining a successful result. Due to the nearest neighbor projection this method can generate flat *garbage* regions [38] and introduce extreme repetition of pieces of the texture. Figure 3.2 illustrates the nearest neighbor affectation between two point clouds in 2D. Here, most of the red points are affected to a small set of blue points. A similar result can happen at higher dimensions, as is the case of the space of patches. Using the nearest-neighbor model can result in using a small set of patches to build the synthesized sample, resulting in enlarged constant regions and high repetition of a given pattern. In Figure 3.3 we show some results of Kwatra et al. [6] where we highlight these artifacts.

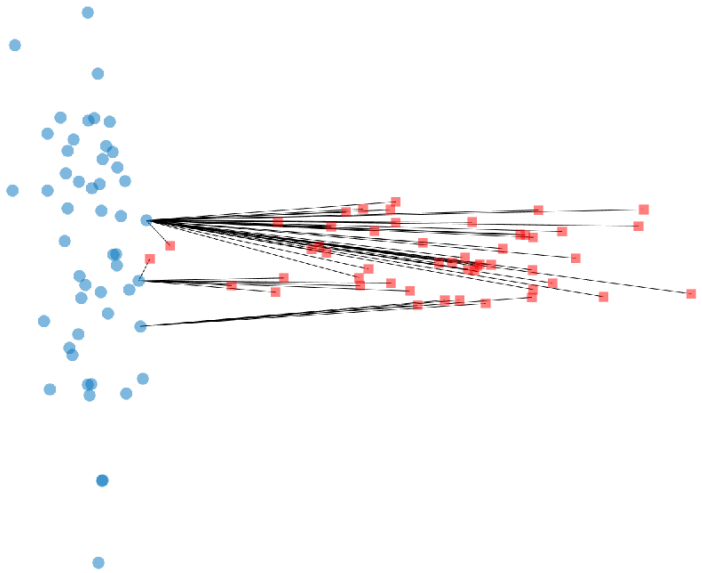


Figure 3.2 **Illustration of a nearest-neighbor (NN) affectation between points in 2D.** The two sets are differentiated by color and shape. The black lines indicate the affectation between pairs of points. Even at higher dimensions, using the nearest-neighbor model can result in an extremely uneven affectation between two sets of points.

## 3.2 Uniformity Mechanisms for Texture Optimization

Several methods that build upon Texture Optimization point out the potential loss of content inherent to the nearest neighbor formulation and propose different *uniformity mechanisms* that encourage an even utilization of the patches from the example. These methods are more robust to failure and produce some of the best quality results among the patch-based literature. Kopf et al. [17] propose a *Histogram Matching* that alters the patch aggregation step to discourage the over-utilization of a given patch. With the same goal, Kaspar et al. [45] complement Histogram Matching with a *Spatial Uniformity Distance* that empirically modifies the patch distance computation before the patch projection step. The bidirectional model of Wei et al. [77], explicitly incorporates the uniformity mechanism in the model. If used for texture synthesis it improves the quality of the samples [45]. In this section we detail these three mechanisms as they relate to our hypothesis that a match between the patch's statistics results in a better quality synthesis.

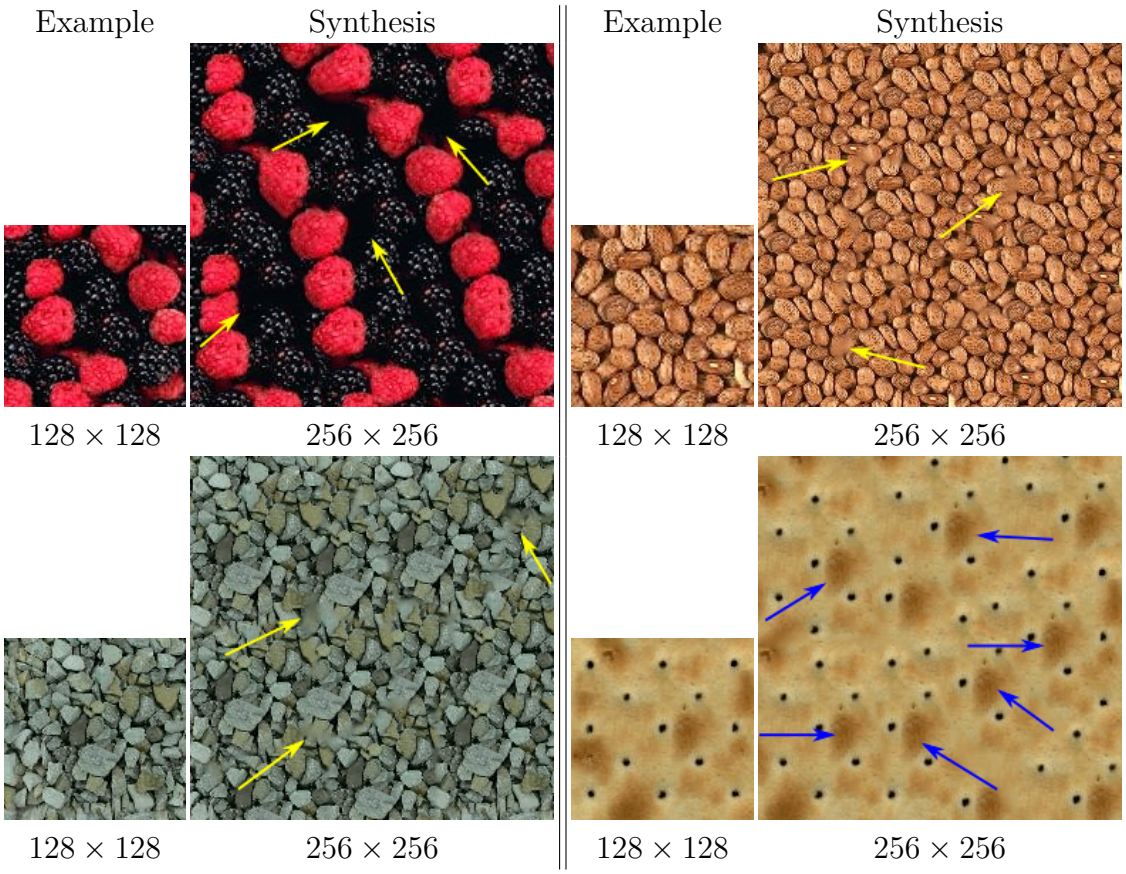



Figure 3.3 **Examples of artifacts introduced by the method of Kwatra et al. [6].** We use yellow arrows to point at blurry/constant regions and orange arrows for the repeated identical pieces of texture. Images from the website of [6].

### 3.2.1 Histogram Matching

The Histogram Matching proposed by Kopf et al. [17] discourages the utilization of pixel's colors surpassing their occurrence in the example texture. The main goal is to push the synthesis towards using colors in the same proportion as the example. This mechanism alters the aggregation step. It modifies the weights $\omega$ in the weighted average Eq. (3.7) depending on the current histogram, therefore it requires to update each pixel sequentially and in a random order. Let us call $H^+_{u,c}$ and $H^+_{u_0,c}$ the updated histogram (up to the last synthesized pixel) for color $c$ of the sample $u$ and the example $u_0$ respectively. Then $H^+_{u,c}(u(x))$ denotes the value of the histogram at the bin to which the color $u(x)$ belongs. To simplify the notation we use $y = t' + k$. The re-weighting is given by:

$$\omega'(x'' - y, \sigma(x'' - y)) = \frac{\omega(x'' - y, \sigma(x'' - y))}{1 + \sum_{c\in\{R,G,B\}} \max[0, H^+_{u,c}(u_0(\sigma(x'' - y) + y)) - H^+_{u_0,c}(u_0(\sigma(x'' - y) + y))]}. \quad (3.8)$$

Histogram Matching modifies the contribution of each candidate pixel participating in the weighted average. It attenuates the values that make the count of its color in the new

sample exceed the proportion in the example. Figure 3.4 illustrates how to compute the new weight of a given candidate pixel. In practice this histogram monitoring technique helps to better employ the full color richness of the example, however, it requires a sequential pixel optimization, making it a bottleneck for the synthesis process.

### 3.2.2 Bidirectional Texture Energy

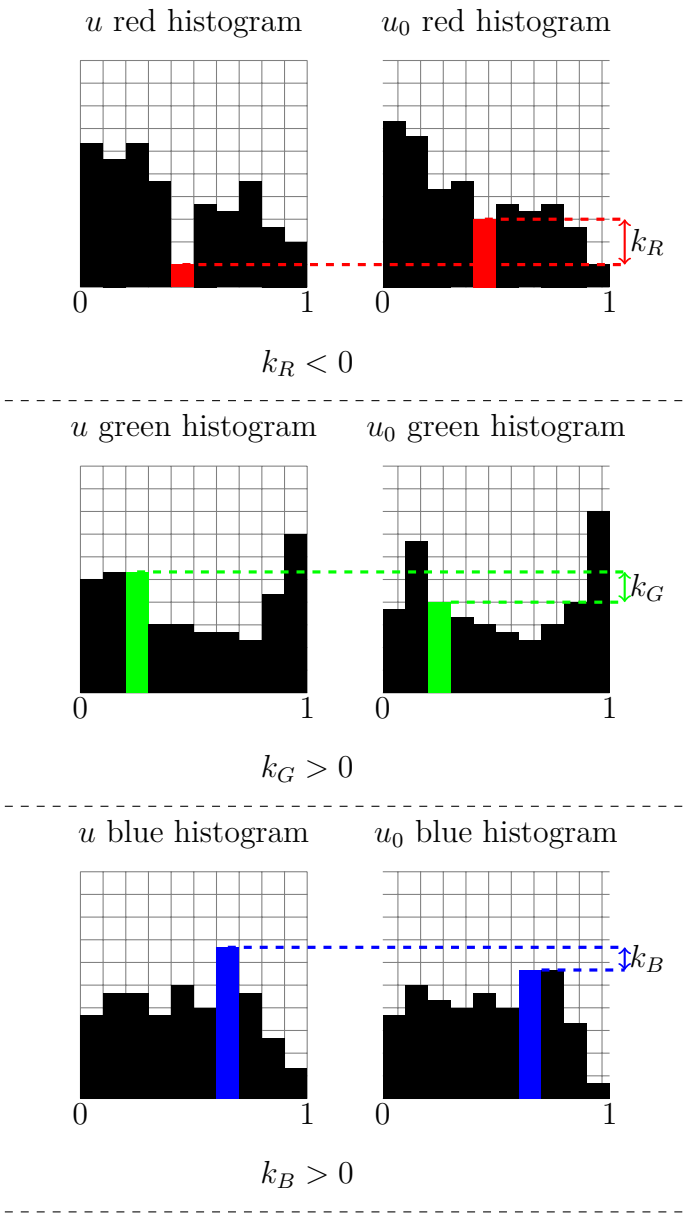
Simakov et al. [78] and Wei et al. [77] propose the addition of a new *inverse* term to the dissimilarity measure (Eq. (3.1)) whose function is to encourage the preservation of the whole information of the input. This formulation can be used for texture synthesis [45], although it is not its original application. The proposed term accounts for the distance between each patch in the example to their nearest neighbor in the synthesized sample. Intuitively, in order to get a low value of this energy term, the sample must have a patch similar to every patch in the example. Integrating the new term in the framework of Eq. (3.1) would formally result in:

$$\min_{u} \quad \alpha E(u, u_0) + (1 - \alpha) E(u_0, u), \tag{3.9}$$

with:

$$E(u, u_0) = \frac{1}{|\Omega_1^{\downarrow}|} \sum_{x \in \Omega_1^{\downarrow}} \min_{\sigma} \|u \circ \mathcal{P}(x) - u_0 \circ \mathcal{P} \circ \sigma(x)\|_p^r,$$

where $\sigma : \Omega_1^{\downarrow} \to \Omega_2^{\downarrow}$ is a mapping between the indexes of the patches in the first and second images, $p = 2$ and $r = 2$. The optimization scheme using this model involves two nearest neighbor searches, each corresponding to one term in Eq. (3.9). The aggregation step can also be solved analytically. When applied to texture synthesis this model slightly improves the results of the basic texture optimization model as shown by Kaspar et al. [45]. Figure 3.5 illustrates the bidirectional nearest neighbor affectation for the same two point clouds of Figure 3.2. Again, in the direct affectation (black lines) most of the red points are affected to a small set of blue points. Additionally, in the inverse affectation (blue lines) most of the blue points are affected to a small set of red points. For texture synthesis the inverse affectation attempts to force using all the patches from the example. There is, however, no guarantee of success. Because the inverse term involves also a NN affectation, it can also collapse into a small set of patches in the synthesized texture. Thus, only a small part of this sample would be impacted by the inverse term.




Only the positive differences are used in the reweighting:

$$\omega'(x'' - y, \sigma(x'' - y)) = \frac{\omega(x'' - y, \sigma(x'' - y))}{1 + k_G + k_B}.$$

Figure 3.4 **Empirical reweighting for the histogram matching used by Kopf et al. [17].**

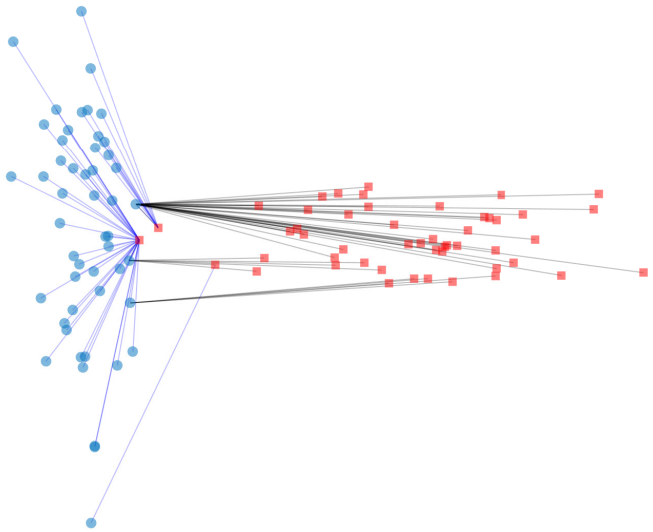


Figure 3.5 **Illustration of a bidirectional nearest neighbor (NN) affectation between two point clouds in 2D.** The two sets are differentiated by color and shape. The black and blue lines indicate the direct and inverse NN affectations between pairs of points.

### 3.2.3 Spatial Uniformity Distance

The Spatial Uniformity Distance [45] promotes the use of each pixel in the same proportion as the example. The idea is to simply penalize the pixels that are used in excess by adding a term to the patch's distance computation.

This requires to count the number of times a pixel from the example is used in the synthesized sample. For a fixed $\sigma$ (of the precedent iteration), the counter is $\mathcal{U}_\sigma : \Omega_0 \to \mathbb{N}^+$ with

$$\mathcal{U}_\sigma(y) = |y \in \mathcal{P} \circ \sigma(x), x \in \Omega\}|. \tag{3.10}$$

For each $x$ index in the grid of the synthesized sample, we add one value to the counter $\mathcal{U}(y)$ if $y$ is in the patch associated with $x$ via the mapping $\sigma$.

Given that the ideal proportion of pixels is:

$$\Pi_{best} = \frac{|\Omega|}{|\Omega_0|} w^2, \tag{3.11}$$

the counter is used to alter the distance computation before the patch projection step. The Spatial Uniformity Distance optimization with respect to the mapping $\sigma$ is given by:

$$\underset{\sigma}{\operatorname{argmin}} \sum_{x \in \Omega^{\downarrow}} \Big( \| u \circ \mathcal{P}(x) - u_0 \circ \mathcal{P} \circ \sigma(x) \| \Big) + \lambda_{occ} \frac{\Pi_{patch}(\sigma(x))}{\Pi_{best}}, \tag{3.12}$$

where

$$\Pi_{patch}(\sigma(x)) = \frac{1}{w^2} \sum_{y \in \mathcal{P} \circ \sigma(x)} \mathcal{U}_{\sigma}(y)$$

is the average occurrence of the pixels in the patch of the example associated with the patch in the sample centered at $x$. This uniformity mechanism improves the quality of the synthesis but it also forbids parallel computation because of the counter keeping of the pixel usage count.

## 3.3 Optimal Patch Assignment for Statistically Constrained Texture Synthesis

In this section we present our proposed model for example-based texture synthesis. It builds upon Texture Optimization [43] as we formulate the problem as the minimization of a patch-based dissimilarity measure. Similar to the methods addressed in Section 3.2 we seek to obtain high quality samples by the means of an even utilization of the patches in the example. However, instead of using an empirical mechanism, we propose an explicit model based on the *Optimal Assignment Problem*, where all of the patches extracted from the example must be used for synthesizing a new sample.

We claim that matching the patch distribution of the example prevents the formation of blurry or constant results and allows for repetition only in the same amount present in the example. Here we detail the proposed model along with the optimization strategy, and study the quality of the results.

### 3.3.1 Optimal Assignment Model

The proposed model follows the generic Texture Optimization [6] framework detailed in Section 3.1 and it is closely related to several approaches from the literature such as [17,44,45]. The core of the model is the Optimal Assignment between the sets of patches in the example and synthesized textures. Here we follow the same notation as in Section 3.1. In order to simplify the presentation, without loss of generality, we consider the pixel grids $\Omega_0$ and $\Omega$ to have the same size (i.e., $N = |\Omega| = |\Omega_0|$). In Section 3.3.3 we will detail our implementation for arbitrary grid sizes. The patches are extracted on sub-grids $\Omega_0^{\downarrow}$ and $\Omega^{\downarrow}$ of $\Omega_0$ and $\Omega$,

having a step size $\frac{w}{4}$ and size $N^{\downarrow} = |\Omega_0^{\downarrow}| = |\Omega^{\downarrow}|$. We use a symmetric boundary condition for patches of $u_0$ and periodic boundary condition for patches of $u$.

Our texture synthesis approach is driven by the following optimization problem which aims at minimizing the discrepancy between the patches from the synthesized texture and patches from the example image,

$$\min_{u} \min_{\sigma \in \Sigma_{N^{\downarrow}}} \quad \sum_{x \in \Omega^{\downarrow}} \|u \circ \mathcal{P}(x) - u_0 \circ \mathcal{P} \circ \sigma(x)\|_p^r, \tag{3.13}$$

where $\sigma : \Omega^{\downarrow} \to \Omega_0^{\downarrow}$ again is a mapping between of the indexes of the patches but it is constrained to be an element of the set $\Sigma_{N^{\downarrow}}$. The most important aspect of this model is the definition of the set $\Sigma_{N^{\downarrow}}$ as the *set of permutations* of $N^{\downarrow}$ elements.

The motivation is to synthesize a new texture that has the same distribution of patches as the example in order to preserve all its visual characteristics. We also consider $r = 1$ and $p = 1, 2$ that we use to specify the $\ell_{1,2}$ norm of patches, that is the sum of Euclidean norm of color coordinates:

$$\|u \circ \mathcal{P}(x)\|_{1,2} = \sum_{y \in \mathcal{P}(x)} \|u(y)\|_2.$$

These choices offer several advantages, resulting mainly in a separable convex optimization problem when minimizing with respect to $u$. Recall that the model of Kwatra et al. [6] uses $r = 0.8$, an Euclidean norm weighted with a Gaussian falloff, and a mapping $\sigma$ that is not constrained to be a permutation, which results in a nearest neighbor matching.

### 3.3.2 Optimization Scheme

The motivation for Problem (3.13) is to find an image $u$ which is pixel-wise different from the example image $u_0$ while having the same patch distribution. Yet, Problem (3.13) has trivial solutions that are not relevant. For instance, using periodic boundary conditions, any circular shifting of the input image is a global minimizer. Still, as usually done for variational texture synthesis [1, 2, 6], we minimize the non-convex functional of Problem (3.13) by alternate minimization with respect to $u$ and $\sigma$ separately starting from a random image. This relies on the assumption that local minimizers of the functional provide the desired result, which is verified in practice.

More precisely, Problem (3.13) is convex with respect to the image $u$ for a given $\sigma$, i.e., when the assignment $\sigma$ between patches has been fixed: this is the *patch aggregation* step. In addition, the problem of finding an optimal assignment $\sigma$ for a given synthesized image $u$ is a solution of a relaxed convex problem: this is the *patch projection* step. These two steps are

sequentially used in a multi-scale scheme.

- **Patch aggregation** When the assignment $\sigma$ between patches on the sub-grids is fixed, optimizing the synthesized image $u$ boils down to find

$$\underset{u}{\operatorname{argmin}} \quad \sum_{x \in \Omega^{\downarrow}} \|u \circ \mathcal{P}(x) - u_0 \circ \mathcal{P} \circ \sigma(x)\|_{1,2}. \tag{3.14}$$

Here we show that, similar to Eq. (3.5) this is a separable convex optimization problem that can be solved independently for each pixel in $u$. Each pixel is only covered by a small number of overlapping patches that we have to determine to solve this problem efficiently in parallel.

We recall that,

$$\mathcal{P}(x) = \left\{x + t,\ t \in \{-\tfrac{w}{2}, \dots, \tfrac{w}{2} - 1\}^2\right\},$$

then, using $u \circ \mathcal{P}(x) = \{u(x+t),\ t \in \mathcal{N}\}$ and $\mathcal{N} = \{-\frac{w}{2}, \dots, \frac{w}{2} - 1\}^2$ we develop Eq. (3.14):

$$u \in \underset{u}{\operatorname{argmin}} \quad \sum_{x \in \Omega^{\downarrow}} \sum_{t \in \mathcal{N}} \|u(x+t) - u_0(\sigma(x) + t)\|_2. \tag{3.15}$$

For a given $x \in \Omega^{\downarrow}$, the candidate pixels come from its neighbors given by $\{x + t', t' \in \mathcal{N}^{\downarrow}\}$ with:

$$\mathcal{N}^{\downarrow} = \{-\tfrac{w}{2}, -\tfrac{w}{4}, 0, \tfrac{w}{4}\}^2.$$

Otherwise, we define a pixel that is not on the grid as $x' = x + k \in \Omega / \Omega^{\downarrow}$ with $k = [k_1, k_2]$ and $k_i = x'_i - \frac{w}{4} \left\lfloor \frac{x'_i}{w/4} \right\rfloor$ for $i \in \{1, 2\}$. The patches contributing to the value at $x'$ are centered at the neighbors given by $\{x + t' + k,\ t' \in \mathcal{N}^{\downarrow}\}$. Rewriting Eq. (3.15), we want to solve:

$$\min_{u} \quad \sum_{x \in \Omega^{\downarrow}} \sum_{k \in \{0, \dots, \frac{w}{4} - 1\}} \sum_{t' \in \mathcal{N}^{\downarrow}} \|u(x + t' + k) - u_0(\sigma(x) + t' + k)\|_2. \tag{3.16}$$

Because $\forall x \in \Omega^{\downarrow}$ and $\forall k \in \{0, \dots, \frac{w}{4} - 1\}$ there is only one $x' = x + k \in \Omega$, we can write:

$$\min_{u} \quad \sum_{x' \in \Omega} \sum_{t' \in \mathcal{N}^{\downarrow}} \|u(x' + t') - u_0(\sigma(x' - k) + t' + k)\|_2. \tag{3.17}$$

Regrouping the $\frac{w^2}{(w/4)^2} = 16$ combinations of $x' + t'$ that result in the same value $x''$ we

can solve for each $x''$ independently:

$$u(x'') \in \operatorname*{argmin}_{u(x'')} \sum_{t' \in \mathcal{N}^{\downarrow}} \|u(x'') - u_0(\sigma(x'' - t' - k) + t' + k)\|_2, \tag{3.18}$$

which corresponds to computing the color geometric median of the pixel values overlapping at $x''$. We solve the $N$ problems Eq. (3.18) in parallel using a Douglas-Rachford splitting algorithm.

**Geometric median patch aggregation** To simplify the notation, let us restate Eq. (3.18) as follows:

$$\min_p \sum_{t=1}^{K} \|p - q_t\|_2, \tag{3.19}$$

where $t$ enumerates the $K = \frac{w^2}{(w/4)^2} = 16$ candidate values that contribute to the value of pixel $p = u(z) \in \mathbb{R}^n$, $n = 3$ for color pixels, and with $q_t = (u_0(\sigma(z - t) + t))$.

Eq. (3.19) can be formulated as a consensus problem:

$$\begin{aligned} \min \quad & \sum_{t=1}^{K} f_t(p_t) \\ \text{s. t.} \quad & p = [p_1, \ldots, p_K] \in C = \{[p_1, \ldots, p_K] \in \mathbb{R}^{nK}, p_1 = p_2 = \cdots = p_K\}, \end{aligned} \tag{3.20}$$

with $f_t(p_t) = \|p_t - q_t\|_2$ and with now $p$ being the concatenation of the $K$ pixels $p_t \in \mathbb{R}^n$. The linear constraint $p \in C$ requires that $p_i = p_j$, for all $i, j \in \{1, \ldots, K\}$.

