ABSTRACT

Title of Dissertation: Repetitive Patterns in Digital Images:
Raster Analysis and Vector Synthesis

Peihan Tu
Doctor of Philosophy, 2024

Dissertation Directed by: Professor Matthias Zwicker
Department of Computer Science

Repetitive patterns are ubiquitous in various fields, including textile design, digital art, web design, and graphic design, offering significant practical and creative benefits. This dissertation delves into the automated analysis and synthesis of these patterns within digital imagery. We focus on the replication of geometric structures in vector patterns and geometric analysis of raster patterns, highlighting the unique challenges and methodologies involved in each.

Creating repetitive vector patterns, characterized by their intricate shapes, is notably challenging and laborious. This intricate process not only demands significant time investment but also requires a unique combination of creative insight and meticulous precision. Although computational methods have streamlined some aspects of manual creation, they predominantly cater to simple elements, leaving complex patterns less explored. To overcome this, we introduce a computational approach for synthesizing continuous structures in

vector patterns from exemplars. This approach innovates on existing sample-based discrete element synthesis methods to consider not only sample positions (geometry) but also their connections (topology). Additionally, we present an example-based method to synthesize more general patterns with diverse shapes and structured local interactions, incorporating explicit clustering as part of neighborhood similarity and iterative sample optimization for more robust sample synthesis and pattern reconstruction.

Conversely, raster textures, essential visual elements in both real images and computer-generated imagery, present distinct challenges in editing due to their pixel-based nature. Traditional texture editing, often a repetitive and tedious task, requires manual adjustments of textons—small, recurring patterns that define textures. Addressing this, we propose a novel, fully unsupervised method using a compositional neural model to represent textures. Each texton is modeled as a 2D Gaussian function, capturing its shape and detailed appearance. This discrete composition of Gaussian textons simplifies editing and enables the efficient synthesis of new textures through a generator network. Our method facilitates a broad spectrum of applications, from texture transfer and diversification to animation and direct manipulation, significantly advancing texture analysis, modeling, and editing techniques. This approach not only enhances the capability for creating visually appealing images with controllable textures but also opens up new avenues for exploration in digital imagery.

Repetitive Patterns in Digital Images: Raster Analysis and Vector Synthesis

by

Peihan Tu

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2024

Dissertation Committee:

Professor Matthias Zwicker, Chair/Advisor
Dr. Li-Yi Wei, Co-Advisor
Professor Ming Lin
Professor Jia-Bin Huang
Professor Joseph F. JaJa

© Copyright by
Peihan Tu
2024

Dedication

To my family.

Acknowledgments

I would like to express my deepest gratitude to my advisors, Professor Matthias Zwicker and Dr. Li-Yi Wei, for their unwavering support, invaluable guidance, and continuous encouragement throughout my doctoral journey. Their expertise, insightful feedback, and dedication have been instrumental in shaping my research and academic growth.

I am indebted to Professor Hui Huang and Professor Dani Lischinski for encouraging me pursuing my academic dream in the US. I have had the privilege of working with them and Dr. Li-Yi Wei even before embarking on my PhD.

I would like to extend my heartfelt thanks to my committee members, Professor Jia-Bin Huang, Professor Ming Lin, and Professor Joseph Jaja, for their valuable feedback, insightful suggestions, and scholarly contributions, which have greatly enhanced the quality of this work.

I would like to thank all my colleagues Geng, Shuhong, Saeed, Alex, Jiaye, Hsien-Yu in Zwicker's lab for insightful discussion and help. My PhD research does not belong to one of the most popular topics. I hope we can collaborate on our commonly interested topics in the future. I am also deeply thankful to all my colleagues in the UMD CS department with whom I've engaged in insightful discussions. Their diverse perspectives, intellectual curiosity, and collaborative spirit have played a significant role in shaping my research

journey. Due to the sheer number of individuals, I regret that I am unable to list all of their names here.

Additionally, I want to express my gratitude to my friends. Their friendship, laughter, and shared experiences have been a source of joy and respite during the challenging moments of my academic journey.