The Douglas-Rachford algorithm [79] is used to solve problems of the form:

$$\min_p \; F(p) + G(p).$$

At each iteration $\tau + 1$ the Douglas-Rachford algorithm consists on the following updates:

$$\begin{aligned} p^{(\tau+1)} &= \operatorname{prox}_{\gamma F}(y^{(\tau)}) \\ y^{(\tau+1)} &= y^{(\tau)} + \lambda \left( \operatorname{prox}_{\gamma G} \left( 2p^{(\tau+1)} - y^{(\tau)} \right) - p^{(\tau+1)} \right), \end{aligned} \tag{DR}$$

where $\operatorname{prox}_F$ is the proximal operator of function $F$ and $\lambda \in ]0, 2[$, and $\gamma \in ]0, \infty[$ are parameters of the optimization algorithm. For the consensus Problem (3.20), we have

that $G(p) = \iota_C(p)$ the characteristic function of the set $C$ given by:

$$\iota_C(p) = \begin{cases} 0 & \text{if } p \in C \\ +\infty & \text{if } p \notin C, \end{cases}$$

and $\text{prox}_{\gamma\iota_C}(p) = P_C(p)$, the orthogonal projection of a point $p = [p_1, \ldots, p_K]$ into $C$, given by $P_C(p) = [\bar{p}, \ldots, \bar{p}]$ with

$$\bar{p} = \frac{1}{K}\sum_{t=1}^{K} p_t.$$

Replacing in (DR) we have:

$$\begin{aligned} p^{(\tau+1)} &= \text{prox}_{\gamma F}(y^{(\tau)}) \\ y^{(\tau+1)} &= y^{(\tau)} + \lambda\left(2\bar{p}^{(\tau+1)} - \bar{y}^{(\tau)} - p^{(\tau+1)}\right), \end{aligned} \tag{3.21}$$

using $\lambda = 1$,

$$\bar{y}^{(\tau)} = \bar{y}^{(\tau-1)} + 2\bar{p}^{(\tau)} - \bar{y}^{(\tau-1)} - \bar{p}^{(\tau)} = \bar{p}^{(\tau)}, \tag{3.22}$$

thus, we obtain the update

$$y^{(\tau+1)} = y^{(\tau)} + \left(\bar{p}^{(\tau+1)} - p^{(\tau+1)}\right) + \left(\bar{p}^{(\tau+1)} - \bar{p}^{(\tau)}\right). \tag{3.23}$$

Regarding $F$, we have the separable function $F(p) = \sum_{t=1}^{K} f_t(p_t)$. Splitting the $K$ problems (DR) becomes:

$$\begin{aligned} p_t^{(\tau+1)} &= \text{prox}_{f_t}(y_t^{(\tau)}) \\ y_t^{(\tau+1)} &= y_t^{(\tau)} + (\bar{p}^{(\tau+1)} - p_t^{(\tau+1)}) + (\bar{p}^{(\tau+1)} - \bar{p}^{(\tau)}). \end{aligned} \tag{DR-C}$$

For the geometric median, the functions $f_t$ are given by

$$f_t(p) = \|p - q_t\|_2.$$

Since

$$\text{prox}_{\gamma\|\cdot\|}(p) = p - P_{\{q, \|q\|_* \leq \gamma\}}(p),$$

for the Euclidean norm $\|\cdot\| = \|\cdot\|_2$, $\|\cdot\|_* = \|\cdot\|_2$, thus we have:

$$\text{prox}_{\gamma}(p) = p - P_{\{q, \|q\|_2 \leq \gamma\}}(p),$$

with the Euclidean projection $P_{\{q,\|q\|_2\leq\gamma\}}$ given by:

$$P_{\{q,\|q\|_2\leq\gamma\}}(p) = \begin{cases} p & \text{if } \|p\|_2 < \gamma \\ \gamma\frac{p}{\|p\|_2} & \text{otherwise,} \end{cases} \tag{3.24}$$

which can be rewritten as:

$$P_{\{q,\|q\|_2\leq\gamma\}}(p) = \frac{p}{\max(1, \frac{\|p\|_2}{\gamma})}.$$

Finally, we set $\gamma = 1$ so that the first step in (DR-C) $\text{prox}_{f_t}(y_t)$ is given by:

$$\text{prox}_{f_t}(y_t) = y_t - \frac{y_t - q_t}{\max(1, \|y_t - q_t\|_2)}. \tag{3.25}$$

Therefore, we solve our consensus problem Eq. (3.19) iterating $\forall t$:

$$\begin{aligned} p_t^{(\tau+1)} &= y_t^{(\tau)} - \frac{y_t^{(\tau)} - q_t}{\max(1, \|y_t^{(\tau)} - q_t\|_2)} \\ \bar{p}^{(\tau+1)} &= \frac{1}{K}\sum_{t=1}^{K} p_t^{(\tau+1)} \\ y_t^{(\tau+1)} &= y_t^{(\tau)} + (\bar{p}^{(\tau+1)} - p_t^{(\tau+1)}) + (\bar{p}^{(\tau+1)} - \bar{p}^{(\tau)}). \end{aligned} \tag{3.26}$$

- **Optimal patch assignment** The output image $u$ being fixed, an optimal assignment $\sigma$ is solution of :

$$\underset{\sigma\in\Sigma_{N\downarrow}}{\text{argmin}} \quad \sum_{x\in\Omega\downarrow} \|u\circ\mathcal{P}(x) - u_0\circ\mathcal{P}\circ\sigma(x)\|_{1,2} \ . \tag{3.27}$$

Similar to the nearest neighbor projection, the optimal assignment creates a mapping between the two sets of patches. The permutation constraint, however, forces this correspondence to be one-to-one. Figure 3.6 illustrates the optimal assignment between the same two point clouds of Figure 3.2. The optimal assignment affectation by definition pairs all the elements in both sets. This problem can be recast as a *Linear Sum Assignment Problem* (LSAP) and solved in many different ways [80]. In practice, we use the Hungarian algorithm which is very fast for small assignment problems [81]. For synthesizing larger images or to reduce the computation time, alternative methods should be considered such as a parallel implementation of the auction algorithm [82], as we will discuss in Section 3.5.1, or the approximate assignment approach using the

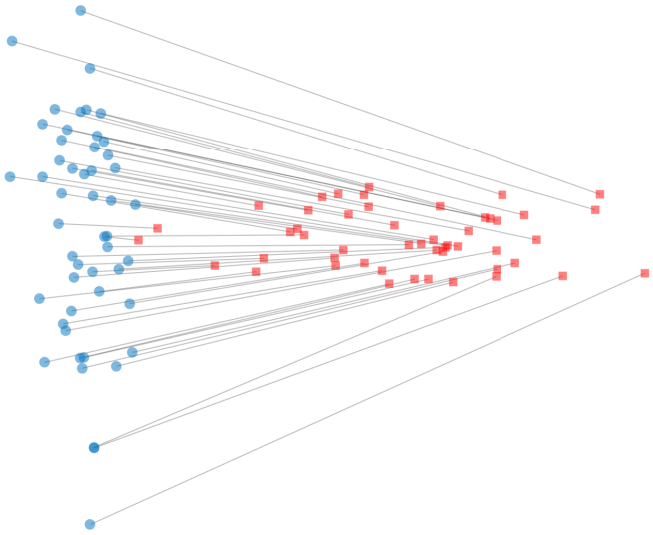


Figure 3.6 **Illustration of an optimal assignment affectation between two point clouds in 2D.** The two sets are differentiated by color and shape. The black lines indicate the affectation between pairs of points. By definition the optimal assignment makes an even affectation that pairs all of the elements in both sets. In contrast to a simple permutation, the optimal assignment is the one that minimizes the overall distance.

Sliced-Wasserstein distance [30].

### 3.3.3 Implementation Details

Here we discuss some more implementation details and summarize the proposed texture synthesis method in an algorithm.

**Multi-scale scheme and initialization** In order to capture large scale correlations between the patterns in the example, we use a multi-scale coarse-to-fine synthesis [2, 43, 68]. Similar to the strategy detailed in Section 3.1, we perform multi-scale synthesis by computing the Gaussian pyramid $\{u_s\}_{s=0}^{S-1}$ of the example image, which is composed of $S$ images $u_s$ computed by filtering $u_0$ with a Gaussian kernel with standard deviation 0.8 and sub-sampled with a step size of $2^s$. We initialize the synthesized sample at the coarsest scale using a Gaussian white noise image, as done in several varational texture synthesis methods [1, 29, 31].

For subsequent scales, $u$ is first upsampled by a factor 2 using bilinear interpolation. The

resulting algorithm is described in Algorithm 1. In all experiments, the patch width is fixed to $w = 8$, the number of scales is $S = 3$, and the number of iterations $I_s$ at scale $s$ is $\{I_s\}_{s=0}^{S-1} = \{10, 50, 50\}$.

**Algorithm 1:** Optimal patch assignment texture synthesis

**Input:** Example texture $u_0$

**Parameters:** Number of scales: $S = 3$, patch width: $w = 8$ px,
Iterations per scale: $\{I_s\}_{s=0}^{S-1} = \{10, 50, 50\}$

**Initialization:** $\{u_s\}_{s=0}^{S-1} \leftarrow$ Gaussian pyramid of $u_0$
$u \leftarrow$ Gaussian white noise image with same size as $u_{S-1}$

**for** $s = S - 1$ *to* $0$ **do**
  **for** $i = 1$ *to* $I_s$ **do**
    $\sigma \leftarrow$ Optimal assignment of patches of $u$ to patches of $u_s$ (Eq. (3.27))
    $u \leftarrow$ Patch aggregation using $\sigma$ (Eq. (3.26))
    **if** $s \neq 0$ **then**
      $u \leftarrow$ Bilinear interpolation of $u$ at scale $s - 1$

**Output:** Synthesized image $u$

**Output size** For simplicity, we have assumed up to this point that the new sample pixel grid $\Omega$ has the same size as the example grid $\Omega_0$, which results in defining $\sigma$ as a permutation of the indexes of patches. However, in practice it is necessary to be able to synthesize textures larger than the example. If $N^{\downarrow} = nN_0^{\downarrow}$ for some integer $n \geq 2$, a simple solution used in the experiments consists of duplicating $n$ times the example patches. If $N^{\downarrow}$ is completely arbitrary, a usual solution is to simply crop a larger synthesized sample. More relevant solutions might be obtained from bootstrapping, i.e., a random sampling of $N^{\downarrow}$ patches from the example distribution, or by considering multiple assignments [81].

### 3.3.4 Experimental Results and Comparison

Figure 3.7 shows the value for the energy in Eq. (3.13) for the synthesis of three different textures. On all cases the energy decreases across iterations. A peak appears at the scale transition but immediately decreases and settles to a lower value than before.

Figure 3.8 shows several synthesis examples generated with the proposed method. Thanks to the patch based formulation, our method is capable of reproducing the local structure of the example. In contrast to nearest neighbor methods, we do not observe enlarged constant or blurry regions in the synthesized samples. This suggests that the optimal assignment constraint helps preventing those flaws. One disadvantage that we observe in Figure 3.8 is that our method seems to produce more distorted patterns, meaning that some patterns are

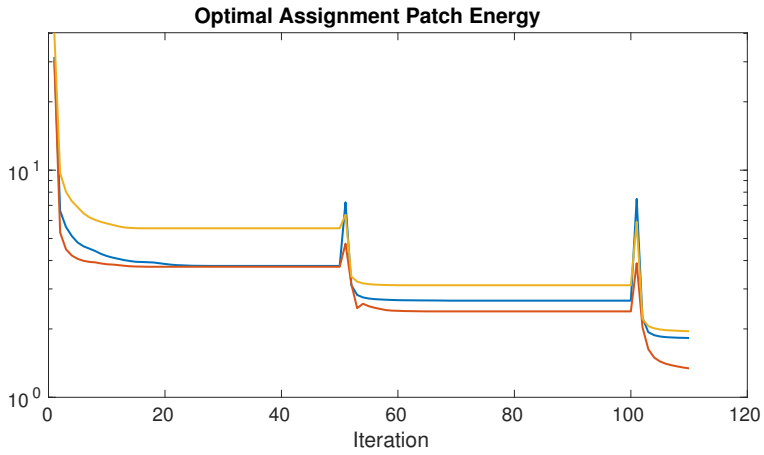



Figure 3.7 **Value of Eq.** (3.13) **for three different textures during the optimization.** We take into account the values at different scales of the synthesis by normalizing by the number of patches. The spikes are due to the initialization at the scale transition. The vertical axis is in logarithmic scale.

combined in a less consistent way. For example, in the top-right synthesis, the generated beans are sometimes enlarged or deformed. This might be due to the hard constraint of using all of the patches from the example. It forces the algorithm to stitch together patches that are not the most coherent.

**Innovation** We include in Figure 3.8 the correspondence maps of the synthesis to illustrate the last optimal assignment $\sigma$ computed during the optimization process. One can observe with the double size output that the strict assignment forces the synthesis to reproduce four times the structures and colors from the input. The correspondence maps demonstrate that our approach tends to synthesize textures that are local verbatim copies of the example, i.e., piece-wise identity. Similarly, as demonstrated by Arias et al. [83], patch methods based in the nearest neighbor (NN) model encourage piece-wise constant and piece-wise identity results. The optimal assignment model, however, seems to produce less large smooth regions than the NN model of Kwatra et al. [6]. In this kind of correspondence maps, innovation is only found in the patterns that are at the border between the copied regions. The fairly large regions in Figure 3.8 mean that the innovation capacity of the model is limited. For larger samples there is a risk of repetition becoming striking.

**Statistical compliance** The patch assignment step generates a patch distribution that is identical to the one of the input. However, the patch aggregation step combines the overlapping pixels in neighboring patches, creating new patches. Therefore the synthesized sample

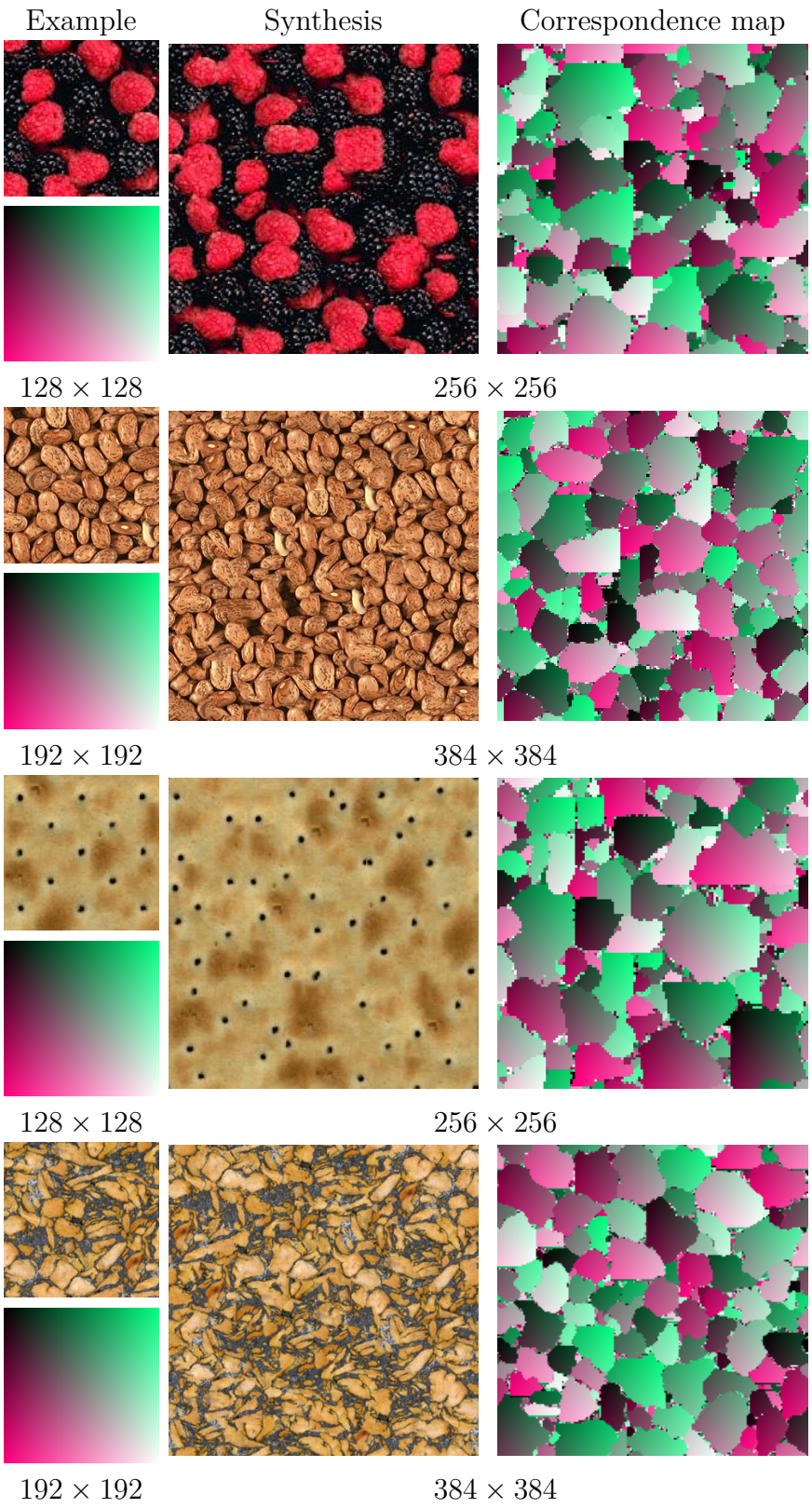



Figure 3.8 **Examples synthesized with the proposed patch optimal assignment method.** The correspondence map highlights the piece-wise identity nature of the results.

patch distribution might not be strictly identical to the one of the example. In order to assert experimentally that these two patch distributions are close, we propose to compare the distributions of first-order statistics between the example and the synthesized sample. In Figure 3.9 we analyze the statistics of the synthesis in the last row of Figure 3.8. We show a comparison of the RGB per-channel color distributions and 1st and 2nd principal components distributions of patches using a PCA analysis on the example's patch distribution. From Figure 3.9 we can conclude that our method generates samples that comply with the example's statistics.

**Impact of the pixel sub-grid regularity** One explanation for the piece-wise identity behavior of the results in Figure 3.8 is that neighboring patches from the example *stitch* perfectly together and thus produce a low value for the dissimilarity measure in Eq. (3.13).

Here, we explore the use of a random subset of patches from the example instead of the regularly spaced grid $\Omega^{\downarrow}$. Figure 3.10 illustrates these two types of *grids* of patches (formally $\Omega^{\downarrow}$). Both grids contain the same number of patches, but the random grid contains some larger *holes* that can disrupt spatial regularity. Figure 3.11 shows a typical synthesis result using our method with a random grid. This sub-sampling random grid is fixed for each scale and maintained constant for all the iterations. The level of randomness in the correspondence map is higher than in the examples in Figure 3.8. This finding might have impact on many patch-based methods. The more stochastic correspondence maps mean that the patterns in the synthesized textures are less of a verbatim copy of the patterns in the examples. We interpret this increase on innovation capacity as a forced behavior. By using patches that are not regularly spaced, it is more difficult to grow large local identity regions, each *hole* in the random sub-grid of the example forces the synthesis to jump to a different region. This innovation, however, comes at a cost. The synthesized sample contains a uniform blur that degrades its visual quality. This might be due to mismatch between overlapping patches.

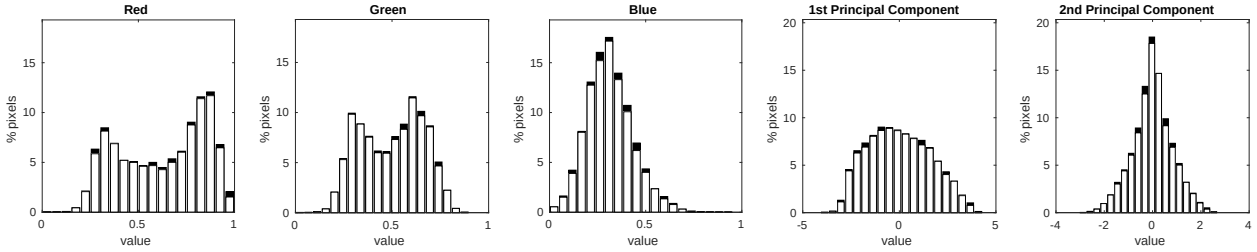



Figure 3.9 **RGB and patch histograms of the synthesized texture in the last row of Figure 3.8.** The absolute differences with the histograms of the example image are displayed in black.

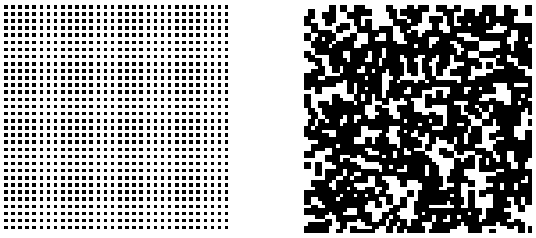


Figure 3.10 **Illustration of the sub-grids of pixels $\Omega^{\downarrow}$ for an image resolution of $64 \times 64$ pixels.** The black dots represent the centers of the patches that are used. Left, regular spaced sampling of patches with stride of two which results in taking one patch out of four. Right, example of a random sampling of patches that takes one out of four patches.

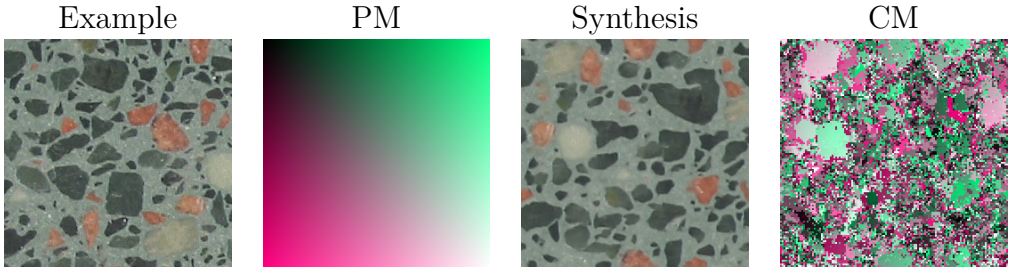



Figure 3.11 **Innovation in a synthesis using a random pixel sub-grid in our patch assignment method.** The noisy correspondence map (CM) shows that the synthesis contains little local copy. All the images have a resolution of $256 \times 256$.

We propose to perform a *hybrid* grid synthesis, where we use a random grid at all the scales but the finest, where we use a regularly spaced grid. This way, the overall layout can be fixed at the early stages of the process and the last scale synthesis can bring back the fine details by using patches that *stitch* better. Figure 3.12 compares the results of synthesis using a regularly spaced grid, a random grid, and the hybrid approach. We observe that the hybrid approach is successful. The hybrid and the random synthesis have an almost identical layout but the hybrid samples do not present any blur. It effectively reduces the size of the local copy regions compared to the all-regular grid synthesis, thus encouraging innovation.

**Comparison with the state of the art** Figure 3.13 shows a comparison of the results obtained using our method (with regular sub-sampling of patches) with the patch-based approaches of Kwatra et al. [6] and Barnes et al. [18] (PatchMatch), and the statistical matching methods of Portilla et al. [1] and Tartavel et al. [2]. For PatchMatch, we use an improved multi-scale algorithm implemented David Tschumperlé for texture synthesis [19]. The original version produce inferior results [45]. The two statistical matching methods successfully preserve the color distributions. However, they often fail to faithfully reproduce

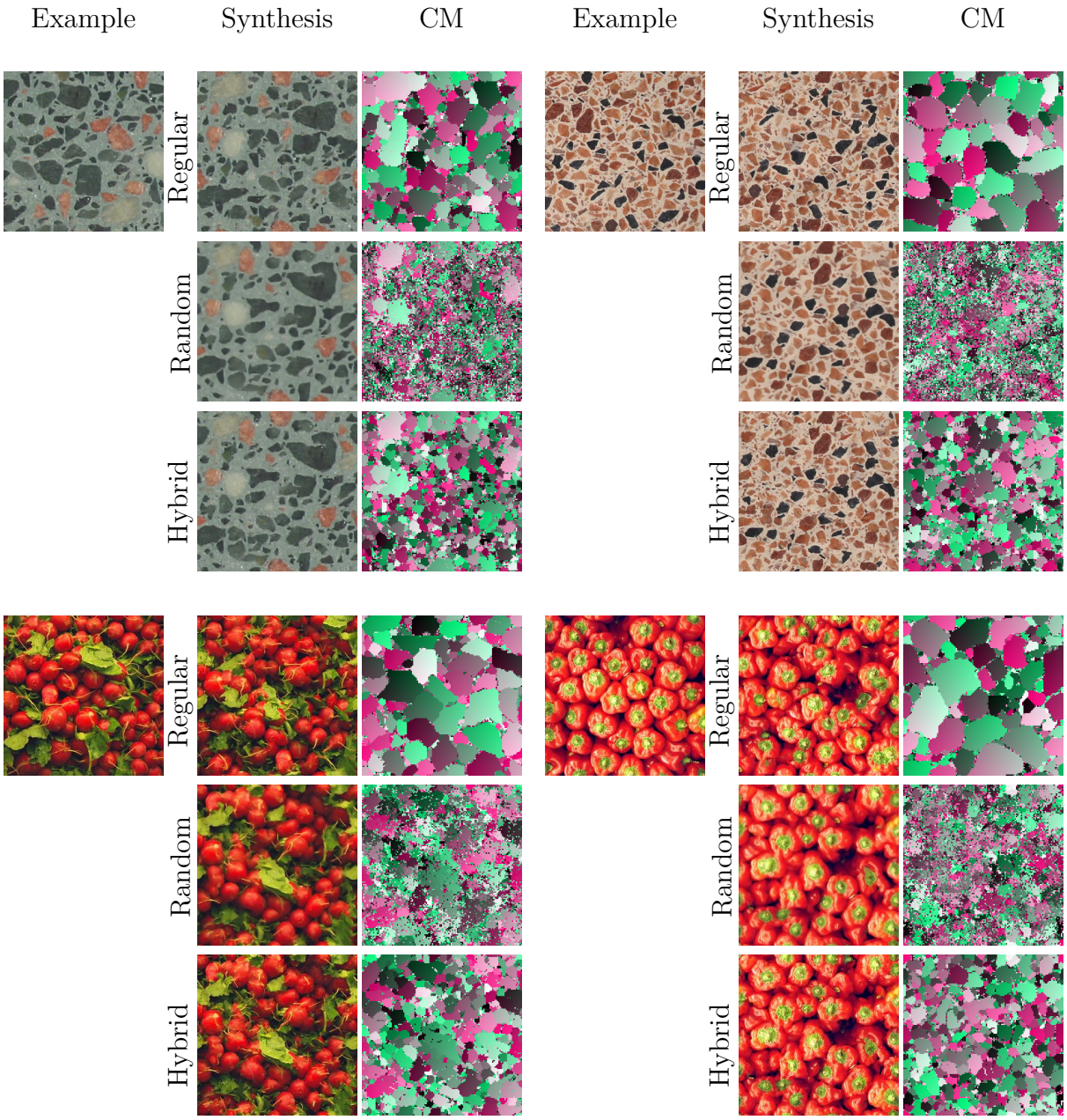



Figure 3.12 **Impact of the type of grids of pixels from the example in the patch assignment synthesis.** The hybrid synthesis refers to using random grids at the two coarse scales and a regular grid at the last, finest scale. The correspondence maps (CM) highlight the innovation/copy in the synthesis. All the images have a resolution of $256 \times 256$ and were synthesized using the same random *seed*.

local structured elements. Several issues can be noticed with the results from Kwatra et al. [6]. Without any statistical constraints, this method sometimes tends to *simplify* the output content. First, copies of large areas from the input may be used (first and second rows). Second, low-cost patch combination tends to be favored: constant regions therefore tend to appear (fourth row), as well as simplified characteristics (loss of white strokes in the third example, and black spots in the fifth row). Last, when a small set of patches allows a periodic compositing, this set tends to be reused over and over (rows 6 and 7). The same issues are also raised with the non-variational state-of-the-art patch-based approach called PatchMatch [18] (see last column of Figure 3.13). While our method is rather successful at both reproducing local elements and at preserving the global colors of the input, it has difficulty retaining long distance correlations on highly structured textures like in the last example of Figure 3.13. The trade-off between the number of scales used, the patches' width, and the size of the input limits the scale of the structures that can be captured. Thus, compared to using patches of multiple widths for each scale (as in Kwatra et al. [6]), our single-width approach loses long-distance correlations at a specific scale. However, this also means that smaller regions are copied from the input in comparison with Kwatra et al. [6], thereby avoiding repetitive artifacts (such as the shadows in the first example) and copies of large regions (second example).

**Discussion on patch aggregation** As described before, once each patch of $u$ is assigned to a patch $u_0$, the image $u$ is updated by minimizing the $\ell_{1,2}$-norm, which results in computing the color median among the overlapping pixels. As a comparison, Kwatra et al. [6] use an $\ell_2$-norm (with Gaussian weights) to the power $r = 0.8$, and use an iteratively reweighted least squares scheme to minimize this functional (note that this kind of methods encounter numerical issues). Figure 3.14 illustrates that our algorithm gives similar results when using this variant and that simply averaging by minimizing the squared $\ell_2$-norm leads naturally to blurry textures. We favored minimizing the $\ell_{1,2}$-norm since it is a well-posed convex problem.

**Limitations** A limitation of our approach, due to the strict assignment constraint, is the guarantee to synthesize the same distribution of patches as in the input. Although this can be considered as an advantage in most situations, it may lead to undesired effects when inputs do not sufficiently satisfy the stationarity hypothesis.

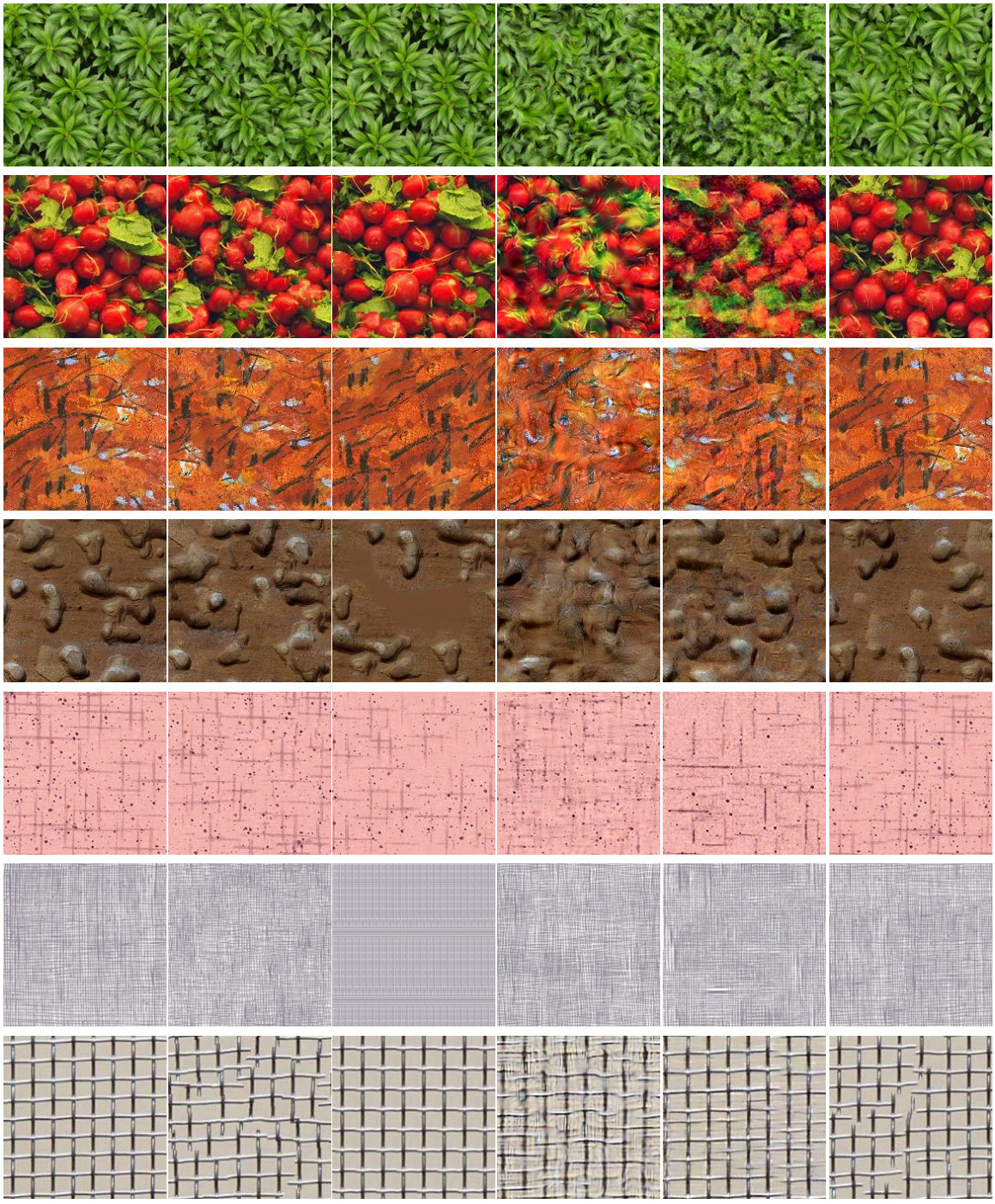



Figure 3.13 **Comparison with other methods.** We compare our approach with the patch-based method of Kwatra et al. [6], the two statistical matching methods of Portilla et al. [1] and Tartavel et al. [2], and the PatchMatch method of Barnes et al. [18] (with the multi-scale implementation of David Tschumperlé for texture synthesis [19]). These approaches are used with the default parameters described in the respective papers.



Figure 3.14 **Comparison of results for different values of r.** From left to right: $r = 2$, $r = 0.8$, $r = 0.8$ with Gaussian falloff weighting, and $r = 1$ using the same random initialization. Results are quite similar, except for $r = 2$ which is noticeably less sharp.

## 3.4 Relaxed Patch Assignment

As demonstrated in the previous section, the hard constraint of using all the patches of the example in the same proportion may lead to some limitations. On one hand it permits an inferior spatial coherence, which can create deformed patterns. On the other hand it is not able to reject irrelevant features (e.g., due to illumination, scale change, or artifacts) of examples that are not completely stationary. For instance the radishes example in the second row of Figure 3.13 contains essentially only one complete leaf, which makes it not completely stationary. Finally, the optimal assignment step is computationally expensive.

In this section we propose two relaxed assignment texture synthesis methods that have softer constraints on the synthesis. The first model is capable of ignoring some of the patches but still encourages a close patch distribution match. The second model aims at accelerating the synthesis. It softens the optimality constraint on the patch permutation. It no longer seeks the optimal assignment, only a low energy patch permutation. A parameter controls how far the algorithm is allowed to be from optimality. A similar approach proposed by Webster [84] is capable of producing high visual quality results. Another relaxation alternative, proposed by Galerne et al. [85], involves the pre-computation of the optimal transportation (a more general formulation of the optimal assignment) between the patches distributions. The result is then used to guide a patch-based transformation that complements a Gaussian random field model.

### 3.4.1 Relaxed Assignment Model

We define a new synthesis model relying on a *relaxed* assignment of patches. In this model the one-to-one assignment constraint of Problem (3.13) is relaxed to enable multiple matching of some example patches. Such an idea was first proposed in [86] and refined by [87] to

overcome the problem of color inconsistency in color transfer.

First, let us recall that an optimal assignment problem such as Problem (3.27) can be recast as a Linear Sum Assignment Problem [80] of the form:

$$\min_{\sigma\in\Sigma_{N^\downarrow}} \sum_{i\in\Omega^\downarrow} C_{i,\sigma(i)} = \min_{A\in\mathcal{A}} \sum_{(i,j)\in\Omega^\downarrow\times\Omega_0^\downarrow} A_{i,j}C_{i,j} \tag{3.28}$$

where $C$ is a fixed cost matrix that corresponds to the distance between patches, that is, $C_{i,j} = \|u\circ\mathcal{P}(i) - u_0\circ\mathcal{P}(j)\|_{1,2}$ and where

$$\mathcal{A} = \left\{A\in[0,1]^{N^\downarrow\times N^\downarrow},\ \forall\, i\le N^\downarrow\ \textstyle\sum_{j=1}^{N^\downarrow} A_{i,j} = 1,\ \forall\, j\le N^\downarrow\ \ \sum_{i=1}^{N^\downarrow} A_{i,j} = 1\right\},$$

is the set of bistochastic matrices (which is the convex hull of the set of permutation matrices).

Now, following [87], we consider the relaxed assignment problem:

$$\min_{P\in\mathcal{A}_k,\, k\ge 0} \textstyle\sum_{i,j} P_{i,j}C_{i,j} + \rho\sum_j |k_j - q_j|, \tag{3.29}$$

where the set of relaxed assignment matrices is defined as

$$\mathcal{A}_k = \{P\in[0,1]^{N^\downarrow\times N^\downarrow},\ \forall\, i\le N^\downarrow\ \textstyle\sum_{j=1}^{N^\downarrow} P_{i,j} = 1,\ \forall\, j\le N^\downarrow\ \sum_{i=1}^{N^\downarrow} P_{i,j} = k_j\}. \tag{3.30}$$

In this model, $q_j$ is the desired number of matches for the example patch indexed by $j$, and $k_j$ is the number of times this patch is actually matched. The relaxation parameter $\rho$ controls the soft constraint on assignment: if $\rho = 0$, the problem boils down to a nearest neighbor matching (as done in Kwatra et al. [6]), and for large enough values of $\rho$, the problem is the Optimal Assignment. To sum up, this model provides relaxed assignments that range from nearest neighbor matching to optimal assignment.

### 3.4.2 Results

Figure 3.15 illustrates the relaxed assignment between two point clouds and the effect of parameter $\rho$ with fixed value $q = 1$ (favoring using each patch once). As expected, we observe that for larger values of $\rho$ the relaxed matching is an optimal assignment, while smaller values of $\rho$ yield a relaxed assignment so that some points of the blue set may not be used.

Figure 3.16 illustrates the effect of using the relaxed assignment model for texture synthesis. We replace the optimal assignment step in Algorithm 1 with the relaxed assignment for

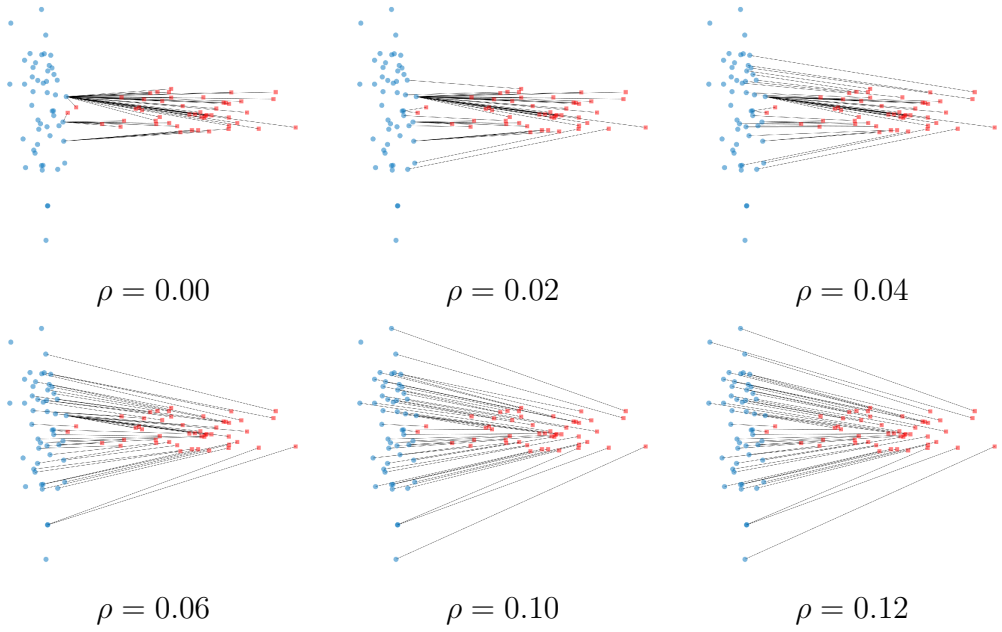



Figure 3.15 **Illustration of the affectation between two point clouds using the relaxed assignment model.** The two sets are differentiated by color and shape. The black lines indicate the affectation between pairs of points. The value of parameter $\rho$ controls the shift from a Nearest Neighbor affectation and an Optimal Assignment.

different values of $\rho$. These experiments use the same random initialization and the vector $q$ constant to 1.

The main practical interest of this model is that it allows for discarding some patches that may represent a rare pattern in the texture (e.g., the red dots in the first example of Figure 3.16 and the green leaf of the second example). Contrarily, some other features may now be replicated (the yellow dots in the first example, some radishes in the second). However, parameter $\rho$ does not offer an explicit control over the structures of the input image that may disappear. A solution to achieve this goal might be to let the user define $q$ more precisely.

## 3.5 Near-Optimal Patch Assignment

Most Texture Optimization methods use approximative nearest neighbor search algorithms in order to accelerate the patch projection step. This trades off optimality for speed without hurting the visual quality—if anything, in nearest neighbor models, this improves quality as it reduces the growth of "garbage" by introducing some variability. Similarly, when using the Optimal Assignment, it is clear that an important quantity of patches are not associated with

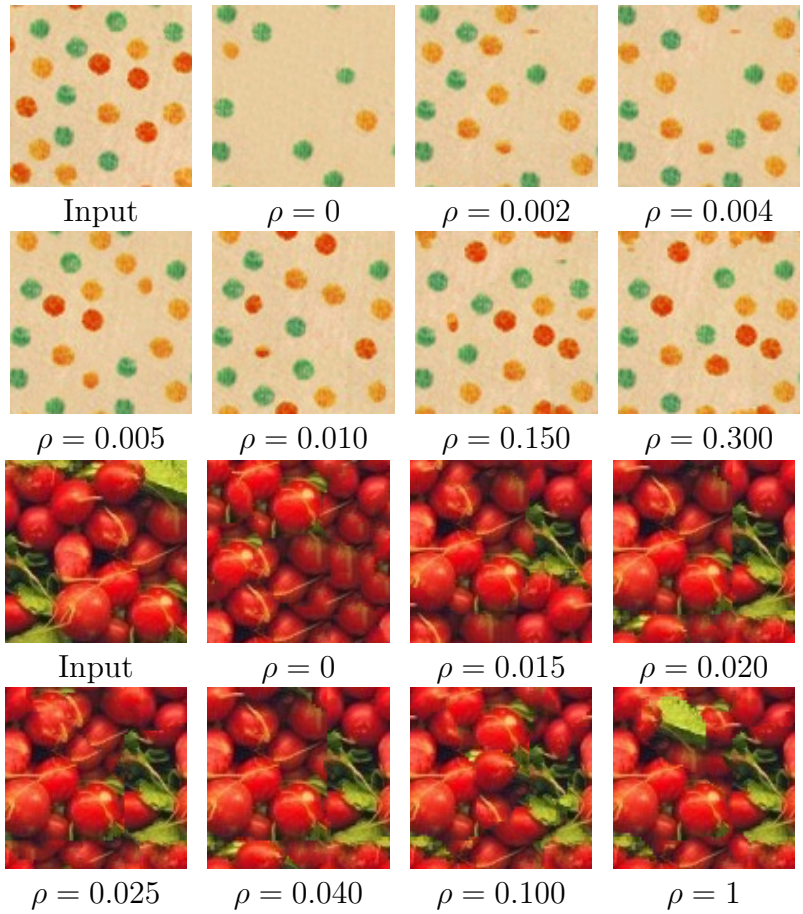



Figure 3.16 **Texture synthesis with relaxed assignments for different values of the relaxation parameter $\rho$.** The value $\rho = 0$ corresponds to nearest neighbor matching, which does not respect patch distribution. Increasing $\rho$ gives relaxed assignments that allow for slight variations of the synthesized sample's patch distribution. For large values of $\rho$, the assignment is an optimal permutation resulting in the reproduction of the characteristics of the input.

their nearest neighbor (except for special cases, e.g., the identity). This does not prevent from obtaining good quality and locally coherent results as shown in Section 3.3. In this section we exploit the nature of the Auction Algorithm in order to perform near-optimal patch assignment texture synthesis. We claim that we can obtain high quality textures using a near-optimal permutation of patches and thus saving computation times.

### 3.5.1 Auction Algorithm

The *Auction algorithm* [88], as the *Hungarian algorithm*, is a specific method for solving the Linear Sum Assignment Problem (LSAP). One advantage of the auction algorithm is that most of its computations can be performed in parallel [89]. Given the experimental results

of Vasconselos et al. [82] we can expect a relevant acceleration of the optimal patch assignment synthesis by using a parallel implementation of the Auction algorithm for solving the projection step. Additionally, the Auction algorithm is capable of obtaining intermediate sub-optimal solutions for the LSAP that are faster to obtain. These solutions are still permutations between the two sets and have a primal-dual gap regarding the objective function in (3.28) with the optimal solution controlled by a parameter $\epsilon$.

Originally the Auction algorithm [88] solves the maximization version of the LSAP:

$$\max_{\sigma \in \Sigma_{N^{\downarrow}}} \quad \sum_{i \in \Omega^{\downarrow}} \tilde{C}_{i,\sigma(i)}, \tag{3.31}$$

which is equivalent to the LSAP by setting $\tilde{C}_{i,j} = -C_{i,j}$. The algorithm finds the *assignment* $\sigma$ by solving the dual formulation of the LSAP:

$$\min_{a,\rho} \quad \sum_{i=1}^{N} a_i \quad + \quad \sum_{j=1}^{N} \rho_j \tag{3.32}$$

$$\text{s. t.} \quad a_i + \rho_j \geq \tilde{C}_{i,j}, \tag{3.33}$$

where vectors $a$ and $\rho$ are the dual variables. For a given $\rho$ the problem is minimized when $a_i$ is equal to the maximum value of $\tilde{C}_{i,j} - \rho_j$. Therefore we can consider vector $\rho$ as the only optimization variable.

The algorithm is often presented in the context of an auction. The two sets to match are called persons $i \in \{1, ..., N\}$ and objects $j \in \{1, ..., N\}$. The matrix $\tilde{C}$ relates to the *benefit* that person $i$ attributes to object $j$. An assignment $S$ is a (possibly empty) set of person-object pairs $(i, j)$ such that for each person $i$ there is at most one pair $(i, j) \in S$ and for each object $j$ there is at most one pair $(i, j) \in S$. $\rho$ is called the price vector and $\rho_j$ the price of object $j$. For a given $\rho$ the *value* of an object $j$ for a person $i$ is:

$$v_{i,j} = \tilde{C}_{i,j} - \rho_j,$$

and the *profit* $\pi_i$ of person $i$ is the maximum value of objects $j$:

$$\pi_i = \max_j v_{i,j}.$$

From linear programming theory [88], a complete assignment $S = \{(i, j) : i = 1, ..., N\}$ and a price vector $\rho$ are primal and dual optimal if and only if:

$$\max_k \{\tilde{C}_{i,k} - \rho_k\} = \tilde{C}_{i,j} - \rho_j \qquad \text{for each} \quad (i, j) \in S,$$

which is known as the *complementary slackness condition*. The key feature of the auction algorithm is that it looks for an assignment $S$ and a set of prices $\rho$ that satisfy:

$$\max_k\{\tilde{C}_{i,k} - \rho_k\} - \epsilon \leq \tilde{C}_{i,j} - \rho_j \qquad \text{for each} \quad (i,j) \in S. \tag{3.34}$$

Called the $\epsilon$-*complementary slackness* and $\epsilon$ is a nonnegative constant. For an integer benefit matrix $\tilde{C}$, a complete assignment $S$ with a price vector that satisfies Eq. (3.34) is optimal if $\epsilon < \frac{1}{N}$.

Here we describe the auction algorithm [89]. It starts with an $\epsilon > 0$ and an initial assignment $S$ and prices $\rho$ that satisfy Eq. (3.34), then proceeds by iterating two phases, the *bidding phase* and the *assignment phase*.

- In the *bidding phase* for each person $i$ that is not assigned in $S$:
  Find the "best" object $j^*$ giving the maximum value

  $$v_{i,j^*} = \max_j v_{i,j},$$

  then find the value of the "second best" object

  $$w_{i,j^*} = \max_{j, j \neq j^*} v_{i,j}.$$

  Finally compute the bid $\beta_{i,j}$ of person $i$ for the object $j^*$

  $$\beta_{i,j^*} = \rho_{j^*} + v_{i,j^*} - w_{i,j^*} + \epsilon.$$

- The *assignment phase* starts with the sets $I(j)$ of persons that set a bid for object $j$.
  For each object $j$:
  If $I(j)$ is nonempty, increase $\rho_j$ to the highest bid:

  $$\rho_j := \max_{i \in I(j)} \beta_{i,j},$$

  then if the pair $(i,j)$ exists in $S$, remove it and add the pair $(i^*, j)$, where:

  $$i^* = \arg\max_{i \in I(j)} \beta_{i,j}.$$

  If $I(j)$ is empty, $\rho_j$ is not updated and $j$ remains unassigned in $S$.

The algorithm terminates when a complete assignment is obtained, i.e., $|S| = N$.

### 3.5.2 Near-Optimal Patch Assignment Model

We modify the texture synthesis model in Eq. (3.1) to allow the sub-optimality of the mapping $\sigma$. We define $\Sigma^E_{N^\downarrow}$ as the set of permutations of $N^\downarrow$ elements for which the sum of distance between the associated patches is close to the optimal by a value $E$. In practice, the distinction with the optimal model is only in the *projection step* where the algorithm does not have to find an optimal mapping $\sigma$ but only a *near-optimal* permutation $\sigma \in \Sigma^E_{N^\downarrow}$ where:

$$\Sigma^E_{N^\downarrow} = \left\{ \sigma \in \Sigma_{N^\downarrow},\ \sum_{x \in \Omega^\downarrow} \|u \circ \mathcal{P}(x) - u_0 \circ \mathcal{P} \circ \sigma(x)\|_p^r - E = \min_{\sigma \in \Sigma_{N^\downarrow}} \sum_{x \in \Omega^\downarrow} \|u \circ \mathcal{P}(x) - u_0 \circ \mathcal{P} \circ \sigma(x)\|_p^r \right\}.$$

The *relaxed* nature of the auction algorithm allows us to find a near-optimal permutation and having control over the gap with the optimal solution. Recall the $\epsilon$-*complementary slackness* condition with an explicit assignment $\sigma$:

$$\max_k \{-C_{i,k} - p_k\} - \epsilon \leq -C_{i,\sigma(i)} - \rho_\sigma(i) \qquad \text{for all} \quad i \in \{1, ..., N^\downarrow\}, \tag{3.35}$$

with $C_{i,j} = \|u \circ \mathcal{P}(i) - u_0 \circ \mathcal{P}(j)\|_{1,2}$ and $\rho$ the dual variable. It can be shown that for a given $\epsilon$ the auction algorithm obtains an assignment that is within $N^\downarrow \epsilon$ of being optimal [89]. This thanks to the complementary slackness condition allowing for each person-object assignment to only be within $\epsilon$ of being optimal.

Values of $\epsilon$ that give a non-optimal solution are actually used in the auction algorithm when implemented in a recursive form ($\epsilon$-scaling). The algorithm is applied several times, using each resulting price vector $\rho$ as initialization to the next algorithm call, and it starts with a large value of $\epsilon$ that is decreased each time. This technique accelerates the convergence to the optimal values because the algorithm is faster at obtaining solutions for large values of $\epsilon$ and those sub-optimal solutions are good initializations for a more precise algorithm call.

Solving the projection step using the auction algorithm with a predefined relaxed $\epsilon$ we can expect an important acceleration on the synthesis.

### 3.5.3 Results

Figure 3.17 illustrates the *near-optimal* assignment between two point clouds and the effect of the parameter $\epsilon$. Using this model, all points are always affected evenly, that is, the result is always a permutation. Small values of $\epsilon$ yield an affectation close to the optimal assignment. If $\epsilon$ is large compared to the individual distance between points, then the Auction algorithm behaves similarly to sequentially using the nearest neighbor projection but without reusing already affected points.

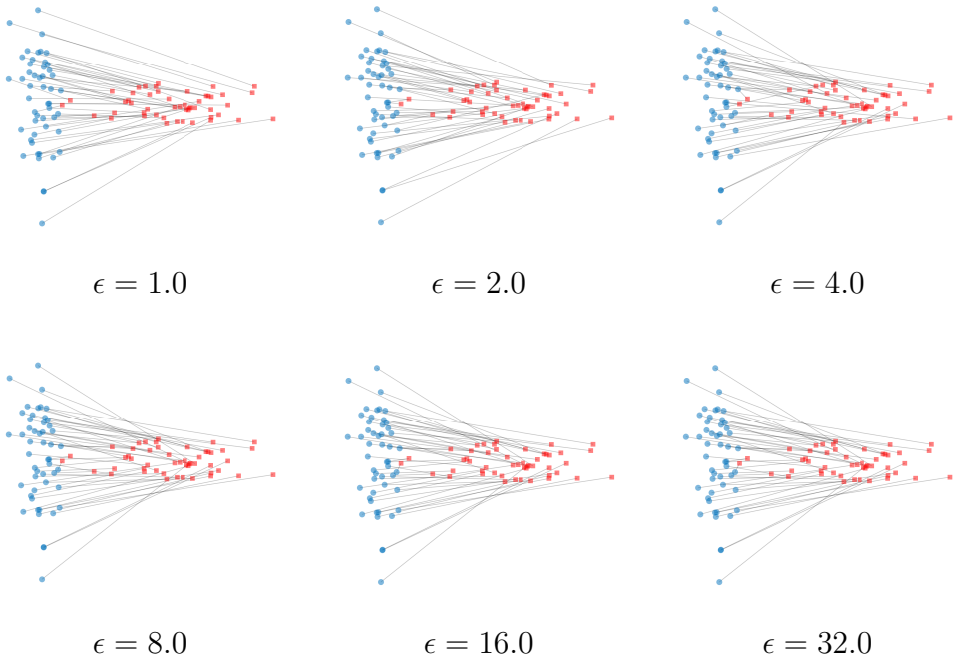



Figure 3.17 **Illustration of the affectation between two point clouds using the near-optimal assignment model.** The two sets are differentiated by color and shape. The black lines indicate the affectation between pairs of points. The value of the parameter $\epsilon$ controls the optimality of the affectation. In all cases the result is a permutation, but for small values of $\epsilon$ the energy is smaller.

Figures 3.18 compares the synthesis results of the near-optimal assignment with the optimal assignment model. For each value of $\epsilon$, the synthesis algorithm is run with the same parameters. We use three scales, a single patch size of $w = 8$ with a stride of $\frac{w}{4}$, and $50, 50, 10$ iterations corresponding to the coarse to fine scales.

We observe that the visual quality is maintained for small values of $\epsilon$, with the synthesized samples having spatial coherence as good as in the optimal synthesis. Figure 3.19 shows the value of the optimization energy through the synthesis process for the two textures of Figure 3.18. Contrary to what we expected, the near-optimal model with $\epsilon \in \{1, 2\}$, attains a lower value of the energy than the optimal assignment model ($\epsilon = 0$). Recall that the model in Eq. (3.13) is not convex, thus, a better assignment during the projection step does not guarantee a better result of the whole process. The lower value of the energy might be caused by larger smooth regions in the correspondence map (CM).

Figure 3.20 shows the computation times for different values of $\epsilon$ for more textures of the VisTex database. This confirms that most of the times, using a larger value of $\epsilon$ reduces computation times. For the settings used here, a value of $\epsilon = 32$ always accelerates the

synthesis, for as much as 80%, however, this value might be too large for the model to maintain good spatial coherence between not smooth regions of the synthesis result.