I am profoundly grateful to my fiancée, Cecilia, for her unwavering love, understanding, and encouragement throughout this demanding endeavor. Her patience, support, and belief in my aspirations have been a constant source of strength and motivation.

To my parents, whose unwavering love, encouragement, and sacrifices have been the cornerstone of my journey, I owe the deepest gratitude. Their endless support, belief in my abilities, and sacrifices have been my guiding light through every challenge and triumph.

Table of Contents

Dedication ii

Acknowledgements iii

Table of Contents v

List of Tables viii

List of Figures ix

Chapter 1: Introduction 1
- 1.1 Synthesis of vector patterns by examples . . . 1
- 1.2 Analysis of raster textures for editing . . . 2
- 1.3 Publications . . . 3

Chapter 2: Continuous Curve Textures 4
- 2.1 Introduction . . . 4
- 2.2 Related Work . . . 7
  - 2.2.1 Example-based Pattern Generation . . . 7
  - 2.2.2 Interactive Authoring . . . 8
- 2.3 User Interface . . . 8
- 2.4 Method . . . 10
  - 2.4.1 Representation . . . 11
  - 2.4.2 Similarity Measure . . . 14
  - 2.4.3 Pattern Synthesis . . . 19
  - 2.4.4 Pattern Reconstruction . . . 29
- 2.5 Evaluation . . . 31
  - 2.5.1 Results . . . 31
  - 2.5.2 Ablation Study . . . 33
  - 2.5.3 Comparison to Previous Methods . . . 34
- 2.6 Conclusions, Limitations, and Future Work . . . 35

Chapter 3: Clustered Vector Textures 39
- 3.1 Introduction . . . 39
- 3.2 Related Work . . . 41
- 3.3 Overview . . . 44
- 3.4 Representation . . . 45

3.4.1 Vector pattern . . . 45
3.4.2 Clustered pattern representation . . . 45
3.5 Neighborhood and Similarity . . . 46
3.6 Pattern Synthesis . . . 48
3.6.1 Optimization objective . . . 48
3.6.2 Initialization . . . 49
3.6.3 Search step . . . 49
3.6.4 Assignment step . . . 49
3.6.5 Clustering step . . . 52
3.6.6 Reconstruction . . . 59
3.6.7 Hierarchical optimization . . . 59
3.7 Results . . . 60
3.7.1 Ablation studies . . . 60
3.7.2 Comparison to previous methods . . . 60
3.7.3 Parameters . . . 61
3.8 Conclusions, Limitations and Future Work . . . 62

Chapter 4: Compositional Neural Textures 73
4.1 Introduction . . . 73
4.2 Related Work . . . 77
4.2.1 Neural Texture Representations . . . 77
4.2.2 Texture Editing . . . 78
4.2.3 Compositional Neural Representations . . . 80
4.3 Method . . . 81
4.3.1 Overview . . . 81
4.3.2 Encoding Textures as Gaussians . . . 83
4.3.3 Gaussian Splatting . . . 88
4.3.4 Decoding Images from Splatted Gaussians . . . 89
4.3.5 Unsupervised Compositional Textons Learning . . . 90
4.3.6 Overall Losses and Training Procedure . . . 96
4.3.7 Dataset . . . 97
4.4 Applications . . . 98
4.4.1 Texture Diversification . . . 98
4.4.2 Texture Transfer . . . 99
4.4.3 Revealing and Modifying Texture Variations . . . 101
4.4.4 Texture Interpolation . . . 102
4.4.5 Texton-centric Editing . . . 103
4.4.6 Texture Animation . . . 104
4.5 Analysis . . . 105
4.5.1 Comparison to “Segment Anything” . . . 105
4.5.2 Comparisons on Texture Transfer . . . 106
4.5.3 Ablation Studies . . . 106
4.6 Limitations and Future Work . . . 110