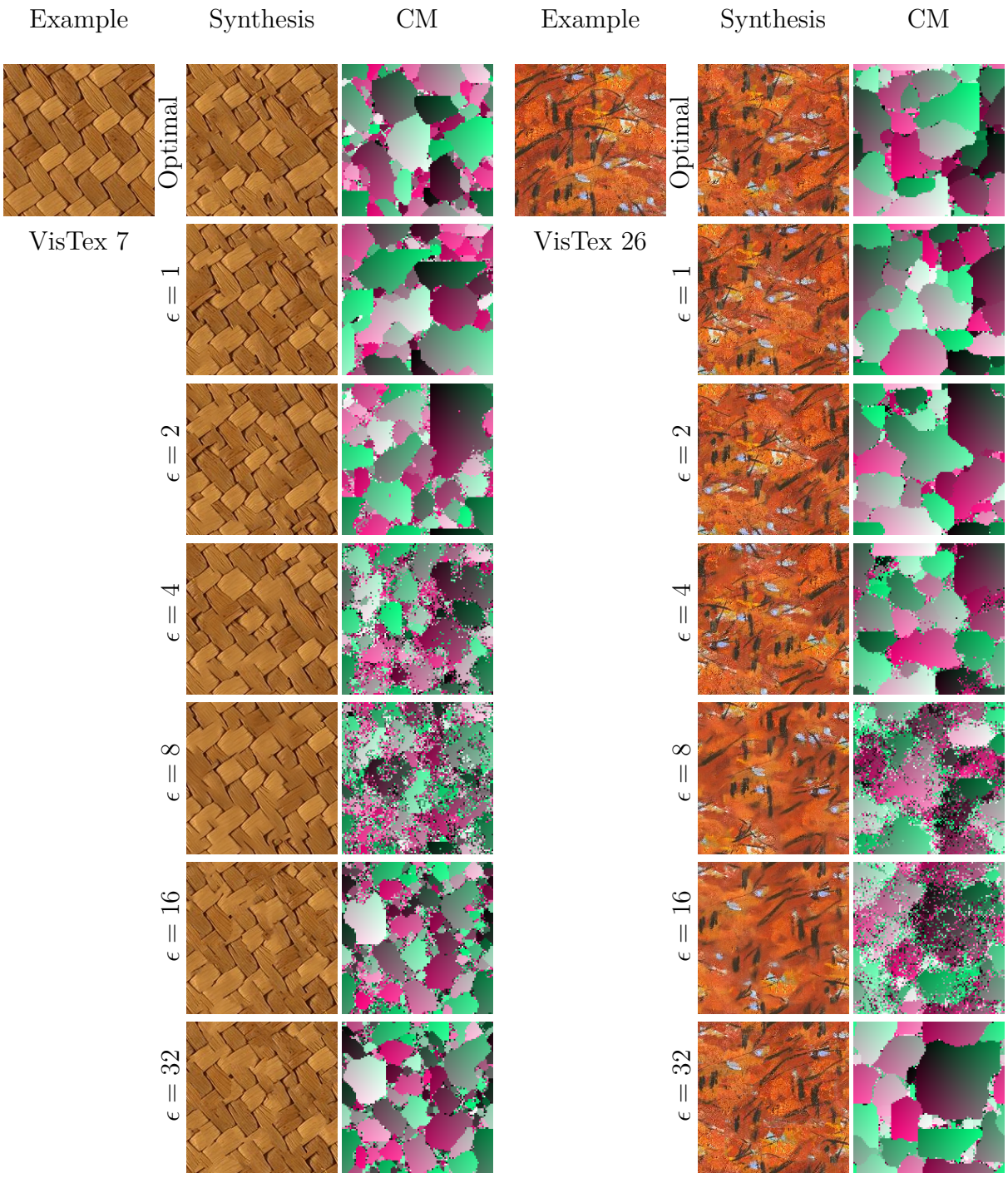



Figure 3.18 **Texture synthesis using the near-optimal patch assignment model**. The top row shows the example and the result of using the optimal assignment model of Section 3.3 ($\epsilon = 0$). The remaining rows show the results using the near-optimal model with varying values of parameter $\epsilon$. All the images have a resolution of $192^2$ pixels and are normalized to values between $[0, 1]$. All the results were synthesized using the same random *seed*.

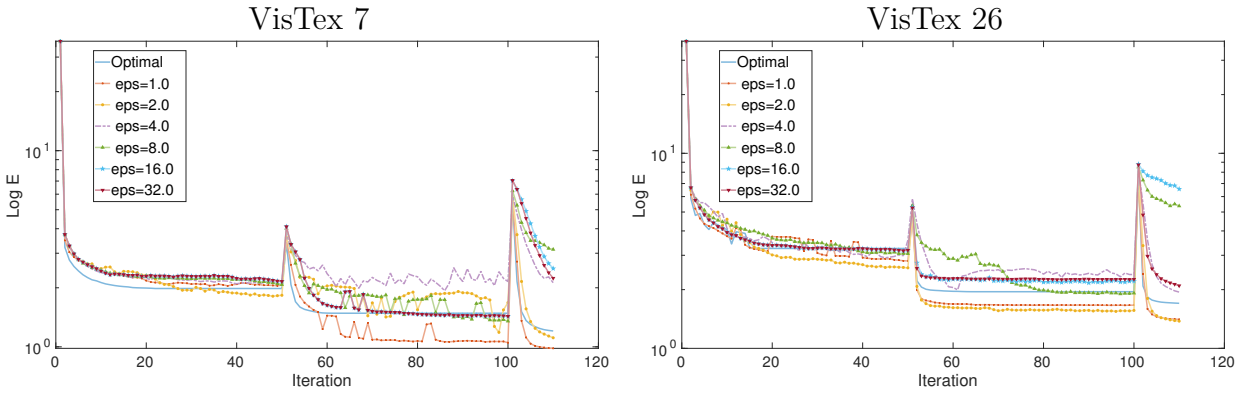



Figure 3.19 **Optimization curves for the near-optimal patch assignment model.** The curves correspond to the results in Figure 3.18.

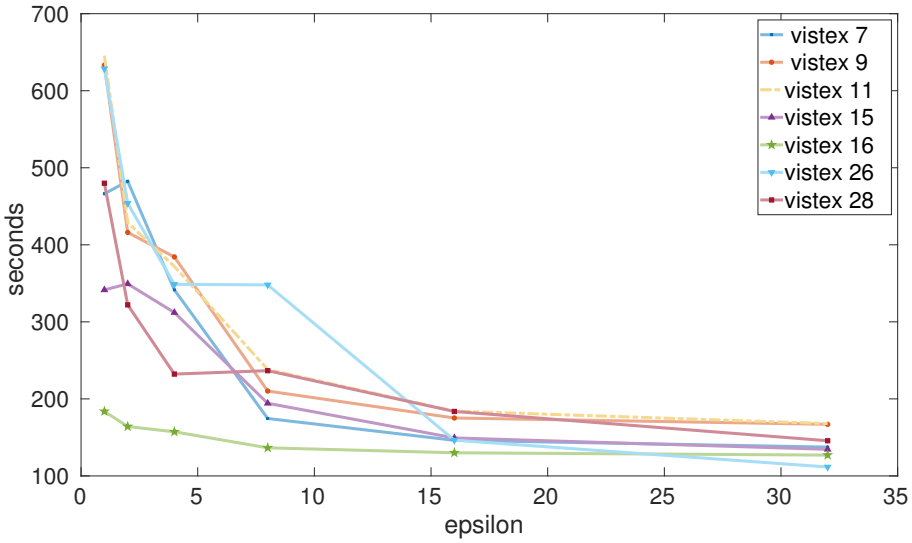



Figure 3.20 **Computation times for different values of** $\epsilon$. We use the near-optimal patch assignment model on several examples from the VisTex database. The resolution of the example textures and synthesized samples were $192^2$ pixels for this experiment.

## 3.6 Conclusion

The main goal of this chapter was to propose a robust example-based 2D texture synthesis method capable of producing high quality samples for different types of textures. The proposed model integrates statistical constraints on the patch distribution in a Texture Optimization formulation [6]. This allows us to bring together the local visual quality of the patch-based methods and the stability of statistical matching methods. Our Optimal Assignment model confirms our hypothesis that matching the patch distribution of the example (while optimizing spatial coherence) results in good quality samples and minimizes the risk of introducing blurry or flat regions. We conclude that overall, the Optimal Assignment is a better metric for measuring texture similarity than the nearest-neighbor model.

The random initialization effectively avoids producing a verbatim copy of the example. We observe, however, a low capacity for innovation due to the piece-wise identity behavior of the results. We promote innovation without sacrificing quality by using randomly sampled patches from the example at lower resolutions of the synthesis process. Additionally, we proposed a relaxed assignment model that is capable of controlling the degree of compliance with the patch distribution of the example.

The biggest limitation of constraining the patch statistics, as we formulated it, is the computation of the patch assignment or the solution to the relaxed assignment. The proposed Near-optimal Patch Assignment Model attempts to mitigate the computational burden. This opens the path to investigate other relaxed methods with faster convergence.

Part of the work presented in this chapter was published in the proceedings of the international conference on Scale Space and Variational Methods in Computer Vision (SSVM) in 2017.

# CHAPTER 4 ON-DEMAND SOLID TEXTURE SYNTHESIS USING DEEP CONVOLUTIONAL NETWORKS

In this chapter we propose a method for on-demand solid texture synthesis from 2D examples. The proposed method extends the Texture Networks framework [8], formerly described in Section 2.1.3. First, we propose a CNN capable of generating solid textures on-demand [64, 69]. The proposed strategy for on-demand evaluation can be used in any fully convolutional generative model. Training the proposed CNN involves a large memory footprint that we overcome using a new slice-based training strategy. This allows the model to handle example image resolutions larger than previous methods. Like some previous methods, our model is capable of generating new samples of any size, given enough memory is available. We show that the model is successful at reproducing isotropic solid materials' visual characteristics. And finally we study the anisotropic setting (i.e., where the patterns are oriented) and discuss the limits of the cross-section based definition of the example-based solid texture synthesis problem.

We start this chapter with a detailed overview of the seminal model of Ulyanov et al. [8] in Section 4.1. Besides laying the basic concepts of the Texture Networks framework, it allows us to study different training settings that can be directly used in our solid texture model. The rest of the chapter is dedicated to presenting our contributions for solid texture synthesis.

## 4.1 2D Texture Networks Overview

Johnson et al. [25] and Ulyanov et al. [8] demonstrated that it is possible to train a feedforward (without recurrent connections) *generator network* to produce textures with visual quality comparable to the state of the art—roughly defined by Gatys et al. [7]—for texture synthesis and style transfer. While the method of Gatys et al. [7] performs a full optimization procedure to synthesize each new sample, as a solution to an inverse problem, these explicit methods map a random input to a texture image. The parameters of the generator network are optimized during the training stage to produce samples similar to a given example.

These methods use the pre-trained VGG-19 [50] *descriptor* network, in the same manner as Gatys et al. [7] discussed in Section 2.1.3, to compare the visual characteristics between the generated and example images. Training a generator network can be more demanding than optimizing a single image, since there is no spatial regularity, however, this training stage only needs to be done once for a given example. After training, the generator efficiently

generates samples similar to the example texture by forward evaluation.

Texture networks are easier to train that GAN-based methods [10, 60] and are better suited to texture synthesis where there is only one example (since they leverage the information contained in the pre-trained VGG-19). In contrast GANs require a large set of texture samples to define the distribution of the example texture. Jetchev et al. [10, 60] overcome this using many randomly drawn large patches from the example.

### 4.1.1 Perceptual Loss

The perceptual loss, proposed by Gatys et al. [7] (the term was coined by Johnson et al. [25]), is so far the most successful way to characterize textures for the purpose of texture synthesis. It summarizes the visual characteristics of a texture using the Gram matrices computed from the feature maps at several internal layers of a pre-trained descriptor network. This descriptor is generally a truncated instance of the VGG-19 classification network [50]. The perceptual loss then compares two textures by measuring a distance between the Gram matrices of the feature maps.

Formally, the feature maps result from the evaluation of the descriptor network $\mathcal{D}$ on an image, i.e., $\mathcal{D} : x \in \mathbb{R}^{N_1 \times N_2 \times 3} \mapsto \{F^l(x) \in \mathbb{R}^{N^l \times M^l}\}_{l \in L}$, where $L$ is the set of considered VGG-19 layers, each layer $l$ having $N^l$ spatial values and $M^l$ feature 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

$$G^l(x) = \frac{1}{N^l} F^l(x)^T F^l(x), \tag{4.1}$$

where $T$ indicates the transpose of a matrix. The loss between an example texture $u$ and a generated sample $v$ is

$$\mathcal{L}(v, u) = \sum_{l \in L} \frac{1}{(M^l)^2} \left\| G^l(v) - G^l(u) \right\|_F^2, \tag{4.2}$$

where $\| \cdot \|_F$ is the Frobenius norm.

The Gram matrices are computed along spatial dimensions to take into account the stationarity of the texture. They encode both first and second order informations of the feature distribution (covariance and mean).

### 4.1.2 Generator's Architecture and Training

Figure 4.1 shows the framework proposed by Ulyanov et al. [8]. The generator network $\mathcal{G}$ with learnable parameters $\theta$ takes a multi-scale noise input $Z = \{z_0, \ldots, z_K\}$ and processes it to produce a color image $v = \mathcal{G}(Z|\theta)$. The goal of this approach is to find a set of parameters $\theta$

such that the generator produces texture samples that have the same visual characteristics as the example $u$. To accomplish this, an optimization procedure (detailed below) is performed to generate samples as close as possible to the example as measured by the perceptual loss Eq. (4.2). The descriptor network is a pre-trained, truncated instance of VGG-19. After such training stage, the generator can synthesize new texture samples similar to the example. One instance of the generator can only be trained to generate one texture.

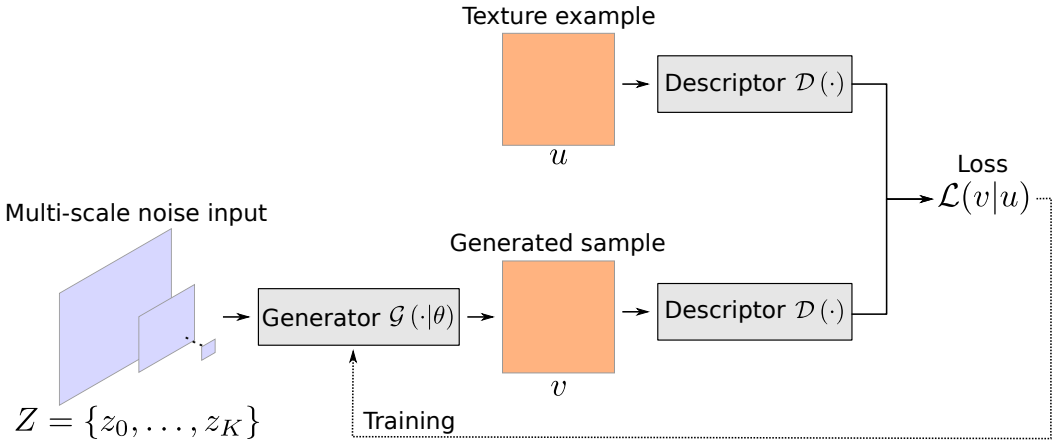



Figure 4.1 **Training framework of Ulyanov et al. [8].** The CNN Generator network $\mathcal{G}(\cdot|\theta)$ processes a multi-scale noise input $Z$ to produce a new texture sample $v$. The loss $\mathcal{L}$ compares the feature statistics induced by the example $u$ in the layers of the pre-trained Descriptor network $\mathcal{D}(\cdot)$ with those induced by the generated sample $v$. The training iteratively updates the parameters $\theta$ to reduce the loss.

### Generator's Architecture

Figure 4.2 details the generator network proposed by Ulyanov et al. [8]. It depicts the process of producing a sample from a multi-scale set of noise inputs $Z = \{z_0, \ldots, z_K\}$. It consists of three main linear operations, convolution, concatenation, and upsampling, and the non-linear *rectified linear unit* (ReLU) operation. From left to right, the generator applies a convolution block to each noise input. Then, it goes from the smallest scale to the biggest final scale by upsampling and concatenating the partial results at the corresponding scales. There are $K$ upsampling and concatenation operations followed by a final single convolution layer that maps the number of channels to three to get a color texture. Here we consider the output to be defined in a square lattice for simplicity. The size $N$ can be any multiple of $2^K$, and the size of each noise input $z_k$ is $\frac{N}{2^k}$, $k = 0, \ldots, K$. In the following paragraphs we detail the three different blocks of operations used in the generator network.

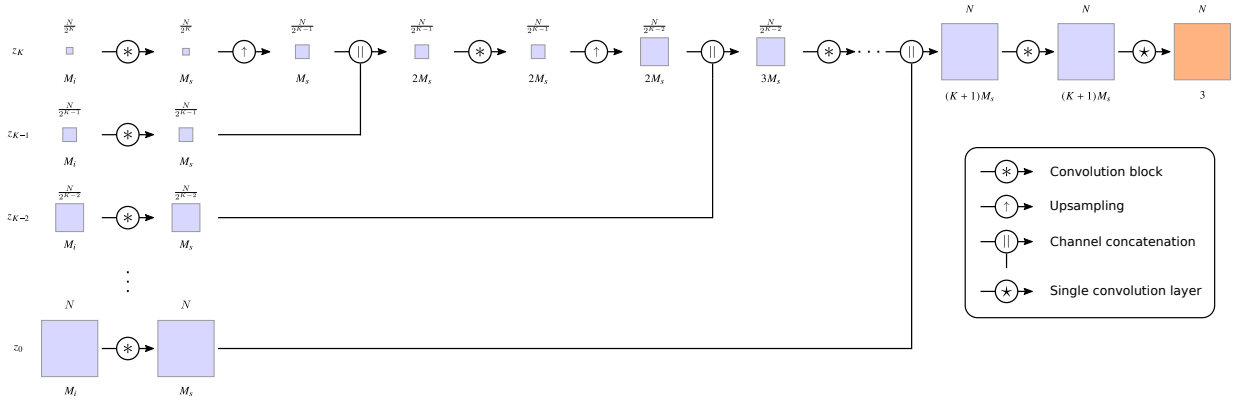



Figure 4.2 **Ulyanov et al. [8] generator's architecture**. It processes a set of noise inputs $Z = \{z_0, \ldots, z_K\}$ at $K+1$ different scales using convolution operations and non-linear activations. The information at different scales is combined using upsampling and channel concatenation. $M_i$ indicates the number of input channels and $M_s$ controls the number of channels at intermediate layers. For simplicity we consider square outputs with spatial size of $N_1 = N_2 = N$. For each intermediate feature the spatial size is indicated above and the number of channels below.

**Convolution Block** A convolution block applies a sequence of three ordinary convolution layers, each of them followed by a batch-normalization and a ReLU activation. The first convolution layer gets an input with $M_{in} = 3$ channels and produces an output with $M_{out}$ channels. The following two convolution layers keep the number of channels constant at $M_{out}$. The size of the kernels is $3 \times 3$ for the first two layers and $1 \times 1$ for the last. The convolutions are computed densely and using circular padding. This keeps the spatial size of the input constant and produces outputs that are periodic.

**Upsampling** An upsampling performs a nearest-neighbor upsampling by a factor of 2 on each spatial dimension (i.e., each pixel is replicated 4 times).

**Channel Concatenation** A channel concatenation first applies a batch normalization operation [90], which normalizes the values using a running estimate of the mean and variance across mini-batches per dimension, and then concatenates the channels of two multi-channel inputs having the same spatial size.

The learnable parameters $\theta$ of the generator are: the convolution's kernels and bias, and the batch normalization layers' parameters, i.e., weights, bias, mean, and variance.

### Training

In Ulyanov et al. [8], training the generator $\mathcal{G}(Z|\theta)$ with parameters $\theta$ corresponds to minimizing the expectation, accordingly to the distribution $\mathcal{Z}$ of random input $Z$, of the loss in Eq. (4.2) given the example $u$

$$\theta_u \in \underset{\theta}{\operatorname{argmin}} \quad \mathbb{E}_{\mathcal{Z}} \left[ \mathcal{L}(\mathcal{G}(Z, \theta), u) \right], \tag{4.3}$$

where $\mathcal{Z}$ is the distribution of independent, multi-scale random uniform samples in the interval $[0, 1]$.

The optimization is performed using back propagation and a gradient-based optimization algorithm using batches of random inputs. Once trained, the generator is able to quickly generate samples similar to the input example by forward evaluation.

### 4.1.3 Analysis, Results, and Experiments

In this section we study the method of Ulyanov et al. [8] using our own implementation. We focus on the visual performance for textures with discernible patterns, which are the most complex to synthesize for other state-of-the-art methods (contrarily to fully stochastic textures). We study the impact on the visual quality of using different training settings such as: the example texture resolution, the type of pooling in the layers of VGG, and gradient normalization [8, 57].

### Implementation

We implement the method of Ulyanov et al. [8] with the following settings:

- **Generator** We set the number of scales to 6, i.e., $K = 5$. We use $M_i = 3$ input channels and $M_s = 8$ (number of channels after the first convolution block and *channel step* across scales) as in [8]. This results in a compact network with $\sim 1.16 \times 10^5$ parameters.

- **Descriptor Network** We use a truncated VGG-19 [50] as descriptor network, with padded convolutions and average pooling. We consider the following layers for the loss: `relu1_1`, `relu2_1`, `relu3_1`, `relu4_1`, and `relu5_1`.

- **Training** We use the *PyTorch* library for Python and the pre-trained parameters for VGG-19 available from the BETHGE LAB [7, 91]. We optimize the parameters of the

generator network using Adam algorithm [92] with a learning rate of 0.1 during 3000 iterations (coefficients for the running averages (betas) by default to 0.9 and 0.999). 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. We used batch sizes from 4 to 16 samples without noticing any impact on the training.

### Example of convergence during training

Figure 4.3 shows samples produced with the generator at different stages of training using the same noise input. The generator first adjusts to produce similar colors as the example. The initial patterns have smooth edges that evolve towards the sharper forms of the example. The fine detail seems to emerge later in training along with the less present colors of the example.

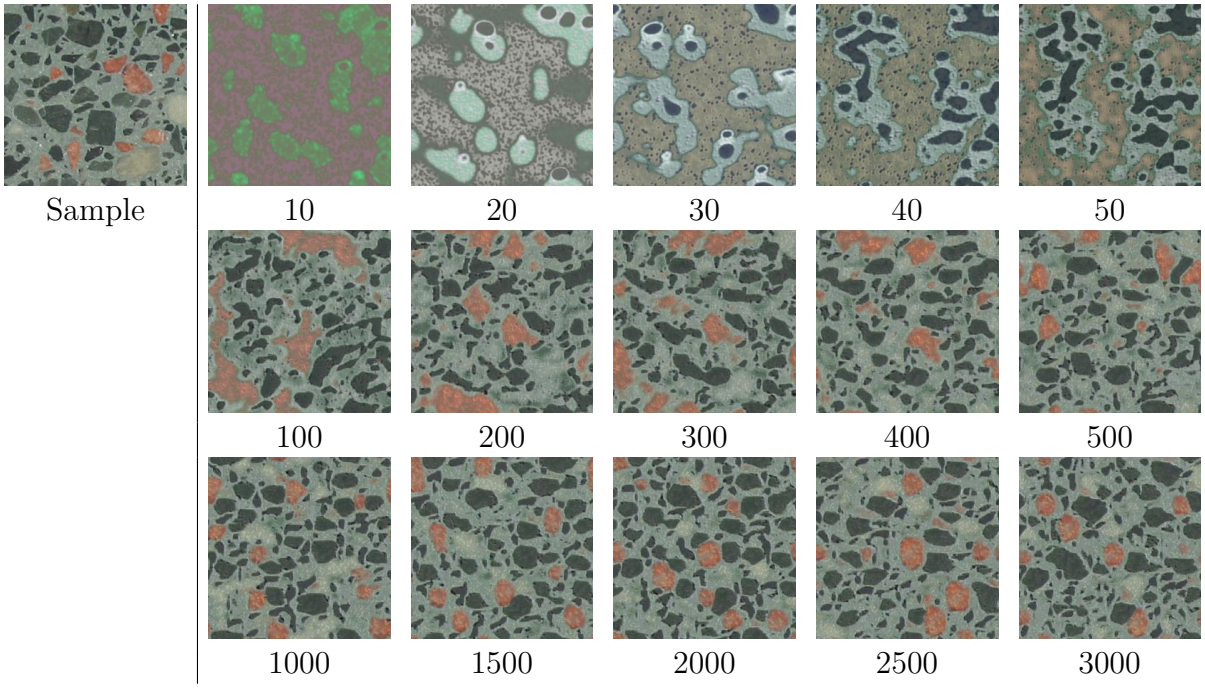



Figure 4.3 **Evolution of the generator in our implementation of Ulyanov et al. [8] method**. Samples generated at different stages of training indicated by the iteration number. We used the same noise input for all the samples.

### Visual quality of the results

Figure 4.4 shows some results using our implementation. We train the generator using example images of size $256^2$ pixels and generate samples of $512^2$ pixels. The method is capable of capturing the colors and most of the patterns present in the different examples. Yet, there is still room for improvement, such as preserving the perfect alignment of the swirly squares texture (bottom row, left) and the long distance relationships in the lined texture (third row, left). Finally, the method is not successful on the pebbles texture (second row, left) where it mostly reproduces one type of patterns (pebble) and ignores the diversity in the example. This can be a perturbation caused by the caption in the bottom left side of the example.

Figure 4.5 shows the value of the empirical estimation of the loss $\mathcal{L}$ during the training of three examples shown in Figure 4.4, dotted (first row left), leafy (first row, right), and jagnow (second row, right). The loss value first decreases fast and then oscillates while decreasing slowly through the iterations.

### Impact of the image resolution used during training on the quality of the results

Here we study the visual quality reached by a generator trained with an example texture at different resolutions. The descriptor network was trained with images at a fixed size, thus it is relevant to analyze if the framework performs better at a given resolution. Figures 4.6 and 4.7 show the results of training several textures with different resolutions. We tested $128^2$, $256^2$, $512^2$, and $1024^2$ pixels examples. During training the generator synthesizes a batch of images at the same resolution as the example. Naturally, lower resolution images are blurry and less detailed than the high resolution versions. But we observe further degradation of the results for images of $128^2$, that is, the result is a poor reproduction of its corresponding example. The generator is capable of capturing the fine details at higher resolutions, but it seems to begin losing the global arrangement for examples at $1024^2$ pixels.

### Impact of different pooling strategies for VGG on the quality of the results

Gatys et al. [7] claim that using average pooling in the layers of VGG helps to obtain a better visual quality for their method. In contrast, Ulyanov et al. [8] use max pooling in their implementation. In Figure 4.8 we compare the results of these two settings. We observe a slightly better result using average pooling where the edges of the patterns look sharper.

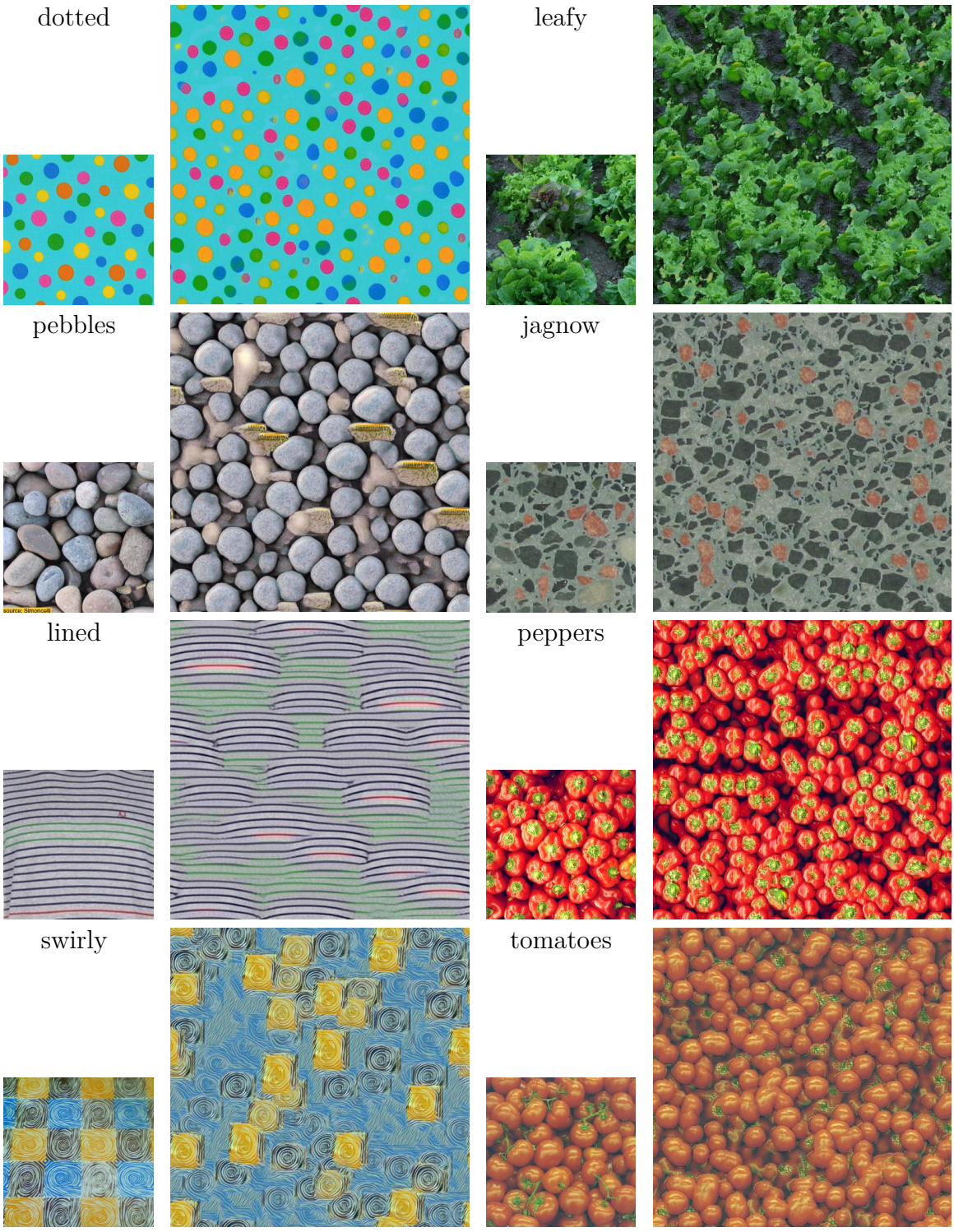



Figure 4.4 **Results using our implementation of Ulyanov et al. [8].** Left, example ($256^2$ pixels), right, synthesized sample of twice the side size. We ran the training for 5000 iterations with 4 samples per mini-batch and a learning rate starting at 0.1 and decreasing by 0.8 every 1000 iterations.

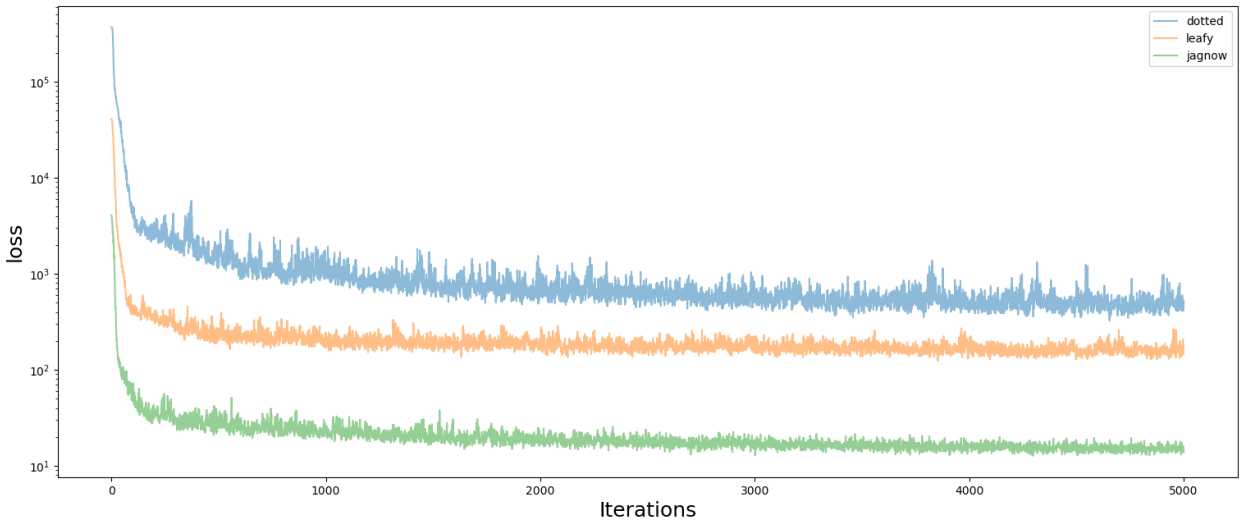



Figure 4.5 **Value of the empirical loss during the training of the generator for the textures dotted, leafy, and jagnow of Figure 4.4.**

### Impact of using gradient normalization during training

Ulyanov et al. [8] perform a *gradient normalization* during training that supposedly improves the visual quality of the results. They modify the gradient computation so that the gradients of the loss with respect to each feature-map output of VGG has a $L_1$ norm unitary before continuing the back-propagation of the gradient. Figure 4.9 compares the results of training with and without gradient normalization. We do not observe any significant improvement by using gradient normalization.

## 4.2 Overview of the Proposed Method for Solid Texture Synthesis

The following sections present our method that extends Texture Networks [8] for solid texture synthesis and on-demand evaluation.

Figure 4.10 outlines the proposed method. We perform solid texture synthesis using the convolutional neural *generator* network $\mathcal{G}$ detailed in Section 4.3. The generator with learnable parameters $\theta$, takes a multi-scale volumetric noise input $Z$ and processes it to produce a color solid texture $v = \mathcal{G}(Z|\theta)$. 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.

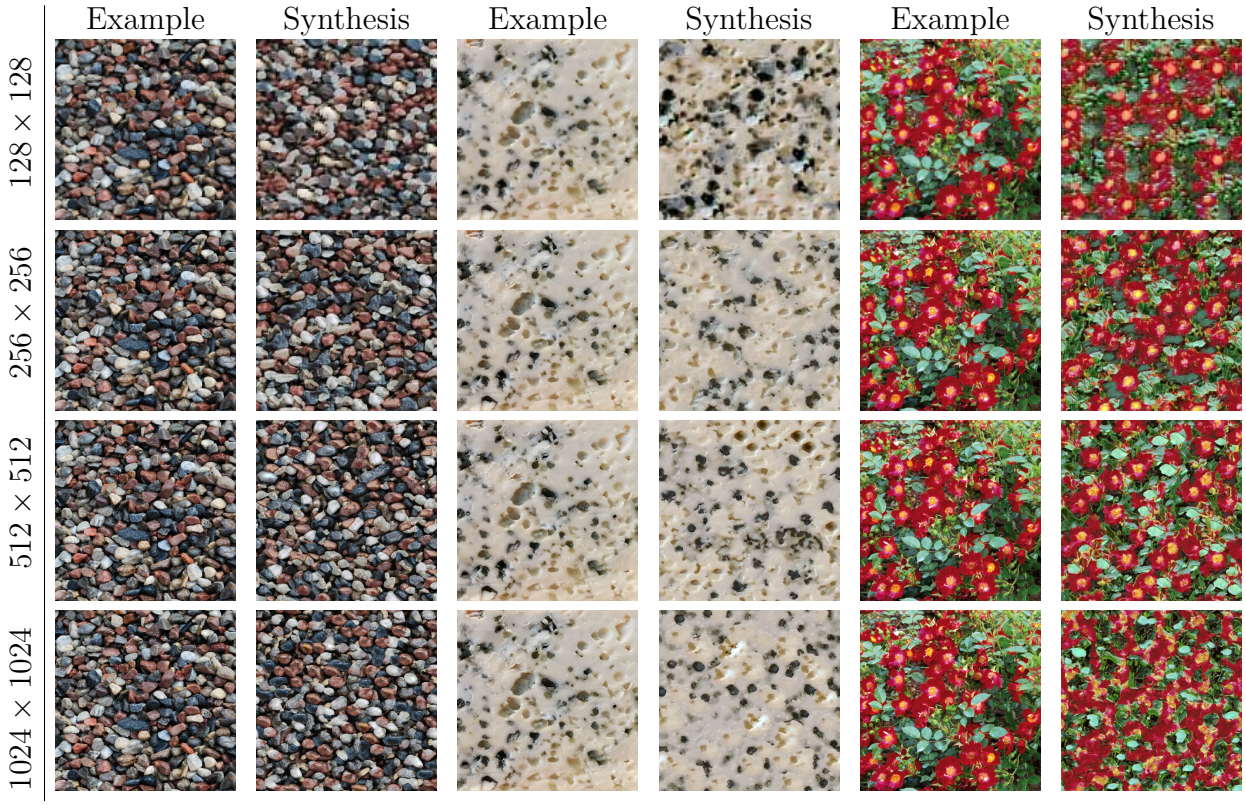



Figure 4.6 **Impact of size of training**. We trained one generator for each example at four resolutions (indicated in the left). During training, the generator produces samples of the same size as the example.

The desired appearance of the samples $v$ is specified in the form of a view dissimilarity term for each direction. The generated 3D 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 three 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 $\mathcal{L}$. It measures how appropriate the appearance of a solid sample is by comparing its 2D slices $v_{d,n}$ ($n^{\text{th}}$ slice in along the $d^{\text{th}}$ direction) to each corresponding example $u_d$. The comparison is carried out in the space of features from the descriptor network $\mathcal{D}$, based on VGG-19 similarly to previous 2D methods.

The training scheme, detailed in Section 4.4, 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 4.5 presents experiments and

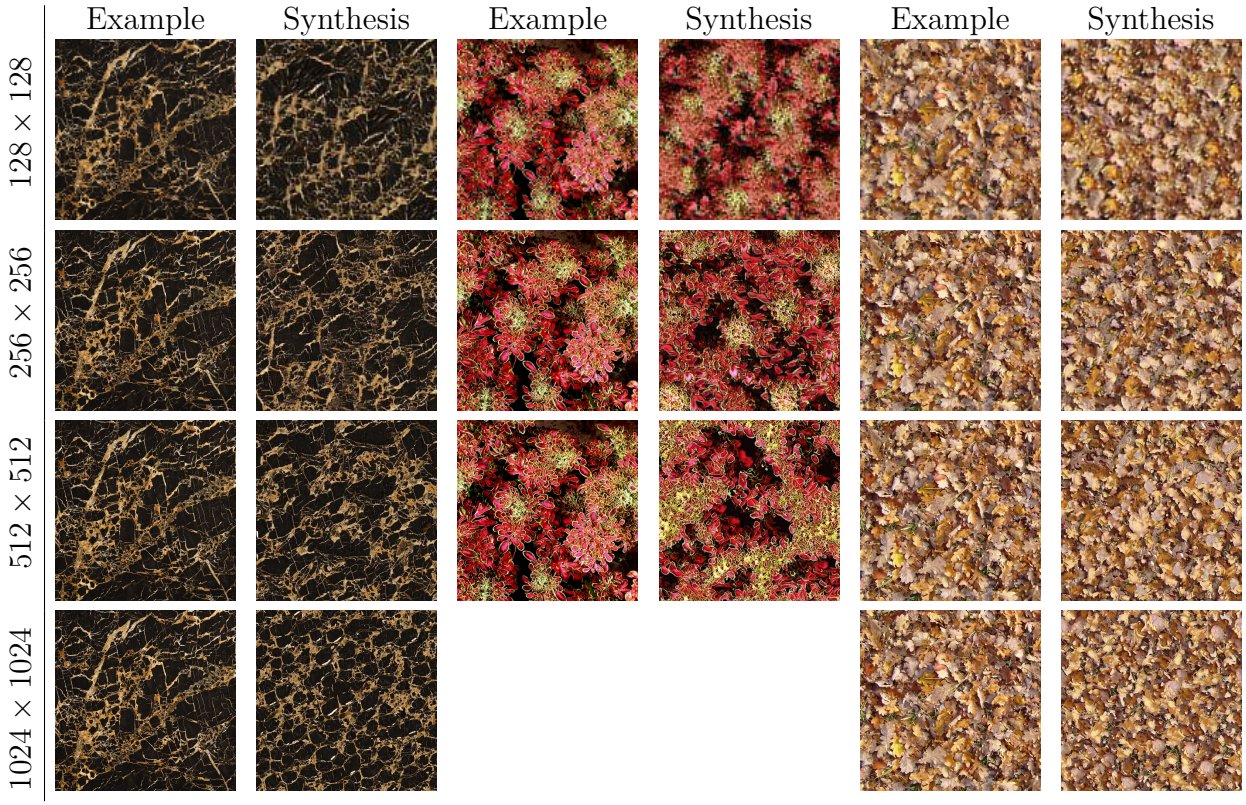



Figure 4.7 **Impact of size of training** We trained one generator for each example at four resolutions (indicated to the left). During training, the generator produces samples of the same size as the example.

comparative results. Finally, in Section 4.6 we discuss the current limitations of the model.

## 4.3 On-demand CNN Generator

Recall that on-demand evaluation is the capacity to independently generating coherent portions (i.e., a single voxel or a large block) of a texture sample. On-demand evaluation was enabled on previous patch-based 2D and solid texture methods [64, 69] but it has not been addressed in any texture method based on CNN. The architecture of the proposed CNN generator is summarized in Figure 4.11 and detailed in Section 4.3.1. 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. [8]'s model presented in Section 4.1, and which stands out for on-demand evaluation thanks to its small number of parameters and its local dependency between output and input. It uses a multi-scale approach, which has been successfully used in many computer vision applications, and in particular for texture synthesis [1, 3, 6, 28–30, 85].

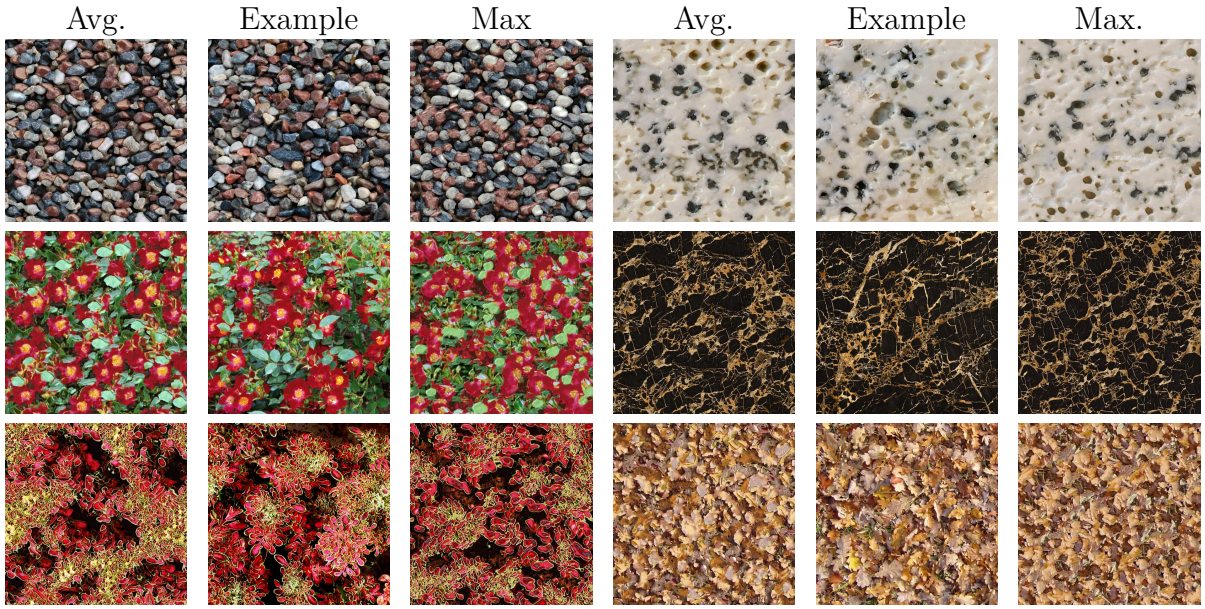



Figure 4.8 **Results using Max pooling vs Average pooling on the layers of VGG during training.**

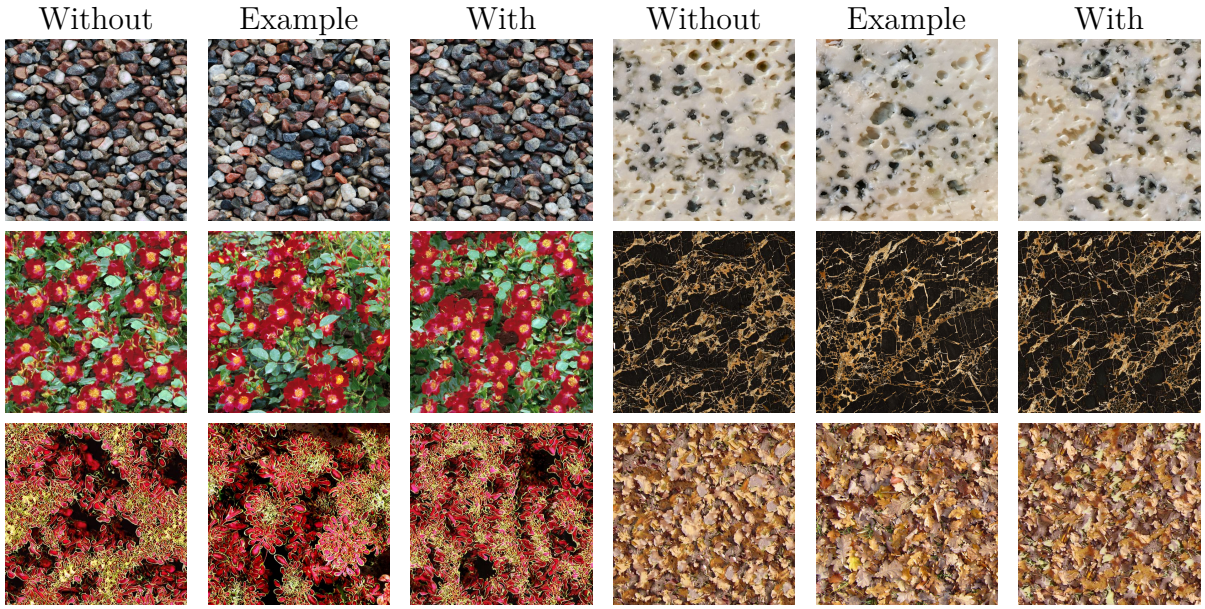



Figure 4.9 **Result of training with and without gradient normalization (using average pooling on VGG)**.

This fully convolutional generator allows the generation of on-demand box-shaped/rectangular volume textures of an arbitrary size (down to a single voxel) controlled by the size of the input. Formally, given an infinite noise input it represents an infinite texture model. A first step to achieve on-demand evaluation is to control the size of the generated sample.

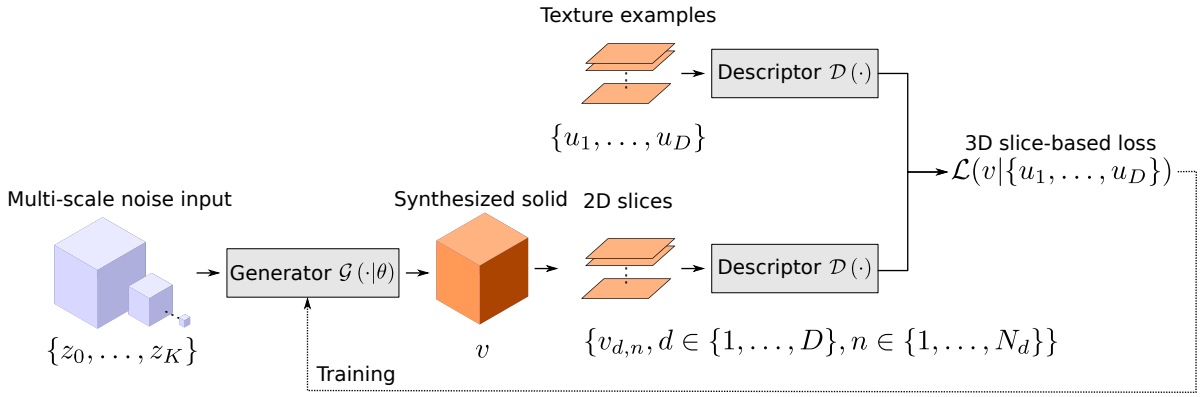



Figure 4.10 **Training framework for the proposed CNN Generator network $\mathcal{G}(\cdot|\theta)$ with parameters $\theta$.** The generator processes a multi-scale noise input $Z$ to produce a solid texture $v$. The loss $\mathcal{L}$ compares, for each direction $d$, the feature statistics induced by the example $u_d$ in the layers of the pre-trained Descriptor network $\mathcal{D}(\cdot)$ to those induced by a each slice of the set $\{v_{d,1}, \ldots, v_{d,N_d}\}$. The Training iteratively updates the parameters $\theta$ to reduce the loss. We show in Section 4.4 that we can perform training by only generating single-slice solids instead of full cubes.

To do so, we unfold the contribution of the values in the noise input to each value in the output of the generator. This dependency is described in Section 4.3.2. Then, on-demand voxel-wise generation is achieved thanks to the multi-scale shift compensation detailed in Section 4.3.3. The resulting generator is able to synthesize coherent and expandable portions of a theoretical infinite texture.

### 4.3.1 Architecture

The generator produces a solid texture $v = \mathcal{G}(Z|\theta)$ 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, i.e., $v$ has the same spatial dimension as $z_0$. The process of transforming the noise $Z$ into a solid texture $v$ is depicted in Figure 4.11. 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.

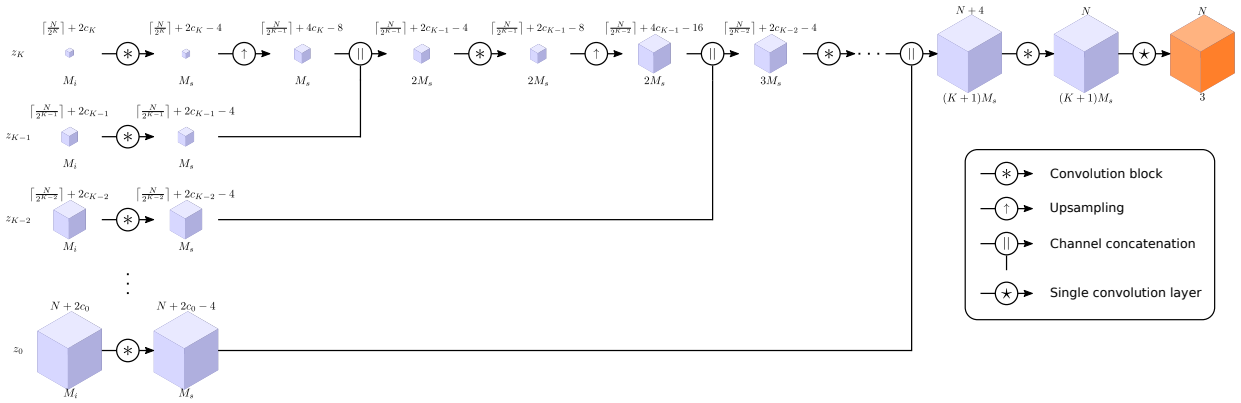



Figure 4.11 **Schematic of the Generator's architecture**. It processes a set of noise inputs $Z = \{z_0, \ldots, z_K\}$ at $K+1$ different scales using convolution operations and non-linear activations. The information at different scales is combined using upsampling and channel concatenation. $M_i$ indicates the number of input channels and $M_s$ controls the number of channels at intermediate layers. For simplicity we consider a cube shaped output with spatial size of $N_1 = N_2 = N_3 = N$. For each intermediate cube the spatial size is indicated above ($\lceil \cdot \rceil$ stands for the ceiling function) and the number of channels below.

**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 [8] and in order to enable on-demand evaluation (see Section 4.3.2), 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 four 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 eight times).

**Channel concatenation** This operation applies a batch normalization followed by the concatenation of 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 4.4.

### 4.3.2 Spatial Dependency

Forward evaluation of the generator is deterministic and local, i.e., each value in the output only depends on a small number of neighboring values in the multi-scale input. By handling the noise inputs correctly, we can feed the network separately with two *contiguous* portions of noise to synthesize textures that can be tiled seamlessly. Let us note that a perfect tiling between two different samples can only be achieved by using convolutions without padding. Current 2D CNN methods [8, 58] perform padded convolutions, not addressing on-demand evaluation capabilities. Therefore we discard the values on the borders of feature maps where the operation with the kernel cannot be carried out completely.

When synthesizing a sample, the generator is fed with an input that takes into account the neighboring dependency values. Those extra values are progressively processed and discarded in the convolutional layers: for an output of size $N_1 \times N_2 \times N_3$ the size of the input at the $k$-th scale has to be $(\frac{N_1}{2^k} + 2c_k) \times (\frac{N_2}{2^k} + 2c_k) \times (\frac{N_3}{2^k} + 2c_k)$ where $c_k$ denotes the additional values required due to the dependency. The size in any spatial dimension $N$ can be any positive integer (provided the memory is large enough for synthesizing the volume).

These additional values $c_k$ depend on the network architecture. In our case, thanks to the symmetry of the generator, the coefficients $c_k$ are the same along each spatial dimension. Each convolutional block requires additional support of two values on each side along each dimension and each upsampling cuts down the dependency by two (taking the smallest following integer when the result is fractional). At scale $k = 0$ there are two convolution blocks, therefore $c_0 = 4$. For subsequent scales $c_k = \lceil (c_{k-1} - 2)/2 \rceil + 4$ (where $\lceil \cdot \rceil$ stands for the ceiling function) except for the coarsest scale $K$ where there is only one convolution block and therefore $c_K = \lceil (c_{K-1} - 2)/2 \rceil + 2$. For example, in order to generate a single voxel, the spatial dimensions of the noise inputs must be $9^3$ for $z_0$ , $11^3$ for $z_1$, $13^3$ for $z_2$ to $z_4$, and $9^3$ for $z_5$, which totals 9380 random values.

### 4.3.3 Multi-scale Shift Compensation

On-demand generation is a standard issue in procedural synthesis [12]. The purpose is to generate consistently any part of an infinite texture model. It enables the generation of small texture blocks separately, whose localization depends on the geometry of the object to be texturized, instead of generating directly a full volume containing the object. For procedural noise, this is achieved using a reproducible Pseudo-Random Number Generator (PRNG) seeded with spatial coordinates.