Chapter 5: Appendix: Continuous Curve Textures 126

5.1 Existence Assignment . . . 126
5.2 Parameters . . . 128
5.3 Performance . . . 129
5.4 Texture Synthesis . . . 130

Chapter 6: Appendix: Clustered Vector Textures 130
6.1 Robustness analysis . . . 130
6.1.1 Initial conditions and optimization . . . 130
6.1.2 Hyperparameters . . . 131
6.1.3 Clustering operators . . . 131
6.1.4 More Comparisons with prior art . . . 132
6.1.5 A large synthesis result . . . 132
6.2 Implementation details . . . 132
6.2.1 Representation . . . 132
6.2.2 Assignment step . . . 132
6.2.3 Performance . . . 133

Chapter 7: Appendix: Compositional Neural Textures 133
7.1 Dataset . . . 133
7.2 Evaluation details . . . 134
7.3 Texton segmentation . . . 135
7.4 Texture transfer . . . 135
7.5 Comparisons . . . 135
7.6 Ablation studies . . . 136
7.7 Implementation details . . . 136
7.7.1 Network architectures . . . 136
7.7.2 Training details . . . 137
7.7.3 Texture animations . . . 138
7.8 Options for High-quality Productions . . . 139

List of Tables

2.1 Sample attributes. 11

4.1 Quantitative evaluations of ablation studies. 108

5.1 Parameters. 128

6.1 Ratios between numbers of clustering operations and samples/clusters in each hierarchy. 131

6.2 Running time of each algorithm component at different hierarchy (unit: second). 143

7.1 The architecture of $\mathcal{E}_{FC}$. 168

List of Figures

2.1 Example inputs and outputs of our method. 4
2.2 Comparison with prior point/sample synthesis algorithms. 5
2.3 User Interface. 9
2.4 Autocomplete and clone. 10
2.5 Pattern representation. 11
2.6 Different neighborhood matching methods. 17
2.7 Existence assignment example. 21
2.8 Edge assignment example. 22
2.9 Orientation assignment example. 26
2.10 Hierarchical synthesis visualization. 28
2.11 Curve reconstruction from a graph using orientation attributes. 30
2.12 Automatic synthesis results by our method. 32
2.13 Automatic synthesis and user-assisted results. 33
2.14 Algorithm robustness to different initialization. 34
2.15 Ablation study for $\mathbf{o}$. 35
2.16 Ablation study. 36
2.17 Comparison of our algorithm with enhanced [1]. 37
2.18 Failure case. 38

3.1 Example inputs and outputs of our method. 40
3.2 Pattern synthesis with clustered sample representation. 41
3.3 Comparison with previous methods. 42
3.4 A toy example of computing the probabilistic pairwise ordering function $f_s(s_o, s'_o)$. 50
3.5 Z-index (depth) assignment. 64
3.6 A toy example of link confidence computation. 65
3.7 Illustration of cluster shape energy computation. 66
3.8 Clustering objective (Equation (3.9)) with respect to number of iterations. 67
3.9 Clustering operators. 68
3.10 Evolution of patterns in the optimization. 68
3.11 Automatic synthesis results with a variety of exemplar patterns. 69
3.12 Automatic synthesis with user-editing. 70
3.13 Comparisons against improved [2] with baseline clustering. 71
3.14 Ablation study. 72

3.15 Limitations. 72

4.1 Compositional neural representation for texture editing. 74
4.2 Architecture of the proposed method. 112
4.3 Texture diversification. 113
4.4 Texture transfer. 114
4.5 Stylizing an image by tranferring textures. 115
4.6 Spatial control in texture transfer. 115
4.7 Scale control in texture transfer. 116
4.8 Revealing and modifying appearance variations. 116
4.9 Revealing and modifying geometric variations. 117
4.10 Texture interpolation and morphing. 118
4.11 Texton editing. 119
4.12 Edit propagation. 120
4.13 Texture animation. 121
4.14 Comparison to segment anything (SA). 122
4.15 Comparisons with existing approaches on texture transfer. 123
4.16 Ablation studies on losses. 124
4.17 Ablation studies on network components. 125
4.18 Improper compositional representation for elongated structures. 125