In our setting, we enforce spatial consistency by generating the multi-scale noise inputs using

a xorshift PRNG algorithm [93] seeded with values depending on the volumetric coordinates, channel, and scale, similarly to [94]. Thus, our model only requires the set of reference 3D coordinates and the desired size to generate spatially coherent samples. Given a reference coordinate $n_0$ at the finest scale in any dimension, the corresponding coordinates at the $k$-th scale are computed as $n_k = \lfloor \frac{n_0}{2^k} \rfloor$ ($\lfloor \cdot \rfloor$ stands for the floor function). These corresponding noise coordinates need to be aligned in order to ensure the coherence between samples generated separately.

Feeding the generator with the precise set of coordinates of noise at each scale is only a first step to successfully synthesize compatible textures. Recall that the model is based on combinations of transformed noises at different scales (see Figure 4.11), therefore requiring special care regarding upsampling to preserve 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. In order to consider these compensation shifts, we make the generator network aware of the coordinate of the sample at hand. In our implementation the reference is on the vertex of the sample closest to the origin. Given the final reference coordinate $n_0$ of the voxel (in any spatial dimension), the generator deduces the set of shifts recursively from the following relation, $n_{k-1} = 2n_k + s_k$, where $n_k$ is the spatial reference 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 $k^{\text{th}}$ upsampling operation. At evaluation time, the generator sequentially applies the set of shifts before every corresponding concatenation operation.

## 4.4 Training

Here we detail our approach to obtain the parameters $\theta_u$ 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 2D examples. In Section 4.4.1 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 solid and examples. We conduct 2D comparisons similarly to the state-of-the-art methods, using the perceptual loss function [7, 25, 57].

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 even the batch processing impractical. Instead, we show in Section 4.4.2 that, thanks to the stationary properties of our generative model, we can carry out the training using batches of single slice solid samples.

### 4.4.1 3D Slice-Based Loss

For a color solid $v \in \mathbb{R}^{N_1 \times N_2 \times N_3 \times 3}$, we denote by $v_{d,n}$ the $n^{\text{th}}$ 2D slice of the solid $v$ orthogonal to the $d^{\text{th}}$ direction. Given a number $D \leq 3$ of slicing directions and the corresponding example images $\{u_1, \dots, u_D\}$, we propose the following slice based loss

$$\mathcal{L}(v|\{u_1, \dots, u_D\}) = \sum_{d=1}^{D} \frac{1}{N_d} \sum_{n=0}^{N_d-1} \mathcal{L}_2(v_{d,n}, u_d), \tag{4.4}$$

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

We use the 2D perceptual loss $\mathcal{L}_2$ from [7], which proved to be successful for training CNN [8, 25] for the task of image synthesis. We formally define the perceptual loss earlier in Section 4.1.1. It compares the Gram matrices of the VGG-19 feature maps of the synthesized and example images. The 2D loss between the example $u_d$ and a slice $v_{d,n}$ is then defined as

$$\mathcal{L}_2(v_{d,n}, u_d) = \sum_{l \in L} \frac{1}{(M^l)^2} \left\| G^l(v_{d,n}) - G^l(u_d) \right\|_F^2, \tag{4.5}$$

where $\| \cdot \|_F$ is the Frobenius norm and $G^l \in \mathbb{R}^{M^l \times M^l}$ is the Gram matrix from the feature maps at layer $l$. As mentioned before, the Gram matrices encode both first and second order informations of the feature maps distribution.

### 4.4.2 Single Slice Training

Formally, training the generator $\mathcal{G}(Z|\theta)$ with parameters $\theta$ corresponds to minimizing the expectation of the loss in Eq. (4.4) given the set of examples $\{u_1, \ldots, u_D\}$,

$$\theta_u \in \underset{\theta}{\operatorname{argmin}} \quad \mathbb{E}_Z \left[\mathcal{L}(\mathcal{G}(Z|\theta),\, \{u_1, \ldots, u_D\})\right], \tag{4.6}$$

where $Z$ is a multi-scale noise, independent, and identically distributed from a uniform distribution in the interval $[0, 1]$.

Note that each scale $z_0, \ldots, z_K$ of $Z$ is stationary. The generator on the other hand induces a non stationary behavior on the output due to upsampling operations. 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 $d^{\text{th}}$ axis, for any coordinate at the scale of the generated sample, which we can write $n = 2^K p + q$ with $q \in \{0, \ldots, 2^K - 1\}$ and $p \in \mathbb{N}$, the statistics of the slice $\mathcal{G}(Z, \theta)_{d,n}$ only depend on the value of $q$, therefore

$$\mathbb{E}_Z \left[\mathcal{L}_2(\mathcal{G}(Z, \theta)_{d,n}, u_d)\right] = \mathbb{E}_Z \left[\mathcal{L}_2(\mathcal{G}(Z, \theta)_{d,q}, u_d)\right]. \tag{4.7}$$

Assuming $N_d$ is a multiple of $2^K$, we have

$$\begin{aligned} \mathbb{E}_Z \left[\mathcal{L}(\mathcal{G}(Z, \theta),\, \{u_1, \ldots, u_D\})\right] &= \sum_{d=1}^{D} \frac{1}{N_d} \sum_{n=0}^{N_d - 1} \mathbb{E}_Z \left[\mathcal{L}_2(\mathcal{G}(Z, \theta)_{d,n}, u_d)\right] \\ &= \sum_{d=1}^{D} \frac{1}{2^K} \sum_{q=0}^{2^K - 1} \mathbb{E}_Z \left[\mathcal{L}_2(\mathcal{G}(Z, \theta)_{d,q}, u_d)\right]. \end{aligned} \tag{4.8}$$

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 Eq. (4.8).

Considering the slice $n = 2^K p + Q_d$ in the $d^{\text{th}}$ 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,

$$\mathbb{E}_{Z,Q_d}\left[\mathcal{L}_2(\mathcal{G}(Z,\theta)_{d,Q_d}, u_d)\right] = \frac{1}{2^K}\sum_{q=0}^{2^K-1}\mathbb{E}_Z\left[\mathcal{L}_2(\mathcal{G}(Z,\theta)_{d,q}, u_d)\right]. \tag{4.9}$$

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

$$\mathbb{E}_Z\left[\mathcal{L}(\mathcal{G}(Z,\theta),\, \{u_1, \ldots, u_D\})\right] \simeq \sum_{d=1}^{D}\mathbb{E}_{Z,Q_d}\left[\mathcal{L}_2(\mathcal{G}(Z,\theta)_{d,Q_d}, u_d)\right], \tag{4.10}$$

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 reduces the required amount of memory. In this setting we can use resolutions of up to 1024 values per dimension 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 $256 \times 256$ (limitation that derives mostly from computation time and that is hard to alleviate due to bottlenecks in the models).

## 4.5 Results

### 4.5.1 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 six, 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 is smaller than the value used in the 2D model in Section 4.1. This keeps the model light and, as we show here, does not affect performance. The configuration results in the last layer being quite narrow ($6M_s = 24$) and the whole network compact, with $\sim 8.5 \times 10^4$ parameters. We include a batch normalization operation after every convolution layer and before the concatenations. As mentioned in previous methods [8] we noticed that such a strategy helps stabilizing the training process.

**Descriptor network** Following Gatys et al. [7], we use a truncated VGG-19 [50] 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 [7,91]. We optimize the parameters of the generator network using Adam algorithm [92] with a learning rate of 0.1 during 3000 iterations. Figure 4.12 shows the value of the empirical estimation of $\mathcal{L}$ during the training of three of the examples shown below in Figure 4.15 (*histology, cheese,* and *granite*). We use batches of ten 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 three slicing directions, the training takes around one hour for a $128^2$ training resolution (i.e., size of the example(s) and generated slices), 3.5 hours for $256^2$, and 12.5 hours for $512^2$ using one GPU Nvidia GeForce GTX 1080 Ti.

**Synthesis** In order to synthesize large volumes of texture, it is more time efficient to choose the size of the building blocks in a way that best exploits parallel computation. Considering computation complexity alone, synthesizing large building blocks of texture at once is also more efficient given the spatial dependency (indicated by coefficients $c_k$) shared by neighboring voxels. In order to highlight the seamless tiling of on-demand generated blocks of texture, most of the samples shown in this work are built by assembling blocks of $32^3$ voxels. However, the generator is able to synthesize box-shaped/rectangular samples of any size, given enough memory is available. Figure 4.13 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 an Nvidia GeForce RTX 2080 GPU. However we can use larger 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. [64] it takes 220 milliseconds to synthesize a $64^3$ volume and 1.7 seconds to synthesize a $128^3$ volume.

### 4.5.2 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 4.14 exhibits the application of textures generated with our model to texturize to 3D mesh models. In this case we generate a solid texture with a fixed size and load it in OpenGL as a regular 3D texture, interpolated with bilinear filtering.

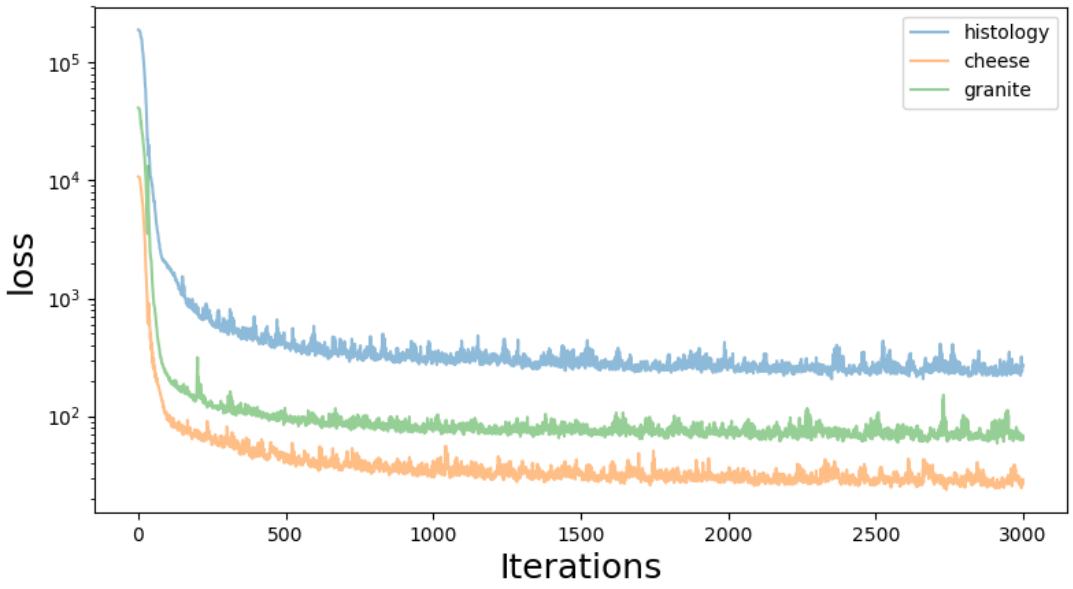



Figure 4.12 **Value of the 3D empirical loss, Eq.** (4.4)**, during the training of the generator for the textures *histology*, *cheese*, and *granite* of Figure 4.15.**

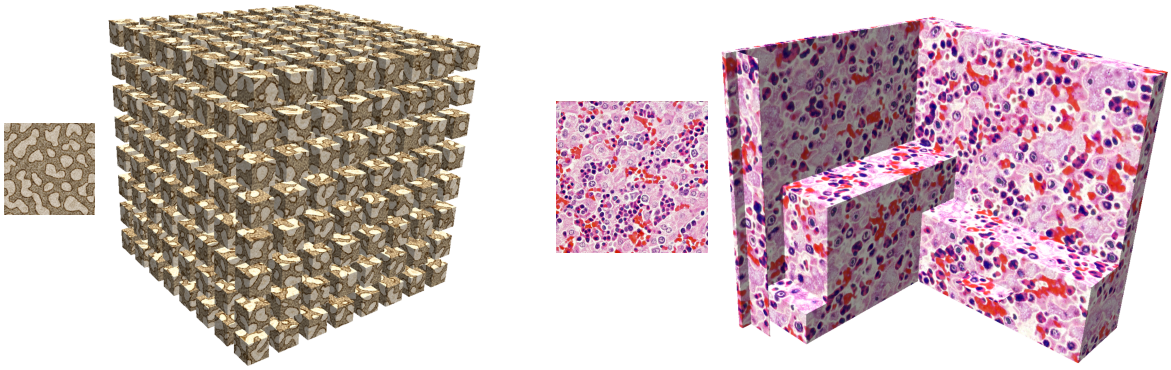


Figure 4.13 **On-demand tiling consistency**. Left: example texture of $128^2$ pixels and 512 cubes of generated texture that tile perfectly, we add a gap between the cubes to show the continuity of the patterns. Right: example texture of $512^2$ pixels with an assembled set of blocks of different sizes generated on-demand. Note that it is possible to generate blocks of size 1 in any direction.

Solid textures avoid the need for surface parametrization and can be used on complex surfaces without creating artifacts.

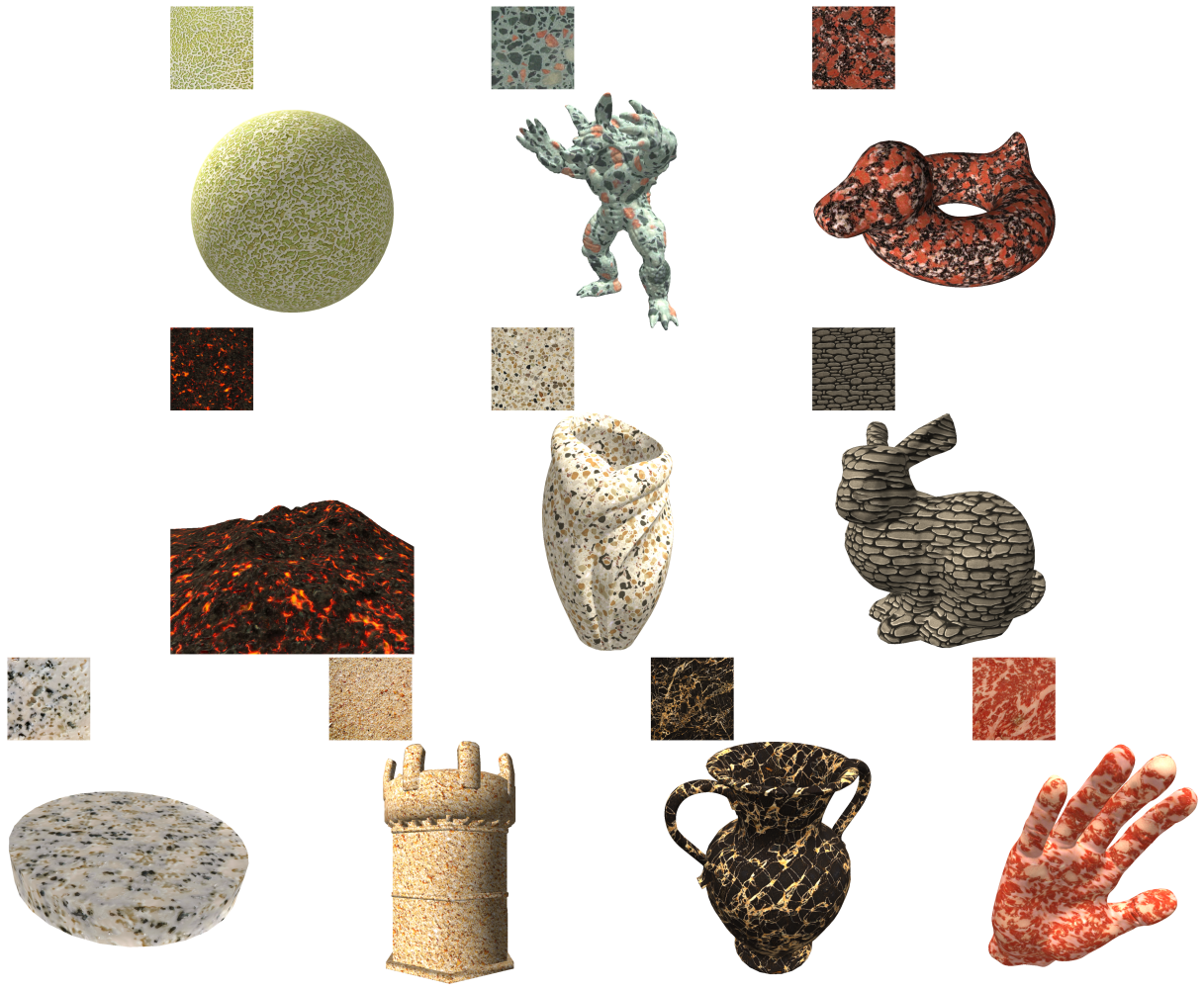


Figure 4.14 **Texture mapping on 3D mesh models**. The example texture used to train the generator is shown in the upper left corner of each object. When using solid textures the mapping does not require parametrization as they are defined in the whole 3D space. This prevents any mapping induced artifacts. *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.*

**Single example setting** Figures 4.15 and 4.16 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 $512^3$ voxels built by assembling blocks of $32^3$ voxels generated using on-demand evaluation. The second column is the example image of size $512^2$ pixels, columns 3-5 show

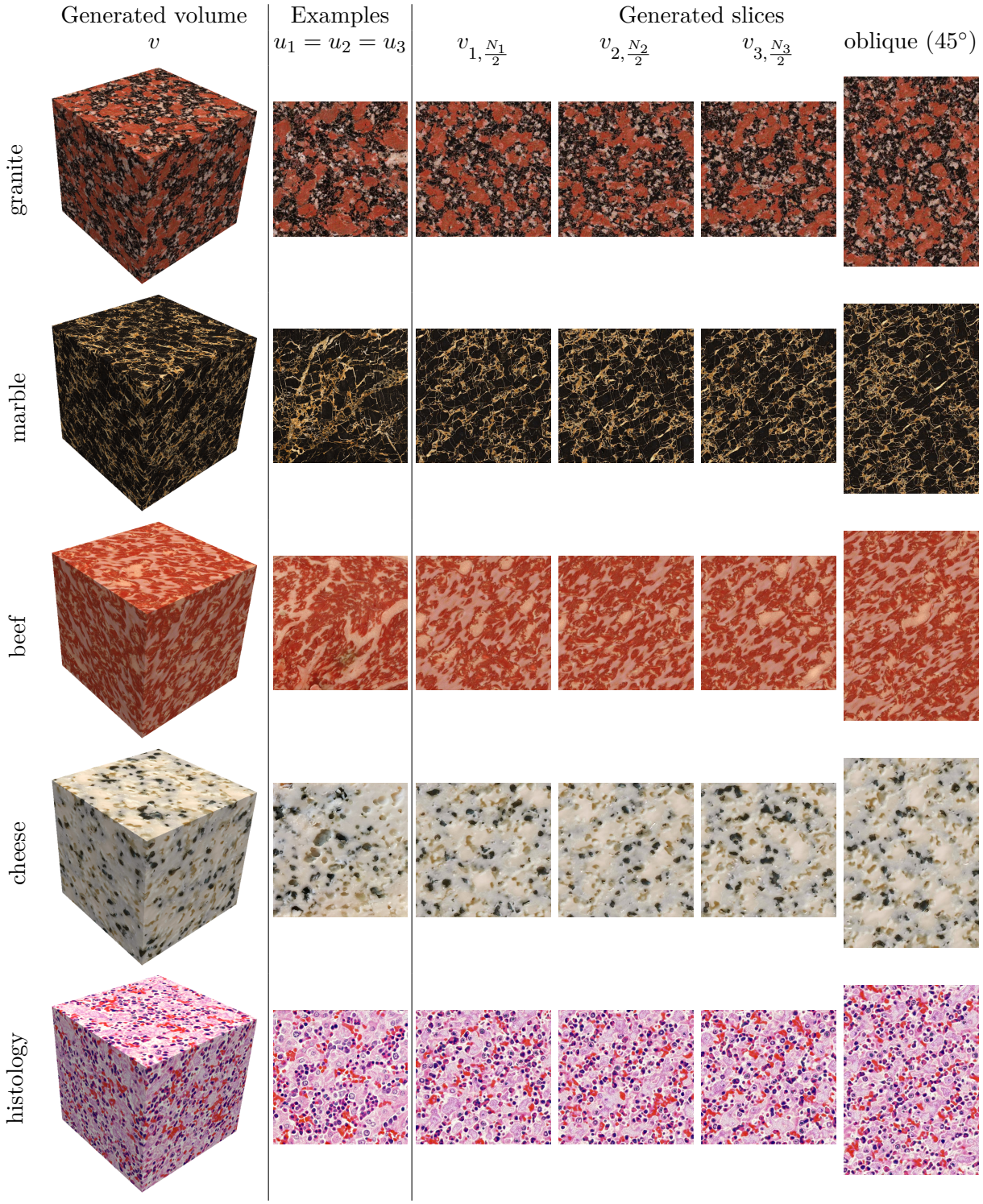



Figure 4.15 **Synthesis of isotropic textures**. We train the generator network using the example in the second column of size $512^2$ along $D = 3$ directions. The cubes on the first column are generated samples of size $512^3$ built by assembling blocks of $32^3$ voxels generated using on-demand evaluation. 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° angle. The trained models successfully reproduce the visual features of the example in 3D.

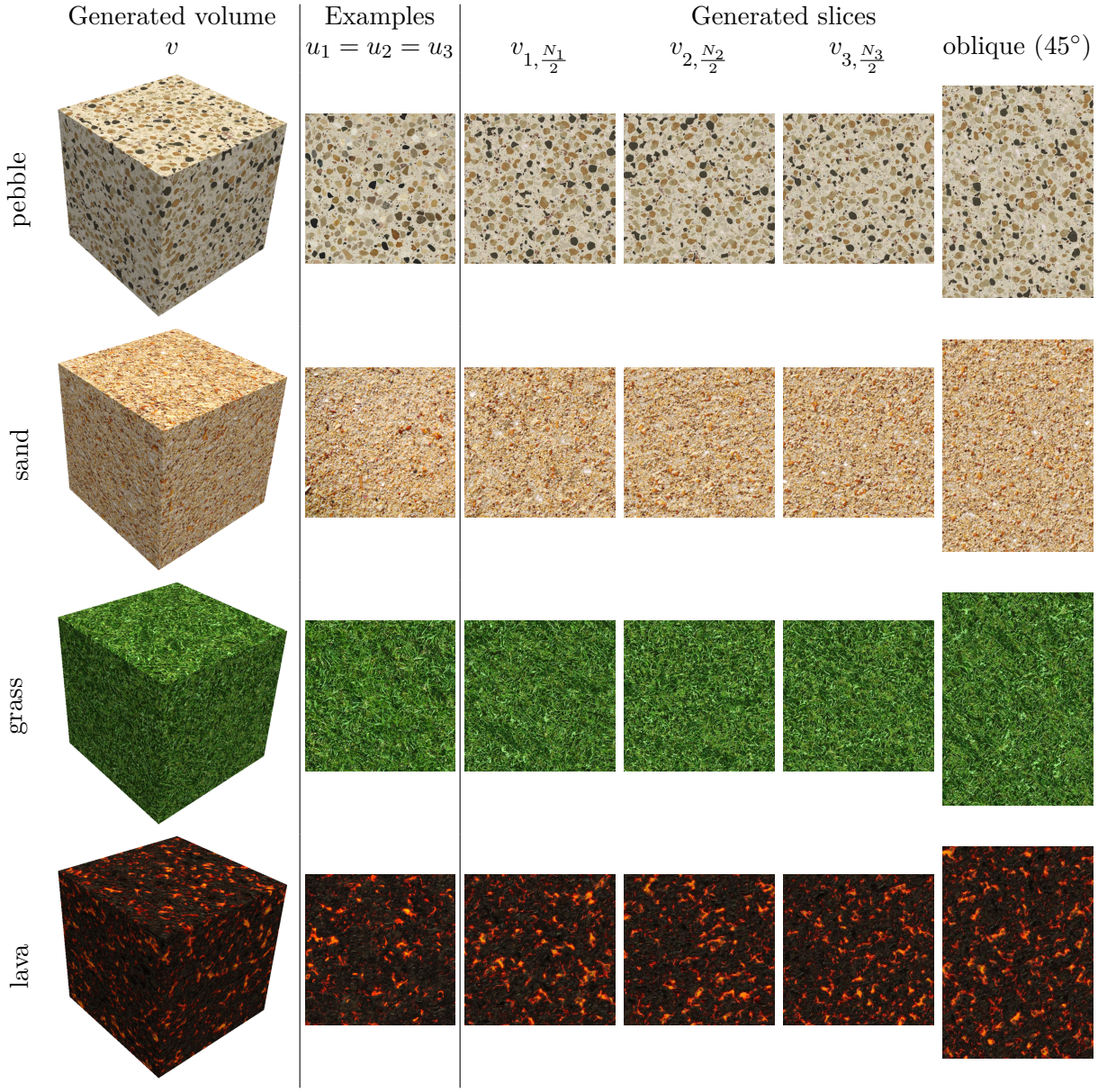



Figure 4.16 **Synthesis of isotropic textures (same presentation as in Figure 4.15)**. While generally satisfactory, the examples in the first and third rows have a slightly degraded quality. In the first row the features have more rounded shapes than in the example and in the third row we observe local high frequency artifacts.

the middle slices of the generated cube across the three considered directions and the last column shows a slice extracted in an oblique direction at 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. Regarding the visual quality, notice that the model successfully reproduces the patterns' structure while also capturing the richness of colors and variations. The quality of the slices is comparable to that of the state-of-the-art 2D methods [7, 8, 57, 58], which is striking as solid texture synthesis is a more constrained problem. However, such successful extrapolation to 3D might not be possible for all textures. Figure 4.17 shows a case where the slices of the synthesized solid contain patterns not present in the example texture. This is related to the question of the existence of a solution discussed in the next paragraph.

**Existence of a solution** The example texture used in Figure 4.17 is isotropic (arrangement of red shapes having green spot inside), but the volumetric material it depicts is not (the green stem being outside the pepper). Training the generator using three orthogonal directions assumes 3D isotropy, and thus, the outcome is a solid texture where the patterns are isotropic through the whole volume. This creates some new patterns in the slices, which makes them somewhat different from the example (red shapes without a green spot inside). Actually, the generated volume just does not make sense physically, as one always obtains full peppers after slicing them. This example shows that for a given texture example $u$ and a number of directions $D > 1$ it is possible that a corresponding 3D texture 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 is one of the limitations of this slice based formulation used by many methods in the literature. It is briefly mentioned in [17, 64] and we extend the discussion here. Let us consider for instance the isotropic example shown in Figure 4.18, where the input image contains only discs of roughly the same size (i.e., a few pixels in 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 natural issue 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 during optimization. In global optimization methods [17, 46], where patches from the example are sequentially copied

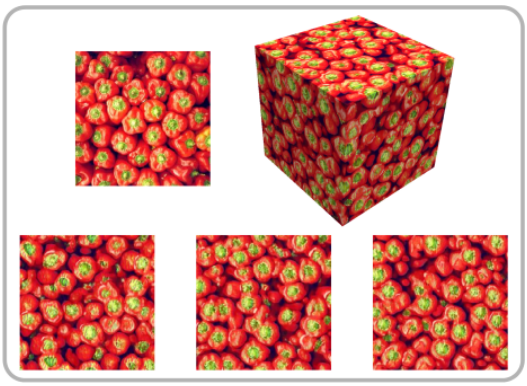


Figure 4.17 **2D to 3D isotropic patterns.** The example texture (top-left) depicts a pattern that is approximately isotropic in 2D, but the material it depicts is not. Training the generator using the example along three orthogonal directions results in solid textures that are isotropic in the three dimensions (top-right). Here, the red and green patterns vary isotropically in the third dimension, this creates a bigger variation on their size and makes some slices contain red patterns that lack the green spot. This is a case where the slices of the solid texture (bottom) cannot match exactly the patterns of the 2D example, thus, not complying with the example in the way a 2D algorithm would.

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 4.20, where the 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 also contrasts with common statistical texture synthesis approaches that are rather based on constrained optimization to guarantee statistical matching, for instance by projecting patches [95], histogram matching [30], or moment matching [1].

**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 4.19, 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

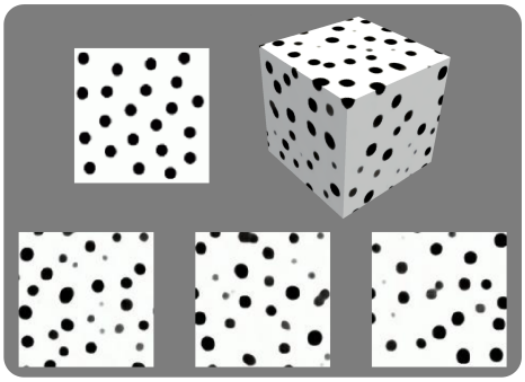


Figure 4.18 **Illustration of a solid texture whose cross sections cannot comply with the example along three directions**. Given a 2D example formed by discs of a fixed diameter (upper left) a direct isotropic 3D extrapolation would be a cube formed by spheres of the same diameter. Slicing that cube would result in images with discs of different diameters. The cube in the upper right is generated after training our network with the 2D example along the three orthogonal axes. The bottom row shows cross-sections along the axes, all of them present discs of varying diameters thus failing to look like the example.

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 4.19 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 $v_{1,\frac{N_1}{2}}$), 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 4.20, 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. Observe that the value of the loss at the end of the training gives a hint on which configuration *works* better. This can be exploited for automatically find the best training configuration for a given set of examples.

These results bring some light to the scope of the slice-based formulation for solid texture synthesis using 2D examples. This formulation is best suited for textures depicting 3D isotropic materials, for which we obtain an agreement with the example's patterns comparable with 2D state-of-the-art methods. For most anisotropic textures we can usually obtain high

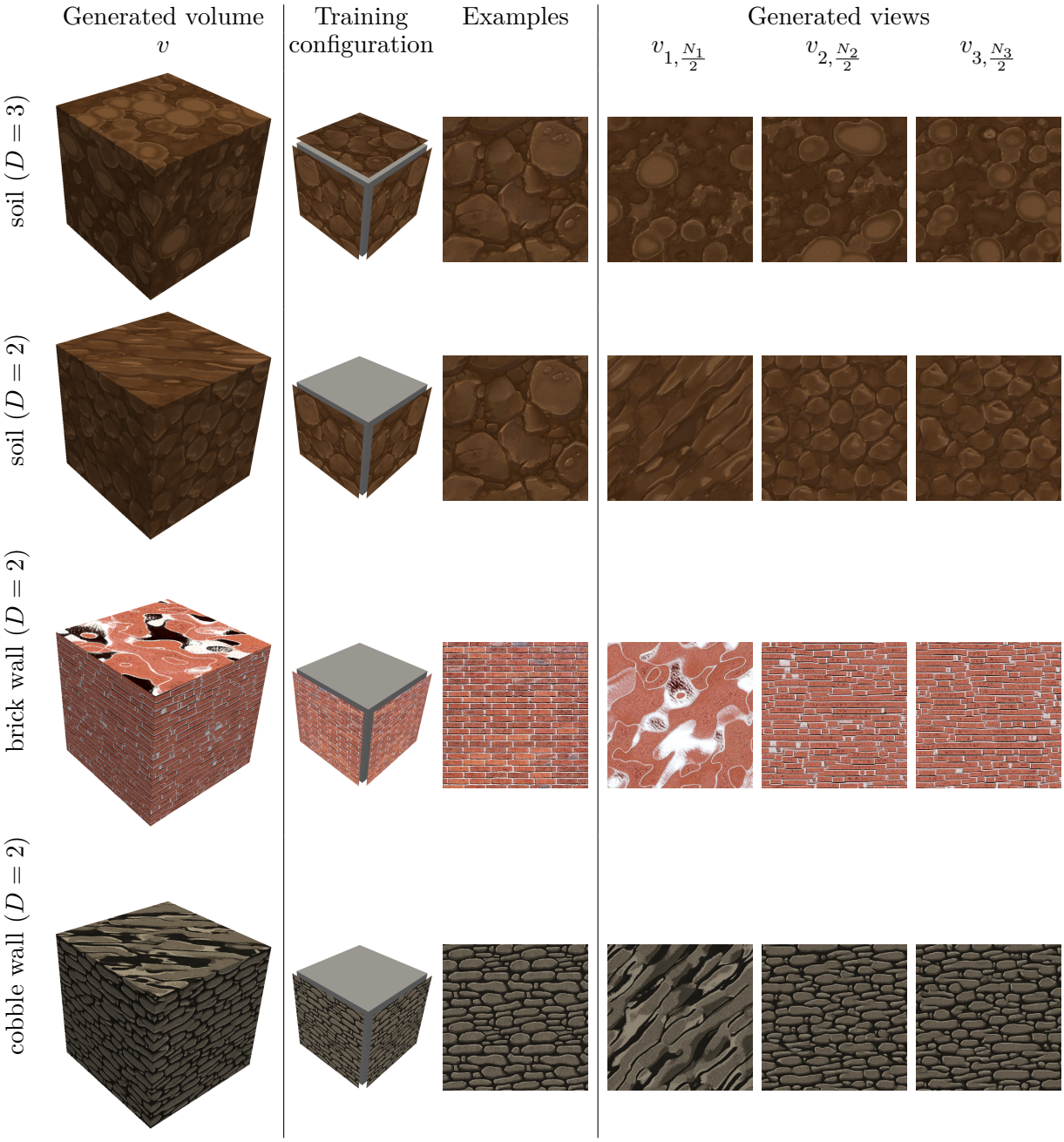



Figure 4.19 **Training the generator using two or three directions for anisotropic textures**. The first column shows generated samples of size $512^3$ built by assembling blocks of $32^3$ voxels generated using on-demand evaluation. The second column illustrates the training configuration, i.e., which 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.

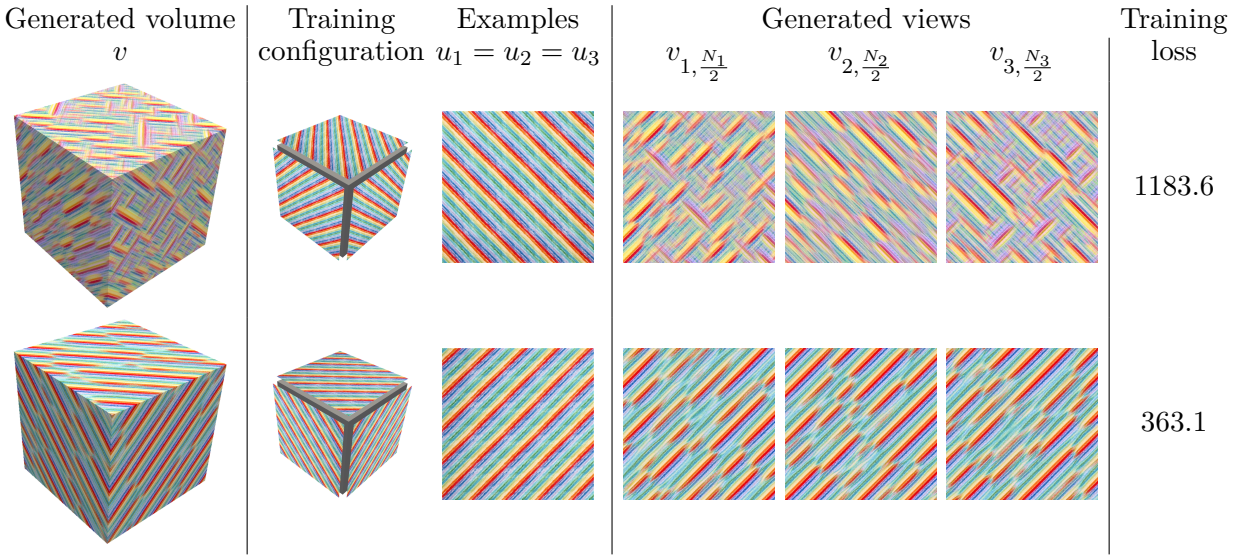



Figure 4.20 **Importance of the compatibility of examples**. In this experiment, two generators are trained with the same image along three directions, but for two different configurations. The first column shows generated samples of size $512^3$ built by assembling blocks of $32^3$ voxels generated 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. Finally, the rightmost column gives the empirical loss value at the last iteration. 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. The lower value of the training loss for this configuration reflects the better reproduction of the patterns in the example.

quality results by considering only two directions. Finally, for textures with patterns that are only isotropic in 2D, using more than one direction inevitably creates new patterns.

**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, as demonstrated in several results in Chapter 3, many methods in the literature for 2D texture synthesis generate images that are local copies of the input image [3,6], which strongly limits the diversity. As reported in Gutierrez et al. [95], the (trivial and undesired) optimal solution of methods based on patch optimization is the input image itself. For most of these methods though, 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 [57, 58] 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. This might be enabled by a large number of parameters in the network.

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 [96]. However, the compactness of 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 4.12). This is consistent with the results of Ulyanov et al. [8], 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 (0.0025% of values in [58] and 0.13% in [57]) and which is easier to ignore, for example by setting the weights in the first layers to zero. On the contrary, our generative network receives an input that accounts for roughly 1.14 times the size of the output. Figure 4.21 demonstrates the capacity of our model to generate diverse samples from a single trained generator. It shows three generated solid textures along with their middle slices $v_{d,\frac{N}{2}}$. To facilitate the comparison, it includes an *a posteriori* correspondence map which highlights spatial similarity by forming smooth regions. In all cases we obtain noisy maps which means that the slices do not repeat arrangements of patterns or colors.

**Multiple examples setting** As already discussed in the work of Kopf et al. [17] 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 [17, 64]. 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 4.20. Figure 4.22 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. When the colors in the example do not match, the network seems to either mix or average them in the synthesized sample. Although the

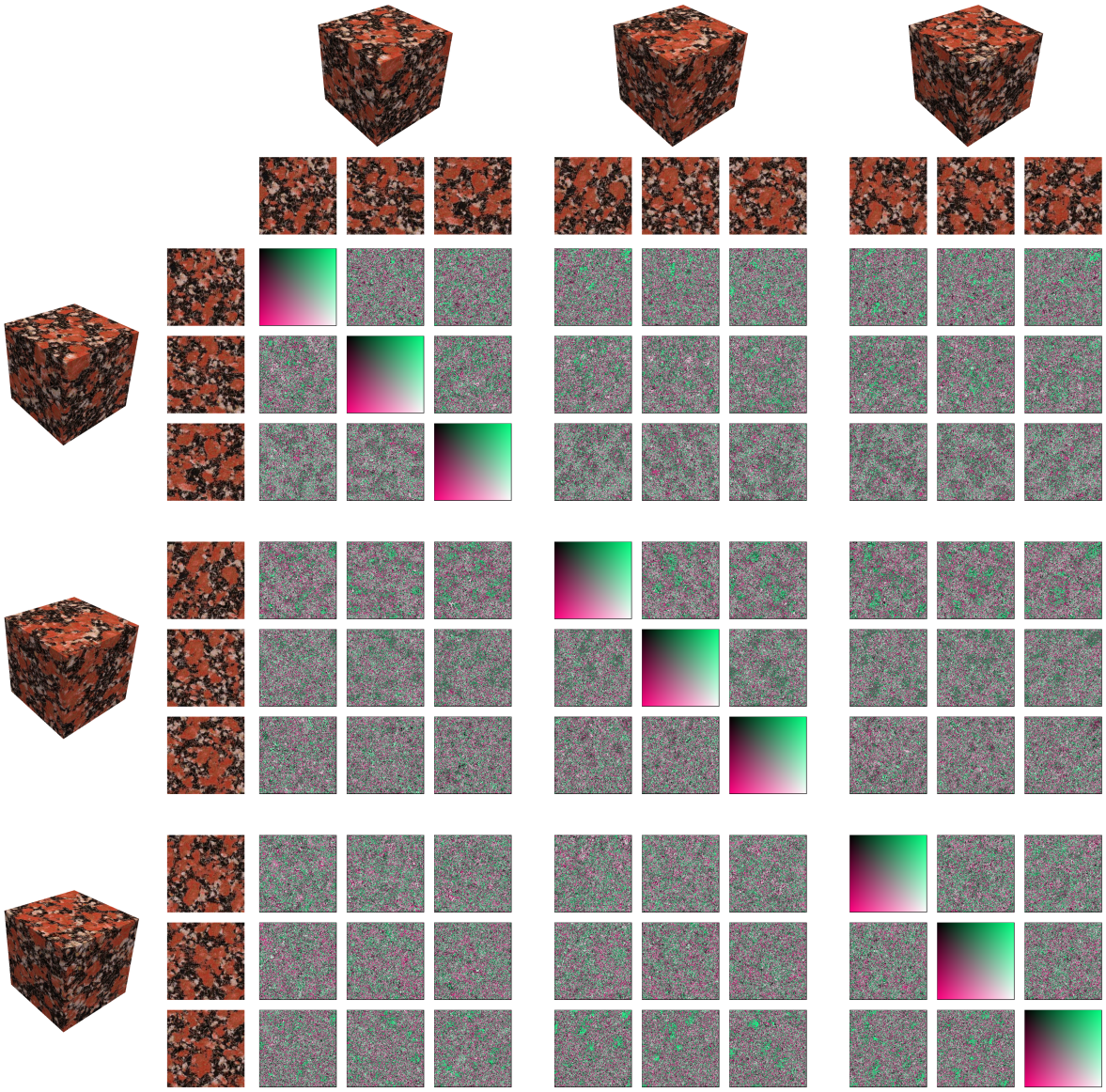


Figure 4.21 **Diversity among generated samples**. We compare the middle slices along three axes of three generated textures of $256^3$ voxels. The comparison consists in finding the pixel with the most similar neighborhood (size $4^2$) in the other image and constructing a correspondence map given its coordinates. The smooth result in the diagonal occurs when comparing a slice to itself. The stochasticity in the rest of results means that the compared slices do not share a similar arrangement of patterns and colors.

patterns are not perfectly reproduced, the 3D structure is coherent and close to the examples.

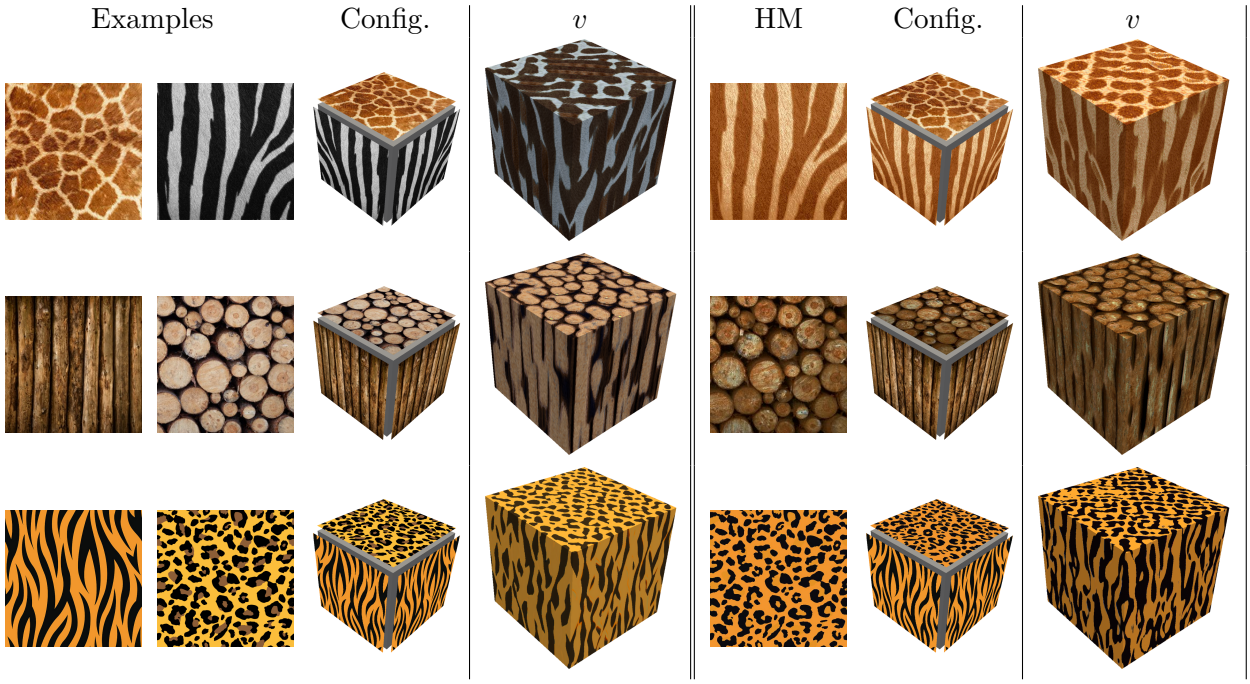



Figure 4.22 **Anisotropic texture synthesis using two examples**. The first three columns show the examples and the training configuration (i.e., how images are oriented for each view) used that lead to the result in the fourth column ($v$). In the last three columns we experiment with preprocessing one of the examples to match the colors of the other (by performing a histogram matching independently on each color channel). The example in the HM column is the one preprocessed and the last column again shows the resulting solid texture. We observe favorable results particularly when the colors of both examples are close.

### 4.5.3 Comparison with the State of the Art

The results in Figures 4.15 and 4.16 prove the capability of the proposed model to synthesize photo-realistic textures. This is an important improvement with respect to classical on-demand methods based on Gabor or local random-phase (LRP) noise [97]. High resolution examples are important in order to obtain more detailed textures. In previous high quality methods [17,46] the computation times increase substantially with the size of the example, so examples were limited to $256^2$ pixels. Furthermore, the empirical histogram matching steps create a bottleneck for parallel computation. Our method takes a step forward by allowing higher resolution example textures, depending only on the memory of the GPU used.

We compare the visual quality of our results against the two existing methods that seem to produce the best results: Kopf et al. [17] and Chen et al. [46]. Figure 4.23 shows some samples (obtained from the respective articles or websites) side by side with results using our method. The most salient advantage of our method is the ability to better capture

high frequency information, making the structures in the samples sharper and more photo-realistic. Considering voxels' statistics, i.e., capturing the richness of the example, both our method and that of Chen et al. [46] seem to obtain a better result than the method of Kopf et al. [17]. The examples used in Figure 4.23 have resolutions of either $128^2$ or $256^2$ pixels. We observe that the visual quality of the textures generated with our method deteriorate when using small examples. This can be due to the discriminator network which is pre-trained using larger images (244 pixel for VGG-19).

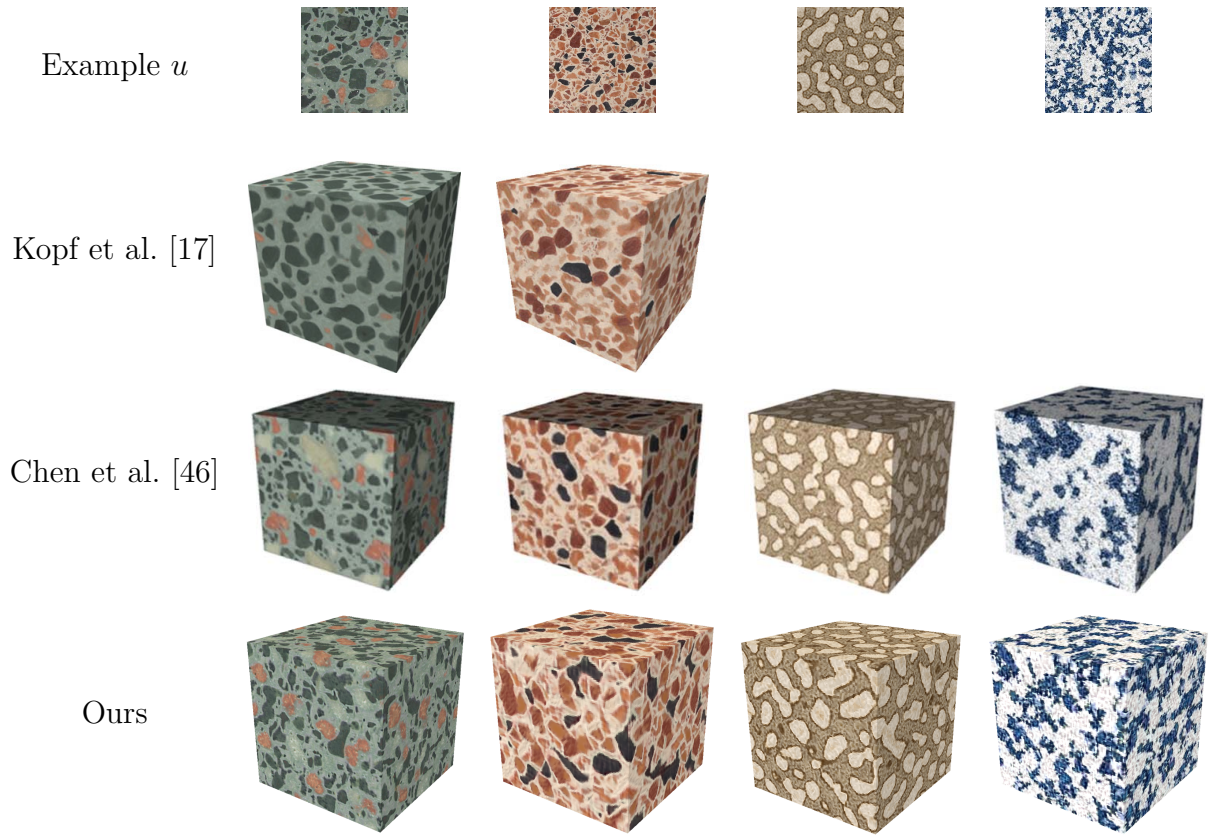



Figure 4.23 **Comparison with the existing methods that produce the best visual quality**. The last row shows the results using the proposed method. Our method is better at reproducing the statistics of the example compared to Kopf et al.'s [17] method and is better at capturing high frequencies compared to both methods.

We do not consider the method of Dong et al. [64] for a visual quality comparison as their pre-computation of candidates limits the richness of information, which yields lower quality results. However, thanks to the on-demand evaluation, their 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. Another advantage of our method is that the computation time for synthesizing new textures with our generator does not depend

on the resolution of the example used during training.

In Figure 4.24, we show some of our results used for texturing a complex surface and we compare them to the results of Kopf et al. [17]. Here the higher frequencies successfully reproduced with our method cause a more realistic impression.

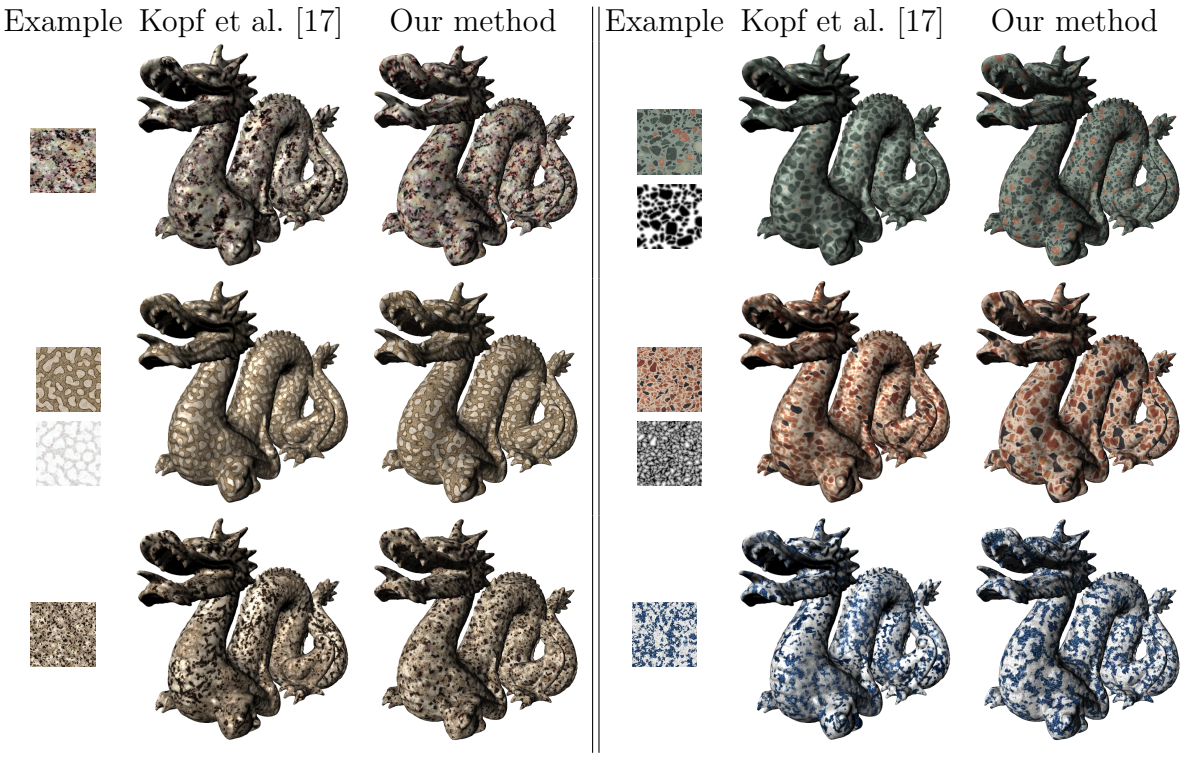



Figure 4.24 **Comparison of our approach with Kopf et al. [17].** Both methods generate a solid texture that is then simply interpolated and intersected with a surface mesh (without parametrization as required for texture mapping). The first column shows the example texture. The second column shows results from [17], 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 $640^3$ and $512^3$ for the rest.

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 $\sim 8.5 \times 10^4$ parameters to store) stands close to the memory footprint of a patch-based approach working with a $170^2$ color pixels input (i.e., $8.67 \times 10^4$ parameters if all the patches are used).

## 4.6 Discussion

**Long distance correlation** As it can be observed in the brick wall texture on Figure 4.19 and in the diagonal texture in Figure 4.20 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 4.19 where the objects size in the synthesized samples do not match the one in the example, again due to the overlooked long distance correlation. One possible explanation comes from the fixed receptive field of VGG as descriptor network. It is likely that it only sees some local patterns which results in breaking long patterns into pieces. A possible solution could be to use more scales in the generator network, similarly to use larger patches in patch-based methods. Another possible improvement to explore is to explicitly construct our 2D loss $\mathcal{L}_2$ incorporating those long distance correlations as in [55, 56].

**Constrained directions** We observed that training the generator with two instead of three constrained directions results in unsatisfying textures along the unconsidered direction, while improving visual quality along the two constrained directions for anisotropic textures (see Figure 4.19). 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 4.16 the synthesized sample presents over-simplified versions of the example's features. 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 [25] where a total variation term is used to try to mitigate the artifact.

**Non-stationary textures** The perceptual loss in Eq. (4.5) stands out for traditional texture synthesis, where the examples are stationary. An interesting problem is to consider non-stationary textures, such as the recent method of Zhou et al. [61], which uses an auto-encoder to extend the example texture twice. This problem is specifically challenging in our setting in absence of any 3D examples of such a texture.

**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.

## 4.7 Conclusion

The main goal of this chapter was to address the problem of example based solid texture synthesis by the means of a convolutional neural network. First, we presented a simple and compact generative network capable of synthesizing portions of infinitely extendable solid texture. The parameters of this 3D generator are stochastically optimized using a pre-trained 2D descriptor network and a slice-based 3D objective function. The complete framework is efficient both during training and at evaluation time. The training can be performed at high resolution, and textures of arbitrary size can be synthesized on-demand. This method is capable of achieving high quality results on a wide set of textures. We showed the outcome on textures with varying levels of structures and on isotropic and anisotropic arrangements. We demonstrate 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 function.

The second aim of this study was to achieve on-demand synthesis for which, to the best of our knowledge, no other method based on neural networks is capable of. 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. The proposed techniques during training and evaluation can be extended to any fully convolutional generative network. 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.

The proposed model presented in this chapter was published in the Computer Graphics Forum (CGF) journal in November 2019.

# CHAPTER 5 CONCLUSION

The main goal of example-based texture synthesis is to simplify the creation of texture images. In this thesis, we addressed two settings of computer graphics where texture synthesis can be particularly useful: the creation of new samples of a given 2D texture and the creation of solid textures.

In the 2D case, we aim at creating new samples of arbitrary sizes that reproduce the visual characteristics of an example image. Previous works hint that, for a 2D texture synthesis patch-based approach, synthesized textures using uniformly the patches from the example present better visual quality. Most of those methods, however, rely on empirical mechanisms that attempt to encourage the even utilization of the information in the example without offering any guarantee. This results in the possibility of introducing imperfections in the synthesized texture, often in the form of constant or blurry regions.

Example-based solid texture synthesis usually refers to using 2D examples to define the desired visual characteristics of the cross-sections of the synthesized solid texture. This is because in contrast to 2D images, 3D color information is difficult to obtain. Solid texture synthesis from 2D examples requires additional assumptions about the way the desired patterns propagate in the three dimensions. Most methods focus on generating isotropic solid textures by using a single example that defines the characteristics of the cross-sections along three orthogonal axes. Creating anisotropic solid textures requires to use different examples for different axis and/or to use two axes instead of three. Anisotropic solid textures have been addressed in a smaller extent compared to the isotropic kind, and the quality of the results is usually inferior.

Example-based solid textures have considerable potential for 3D graphics applications. Solid textures are already widely used in the procedural form because procedural textures integrate easily in the rendering pipeline. The downsides of procedural textures are: first, that each texture needs to be created by a skilled artist, and second, that they are limited to mainly stochastic textures with little structured patterns.

On the other hand, the use of solid textures from 2D examples algorithms is limited by memory. This is because most solid texture synthesis methods do not have the capability to synthesize only the texture at the required points without calling for a full block of texture that has a large memory footprint.

## 5.1 Summary of Work

In Chapter 3 we proposed a method for 2D texture synthesis based on patch statistics. Similarly to previous methods we formulate texture synthesis as an optimization problem. The core and main innovation of the proposed model is the optimal assignment of patches replacing the widely used nearest-neighbor model.

The proposed optimal assignment model guarantees that all the patches of the example texture are evenly used to compose the synthesized sample. Because of the two-step optimization strategy we employed, after the patch aggregation step, the new patch distribution might not perfectly match the one of the example. However, our experiments showed that the resulting patch distribution closely matches the example texture distribution.

Patch-based methods impose spatial coherence via the overlap between patches. Using an optimal assignment assumes that a perfect match is not required for promoting spatial coherence. Our experiments confirm this assumption as the synthesized samples present overall high visual quality.

This model led to a publication in the proceedings of the international conference on Scale Space and Variational Methods in Computer Vision (SSVM) in 2017.

Our experiments also revealed an interesting impact of the regularity of the patch subsampling on the size of the copied regions in the results. We encountered that by using a random subsample of patches, instead of a regularly spaced one, we effectively reduce the amount of verbatim copy that composes the new sample. This finding could extrapolate to other patch-based methods in texture synthesis and other applications where it is desired to minimize the amount of verbatim copy.

Additionally to our optimal assignment model, we presented two variations with softer statistical constraints. In practice, both methods replace the assignment phase of the two-step optimization algorithm. The relaxed assignment model enables an implicit control over the assignment from nearest neighbor to the optimal assignment. Our experiments verified this desired behavior. The near-optimal assignment model aims at reducing computation time by approximating the optimal assignment between patches. It exploits the properties of the Auction algorithm that provides assignments closer to optimality at each call. The visual results show additional evidence that the nearest-neighbor matching is not required for maintaining spatial coherence. Another benefit of this approach is that it can be partially parallelized in a GPU.

In the context of solid textures, we concluded that the assignment model would not be practical since the global nature of the formulation does not allow for the on-demand capabilities

thus limiting the application.

In Chapter 4 we proposed a method for example-based solid texture synthesis based on convolutional neural networks and capable of on-demand evaluation. We designed a solid texture generator network, whose pure-convolutional nature makes it possible to generate samples on-demand. We use the pre-trained VGG-19 network to extract a meaningful representation for textures. It facilitates the evaluation of the similarity of the synthesized sample with the example texture, using the perceptual loss, and makes it possible to train the generator network.

We showed that the generator network can be trained while generating only one-voxel-thick oriented textures. This single slice training makes it possible to train the generator with a low memory footprint, even at high resolution.

In order to maintain spatial coherence, we consider an infinite input generated from a random number generator. The proposed generator network uses the reference coordinate of the piece of texture at hand to enforce spatial coherence with contiguous samples. In order to accomplish this, the generator applies a set of shifts that compensate the up-sampling's non-stationary behavior at the different layers.

We tested the proposed model in different settings. The most common configuration is for isotropic textures, where the same example texture is used along three orthogonal axes. In this setting we obtain high quality solid textures with cross-sections comparable to the state-of-the-art 2D methods.

The capability of the model to infer plausible 3D structures from 2D examples allowed us to evaluate the limitations of the example-based solid texture formulation. We obtain plausible extrapolations of 2D textures that do not necessarily comply with the example's patterns.

The model also produced good results for the anisotropic setting where a single example is used for two orthogonal axes. However, as shown in previous methods, results are less satisfactory when using different examples along different axes. This setting requires a more complex interpolation for mixing the patterns of two different textures.

This work was published in the Computer Graphics Forum (CGF) journal in November 2019.

## 5.2 Limitations

As it is usual in the context of image processing, we conceived and tested the proposed methods with stationary textures in mind. As mentioned in Chapter 2, synthesis of non-stationary textures has been studied recently within GAN frameworks. We believe GANs

might be a better approach for these kinds of images since *the perceptual loss* intrinsically assumes stationarity.

In the context of solid texture synthesis, we focused on single texture synthesis, that is, we train one instance of the generator network for a given configuration of example textures. Contrary to methods that are capable of multi-texture synthesis, this allowed us to maintain a light architecture with a capability of integration in conventional graphics pipeline.

We performed visual quality evaluation subjectively by human observation mainly carried out by ourselves. This is a usual practice in the context of example-based texture synthesis. For methods that reproduce well the example texture, evaluation focuses on looking for common problems such as: lack of structure in the patterns, blurry regions, and high repetition. This undoubtedly limits the validity of the quality assessment, however, as explained in Chapter 2, more objective evaluation of texture synthesis is a complex problem on its own and still an ongoing research topic.

Regarding the quality obtained with the proposed methods, one common problem was the preservation of long distance correlation. As discussed within the corresponding chapters, there are several ways to overcome this issue presented in the literature. A final product application for the proposed models should consider improvements in the form of more complex models. For our patch-based model, we could consider using more scales and multiple sizes of patches for visual quality, and accelerations like PCA approximation of the patches, truncation of the distance matrix, etc. For our solid texture synthesis model we could explore the loss terms recently designed for long distance correlation as mentioned in Chapter 2.

## 5.3 Future Research

Our solid textures generated using an anisotropic training configuration (i.e., taking less than three directions or using multiple examples) are promising. Results of this kind are scarce in the literature, but they could be useful in practice since they cover a broader set of textures.

The proposed solid texture generator shows good interpolation capabilities. In our experiments, the generated 3D patterns are always coherent. In the case where only two directions are specified, however, the free direction often results on long diagonal patterns that might be unpleasant. On the other hand, using a different image to specify the third direction (i.e., the multiple examples configuration), usually results on a middle point interpolation between the patterns in the top and side examples.

Instead of the user having to find a perfectly compatible image, the training should ideally accept some flexible reference patterns for the third direction without sacrificing the quality

of the first and second. We could envision a setting where the user labels pieces of the main example texture, to be used as references for the third direction. These pieces could be edges, constant colors, or a subset of the texture patterns. The single slice training could involve a hybrid loss with the perceptual and a Spatial GAN like loss that takes patches of an image as example database. We could then use weights to specify the importance of each term.

In a different application, the 3D nature of the proposed solid texture model makes it interesting for video applications. In this context, the proposed model could be applied for video style transfer, possibly without a need for computing the optical flow as done with current 2D methods.

Image style transfer is the task of automatically combining the content of one image and the style of another into a stylized output image. Typically the style of an artistic image is *transferred* to a photo-realistic one. This achieves compelling image transformations that can be used for digital photography, design, animation, etc. Recent successful results have been obtained with models similar to the 2D Texture Optimization using CNNs and the perceptual loss [91]. They formulate style transfer as an optimization problem that aims at creating a new image that matches the content characterization of an image and the style characterization of the other.

Video style transfer requires the application of style transfer to all the frames in a video in a way that ensures temporal consistency. Independent stylization creates a flickering artifact between consecutive frames, and inconsistencies after occlusion/dis-occlusion of scenes between far apart frames. Attempts to maintain temporal coherence employ constraints based on the optical flow to take movement into account. The quality of the video highly depends on this temporal consistency, and theoretically, the more frames are taken into account the better the quality. However, each extra frame makes the optimization problem more intensive.

We believe that a deep 3D CNN could facilitate the stylization of blocks of video, that is, several frames at the same time. The 3D nature of the convolution kernels would guarantee the consistency in both the spatial and the temporal dimensions without the need for temporal constraints in the optimization process. This could remove the necessity for estimating the optical flow and allow for the integration of longer temporal consistency. For instance, using the network architecture presented in Chapter 4 each pixel in one frame would have an impact in $\sim$ 300 different frames. For the training, we could envision a similar setting to the proposed single-slice training, allowing us to only compute the loss between a single 2D image and the corresponding frame in the training video and texture example.

# REFERENCES


[1] J. Portilla and E. P. Simoncelli, "A parametric texture model based on joint statistics of complex wavelet coefficients," *IJCV*, vol. 40, no. 1, pp. 49–71, 2000.

[2] G. Tartavel, Y. Gousseau, and G. Peyré, "Variational texture synthesis with sparsity and spectrum constraints," *JMIV*, vol. 52, no. 1, pp. 124–144, 2015.

[3] L. Y. Wei and M. Levoy, "Fast texture synthesis using tree-structured vector quantization," in *SIGGRAPH*, 2000, pp. 479–488.

[4] M. Ashikhmin, "Synthesizing natural textures," in *I3D*, 2001, pp. 217–226.

[5] A. A. Efros and W. T. Freeman, "Image quilting for texture synthesis and transfer," in *SIGGRAPH*, 2001, pp. 341–346.

[6] V. Kwatra, I. Essa, A. Bobick, and N. Kwatra, "Texture optimization for example-based synthesis," in *SIGGRAPH*, 2005, pp. 795–802.

[7] L. Gatys, A. S. Ecker, and M. Bethge, "Texture synthesis using convolutional neural networks," in *NIPS*, 2015, pp. 262 – 270.

[8] D. Ulyanov, V. Lebedev, A. Vedaldi, and V. Lempitsky, "Texture networks: Feed-forward synthesis of textures and stylized images," in *ICML*, 2016, pp. 1349–1357.

[9] T. Karras, T. Aila, S. Laine, and J. Lehtinen, "Progressive growing of GANs for improved quality, stability, and variation," in *ICLR*, 2018.

[10] U. Bergmann, N. Jetchev, and R. Vollgraf, "Learning texture manifolds with the periodic spatial GAN," in *ICML*, 2017, pp. 469–477.

[11] M. Cimpoi, S. Maji, I. Kokkinos, S. Mohamed, , and A. Vedaldi, "Describing textures in the wild," in *CVPR*, 2014.

[12] K. Perlin, "An image synthesizer," *SIGGRAPH*, pp. 287–296, 1985.

[13] L.-Y. Wei, "Texture synthesis by fixed neighborhood searching," Ph.D. dissertation, Stanford University, 2002.

[14] ——, "Texture synthesis from multiple sources," in *SIGGRAPH*, 2003, pp. 1–1.

[15] X. Qin and Y. h. Yang, "Aura 3d textures," *Transactions on Visualization and Computer Graphics*, vol. 13, no. 2, pp. 379–389, 2007.

[16] N. Pietroni, P. Cignoni, M. Otaduy, and R. Scopigno, "Solid-texture synthesis: A survey," *IEEE Computer Graphics and Applications*, vol. 30, no. 4, pp. 74–89, 2010.

[17] J. Kopf, C. Fu, D. Cohen-Or, O. Deussen, D. Lischinski, and T. Wong, "Solid texture synthesis from 2d exemplars," in *SIGGRAPH*, 2007, pp. 2:1–2:9.

[18] C. Barnes, E. Shechtman, A. Finkelstein, and D. B. Goldman, "Patchmatch: A randomized correspondence algorithm for structural image editing," in *SIGGRAPH*, 2009, pp. 24:1–24:11.

[19] G'MIC. (2016) Greyc's magic for image computing. http://gmic.eu/.

[20] L.-Y. Wei, S. Lefebvre, V. Kwatra, and G. Turk, "State of the art in example-based texture synthesis," in *Eurographics 2009 - State of the Art Reports*, 2009.

[21] M. S. Floater and K. Hormann, "Surface parameterization: a tutorial and survey," in *Advances in Multiresolution for Geometric Modelling*, 2005, pp. 157–186.

[22] B. Julesz, E. Gilbert, and J. D. Victor, "Visual discrimination of textures with identical third-order statistics," *Biological Cybernetics*, vol. 31, no. 3, pp. 137–140, 1978.

[23] B. Julesz, "Textons, the elements of texture perception, and their interactions," *Nature*, vol. 290, no. 5802, pp. 91–97, 1981.

[24] D. S. Swamy, K. J. Butler, D. M. Chandler, and S. S. Hemami, "Parametric quality assessment of synthesized textures." in *Human Vision and Electronic Imaging*, 2011, p. 78650B.

[25] J. Johnson, A. Alahi, and L. Fei-Fei, "Perceptual losses for real-time style transfer and super-resolution," in *ECCV*, 2016, pp. 694–711.

[26] A. Mordvintsev, N. Pezzotti, L. Schubert, and C. Olah, "Differentiable image parameterizations," *Distill*, 2018.

[27] B. Galerne, Y. Gousseau, and J.-M. Morel, "Random phase textures: Theory and synthesis," *TIP*, vol. 20, no. 1, pp. 257 – 267, 2011.

[28] J. S. De Bonet, "Multiresolution sampling procedure for analysis and synthesis of texture images," in *SIGGRAPH*, 1997, pp. 361–368.

[29] D. J. Heeger and J. R. Bergen, "Pyramid-based texture analysis/synthesis," in *SIGGRAPH*, 1995, pp. 229–238.

[30] J. Rabin, G. Peyré, J. Delon, and M. Bernot, "Wasserstein barycenter and its application to texture mixing," in *SSVM*, 2012, pp. 435–446.

[31] G. Tartavel, Y. Gousseau, and G. Peyré, "Constrained sparse texture synthesis," in *SSVM*, 2013, pp. 186–197.

[32] E. P. Simoncelli and W. T. Freeman, "The steerable pyramid: A flexible architecture for multi-scale derivative computation," in *ICIP*, 1995, pp. 444–447.

[33] S. C. Zhu, Y. Wu, and D. Mumford, "Filters, random fields and maximum entropy (frame): Towards a unified theory for texture modeling," *IJCV*, vol. 27, no. 2, pp. 107–126, 1998.

[34] V. De Bortoli, A. Desolneux, B. Galerne, and A. Leclaire, "Patch redundancy in images: A statistical testing framework and some applications," *SIAM Journal on Imaging Sciences*, vol. 12, no. 2, pp. 893–926, 2019.

[35] Y. Lu, S.-C. Zhu, and Y. N. Wu, "Learning frame models using cnn filters," in *AAAI Conference on Artificial Intelligence*, 2016, pp. 1902–1910.

[36] T. Briand, J. Vacher, B. Galerne, and J. Rabin, "The heeger & bergen pyramid based texture synthesis algorithm," *IPOL*, vol. 4, pp. 276–299, 2014.

[37] G. Peyré, "Sparse modeling of textures," *JMIV*, vol. 34, no. 1, pp. 17–31, 2009.

[38] A. A. Efros and T. K. Leung, "Texture synthesis by non-parametric sampling," in *ICCV*, 1999, pp. 1033–1038.

[39] A. Hertzmann, C. E. Jacobs, N. Oliver, B. Curless, and D. H. Salesin, "Image analogies," in *SIGGRAPH*, 2001, pp. 327–340.

[40] S. Zelinka and M. Garland, "Jump map-based interactive texture synthesis," *ACM Trans. Graph.*, vol. 23, no. 4, pp. 930–962, 2004.

[41] B. Guo, H. Shum, and Y.-Q. Xu, "Chaos mosaic: Fast and memory efficient texture synthesis," Microsoft Research, Tech. Rep. MSR-TR-2000-32, 2000.

[42] L. Liang, C. Liu, Y.-Q. Xu, B. Guo, and H.-Y. Shum, "Real-time texture synthesis by patch-based sampling," *ACM Trans. Graph.*, vol. 20, no. 3, pp. 127–150, 2001.

[43] V. Kwatra, A.Schödl, I. Essa, G. Turk, and A. Bobick, "Graphcut textures: image and video synthesis using graph cuts," in *SIGGRAPH*, 2003, pp. 277–286.

[44] J. Han, K. Zhou, L.-Y. Wei, M. Gong, H. Bao, X. Zhang, and B. Guo, "Fast example-based surface texture synthesis via discrete optimization," *The Visual Computer*, vol. 22, no. 9, pp. 918–925, 2006.

[45] A. Kaspar, B. Neubert, D. Lischinski, M. Pauly, and J. Kopf, "Self tuning texture optimization," in *CGF*, vol. 34, no. 2, 2015, pp. 349–359.

[46] J. Chen and B. Wang, "High quality solid texture synthesis using position and index histogram matching," *Vis. Comput.*, vol. 26, no. 4, pp. 253–262, 2010.

[47] M. Elad and P. Milanfar, "Style transfer via texture synthesis," *TIP*, vol. 26, no. 5, pp. 2338–2351, 2017.

[48] L. Yann, "Modeles connexionnistes de lapprentissage," Ph.D. dissertation, Universite Paris 6, 1987.

[49] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[50] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, 2014.

[51] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "Imagenet large scale visual recognition challenge," *Int. J. Comput. Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[52] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *International Conference on Multimedia*, 2014, pp. 675–678.

[53] I. Ustyuzhaninov, W. Brendel, L. Gatys, and M. Bethge, "What does it take to generate natural textures?" in *ICLR*, 2017.

[54] R. Zhang, P. Isola, A. A. Efros, E. Shechtman, and O. Wang, "The unreasonable effectiveness of deep features as a perceptual metric," in *CVPR*, 2018, pp. 586–595.

[55] G. Liu, Y. Gousseau, and G. Xia, "Texture synthesis through convolutional neural networks and spectrum constraints," in *ICPR*, 2016, pp. 3234–3239.

[56] O. Sendik and D. Cohen-Or, "Deep correlations for texture synthesis," *ACM Trans. Graph.*, vol. 36, no. 5, pp. 161:1–161:15, 2017.

[57] D. Ulyanov, A. Vedaldi, and V. Lempitsky, "Improved texture networks: Maximizing quality and diversity in feed-forward stylization and texture synthesis," in *CVPR*, 2017, pp. 4105–4113.

[58] Y. Li, C. Fang, J. Yang, Z. Wang, X. Lu, and M. Yang, "Diversified texture synthesis with feed-forward networks," in *CVPR*, 2017, pp. 266–274.

[59] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *NIPS*, 2014, pp. 2672–2680.

[60] N. Jetchev, U. Bergmann, and R. Vollgraf, "Texture synthesis with spatial generative adversarial networks," *CoRR*, 2016.

[61] Y. Zhou, Z. Zhu, X. Bai, D. Lischinski, D. Cohen-Or, and H. Huang, "Non-stationary texture synthesis by adversarial expansion," *ACM Trans. Graph.*, vol. 37, no. 4, pp. 49:1–49:13, 2018.

[62] D. R. Peachey, "Solid texturing of complex surfaces," *SIGGRAPH*, pp. 279–286, 1985.

[63] D. Ghazanfarpour and J. Dischler, "Spectral analysis for automatic 3-d texture generation," *Computers & Graphics*, vol. 19, no. 3, pp. 413–422, 1995.

[64] Y. Dong, S. Lefebvre, X. Tong, and G. Drettakis, "Lazy solid texture synthesis," in *EGSR*, 2008, pp. 1165–1174.

[65] X. Qin and Y.-H. Yang, "Basic gray level aura matrices: theory and its application to texture synthesis," in *ICCV*, vol. 1, 2005, pp. 128–135.

[66] J. M. Dischler, D. Ghazanfarpour, and R. Freydier, "Anisotropic solid texture synthesis using orthogonal 2d views," *CGF*, vol. 17, no. 3, pp. 87–95, 1998.

[67] D. Ghazanfarpour and J. Dischler, "Generation of 3d texture using multiple 2d models analysis," *CGF*, vol. 15, no. 3, pp. 311–323, 1996.

[68] L. Y. Wei and M. Levoy, "Fast texture synthesis using tree-structured vector quantization," in *SIGGRAPH*, 2000, pp. 479–488.

[69] S. Lefebvre and H. Hoppe, "Parallel controllable texture synthesis," in *SIGGRAPH*, 2005, pp. 777–786.

[70] C. De and F. Y. Shih, "Quantification and relative comparison of synthesized texture," in *IPCV*, 2011, p. 1.

[71] Wen-Chieh Lin, J. Hays, Chenyu Wu, Yanxi Liu, and V. Kwatra, "Quantitative evaluation of near regular texture synthesis algorithms," in *CVPR*, vol. 1, 2006, pp. 427–434.

[72] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, X. Chen, and X. Chen, "Improved techniques for training gans," in *NIPS*, 2016, pp. 2234–2242.

[73] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *CVPR*, 2015.

[74] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *CVPR*, 2016.

[75] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter, "Gans trained by a two time-scale update rule converge to a local nash equilibrium," in *NIPS*, 2017, pp. 6626–6637.

[76] D. Coleman, P. Holland, N. Kaden, V. Klema, and S. C. Peters, "A system of subroutines for iteratively reweighted least squares computations," *Trans. Math. Softw.*, vol. 6, no. 3, pp. 327–336, 1980.

[77] L.-Y. Wei, J. Han, K. Zhou, H. Bao, B. Guo, and H.-Y. Shum, "Inverse texture synthesis," *TOG*, vol. 27, no. 3, p. 52, 2008.

[78] D. Simakov, Y. Caspi, E. Shechtman, and M. Irani, "Summarizing visual data using bidirectional similarity," in *CVPR*, 2008, pp. 1–8.

[79] P.-L. Lions and B. Mercier, "Splitting algorithms for the sum of two nonlinear operators," *SIAM Journal on Numerical Analysis*, vol. 16, no. 6, pp. 964–979, 1979.

[80] R. Burkard, M. Dell'Amico, and S. Martello, *Assignment Problems*. SIAM, 2012.

[81] S. Bougleux and L. Brun, "Linear sum assignment with edition," *CoRR*, 2016.

[82] C. N. Vasconcelos and B. Rosenhahn, "Bipartite graph matching computation on gpu," in *EMMCVPR*, 2009, pp. 42–55.

[83] P. Arias, V. Caselles, and G. Facciolo, "Analysis of a variational framework for exemplar-based image inpainting," *Multiscale Modeling & Simulation*, vol. 10, no. 2, pp. 473–514, 2012.

[84] R. Webster, "Innovative non-parametric texture synthesis via patch permutations," *CoRR*, 2018.

[85] B. Galerne, A. Leclaire, and J. Rabin, "A texture synthesis model based on semi-discrete optimal transport in patch space," *SIAM Journal on Imaging Sciences*, vol. 11, no. 4, pp. 2456–2493, 2018.

[86] S. Ferradans, N. Papadakis, J. Rabin, G. Peyré, and J.-F. Aujol, "Regularized discrete optimal transport," in *SSVM*, 2013, pp. 428–439.

[87] J. Rabin, S. Ferradans, and N. Papadakis, "Adaptive color transfer with relaxed optimal transport," in *ICIP*, 2014, pp. 4852–4856.

[88] D. P. Bertsekas, "A new algorithm for the assignment problem," *Mathematical Programming*, vol. 21, no. 1, pp. 152–171, 1981.

[89] ——, "The auction algorithm: A distributed relaxation method for the assignment problem," *Annals of operations research*, vol. 14, no. 1, pp. 105–123, 1988.

[90] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *ICML*, vol. 37, 07–09 Jul 2015, pp. 448–456.

[91] L. A. Gatys, A. S. Ecker, and M. Bethge, "Image style transfer using convolutional neural networks," in *CVPR*, 2016, pp. 2414–2423.

[92] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *ICLR*, 2015.

[93] G. Marsaglia, "Xorshift rngs," *Journal of Statistical Software*, vol. 8, no. 14, pp. 1–6, 2003.

[94] B. Galerne, A. Leclaire, and L. Moisan, "Texton noise," *CGF*, vol. 36, no. 8, pp. 205–218, 2017.

[95] J. Gutierrez, J. Rabin, B. Galerne, and T. Hurtut, "Optimal patch assignment for statistically constrained texture synthesis," in *SSVM*, 2017, pp. 172–183.

[96] V. Lempitsky, A. Vedaldi, and D. Ulyanov, "Deep image prior," in *CVPR*, 2018, pp. 9446–9454.

[97] G. Gilet, B. Sauvage, K. Vanhoey, J.-M. Dischler, and D. Ghazanfarpour, "Local random-phase noise for procedural texturing," *ACM Trans. Graph.*, vol. 33, no. 6, pp. 195:1–195:11, 2014.