5.1 Comparison of our algorithm to texture synthesis. 141
6.1 Energy plots. 142
6.2 Optimization process with patch-based initialization. 144
6.3 Continuation from Figure 6.2. 145
6.4 Optimization process with random sample initialization. 146
6.5 Continuation from Figure 6.4. 147
6.6 Synthesis results using different hyperparameters $w_{\mathbf{f}}$. 148
6.7 Synthesis results using different hyperparameters w_s. 148
6.8 Synthesis results using different combinations of clustering operators. . . . 149
6.9 More comparisons with previous methods. 149
6.10 A large synthesis result. 150
7.1 100 original dataset samples without augmentation. 151
7.2 Uncurated texton segmentation results. 152
7.3 Continuation from Figure 7.2. 153
7.4 Two texture transfer mechanisms. 154
7.5 Scale control in texture transfer. 155
7.6 Uncurated comparisons with SA. 156

7.7 Comparison with state-of-the-art approaches on texture transfer. 157
7.8 Continuation from Figure 7.7. 158
7.9 Uncurated comparisons with state-of-the-art approaches on texture transfer. 159
7.10 Continuation from Figure 7.9. 160
7.11 Continuation from Figure 7.9. 161
7.12 Continuation from Figure 7.9. 162
7.13 Continuation from Figure 7.9. 163
7.14 Continuation from Figure 7.9. 164
7.15 Ablation studies on losses. 165
7.16 Continuation of Figure 7.15. 166
7.17 Ablation studies on network components. 167

Chapter 1: Introduction

Repetitive patterns in digital images play an essential role in the realm of digital image processing, merging the elements of aesthetics, technology, and practical application seamlessly. These patterns, defined by their consistent motifs and structures, enhance the visual impact of images and are fundamental to various computational processes involved in image analysis, synthesis, and editing. This dissertation focuses on exploring these patterns through the synthesis of vector patterns from examples and the analysis of raster textures to enable editing or controlled synthesis.

1.1 Synthesis of vector patterns by examples

Repetitive patterns are ubiquitous in natural and human-made objects,and can be created with a variety of tools and methods. Manual authoring provides unmatched degree of freedom and control, but can require significant artistic expertise and manual labor. Computational methods can automate parts of the manual creation process, but are mainly tailored for discrete pixels or elements instead of more general continuous structures. In this dissertation, we propose two example-based approaches for synthesizing pure curve patterns and for synthesizing more general vector patterns with diverse shapes and structured local interactions without elongated elements.

To synthesize curve patterns by examples, our main idea is to extend prior sample-based discrete element synthesis methods to consider not only sample positions (geometry) but also their connections (topology). Since continuous structures can exhibit higher complexity than discrete elements, we also propose robust, hierarchical synthesis to enhance output quality. Our algorithm can generate a variety of continuous curve patterns fully automatically. For further quality improvement and customization, we also present an autocomplete user interface to facilitate interactive creation and iterative editing. We evaluate our methods and interface via different patterns, ablation studies, and comparisons with alternative methods. We will introduce this approach in Chapter 2. To synthesize more general vector patterns by examples, using a sample-based representation, our main idea is adding explicit clustering as part of neighborhood similarity and iterative sample optimization for more robust sample synthesis and pattern reconstruction. The results indicate that our method can outperform existing methods on synthesizing a variety of structured vector textures. More details will be in Chapter 3.

1.2 Analysis of raster textures for editing

Texture plays a vital role in enhancing visual richness in both real photographs and computer-generated imagery. However, the process of editing textures often involves laborious and repetitive manual adjustments of textons, which are the small, recurring local patterns that define textures. In this dissertation, we introduce a fully unsupervised approach for representing textures using a compositional neural model that captures individual textons. We represent each texton as a 2D Gaussian function whose spatial support

approximates its shape, and an associated feature that encodes its detailed appearance. By modeling a texture as a discrete composition of Gaussian textons, the representation offers both expressiveness and ease of editing. Textures can be edited by modifying the compositional Gaussians within the latent space, and new textures can be efficiently synthesized by feeding the modified Gaussians through a generator network in a feed-forward manner. This approach enables a wide range of applications, including transferring appearance from an image texture to another image, diversifying textures, texture interpolation, revealing/modifying texture variations, edit propagation, texture animation, and direct texton manipulation. The proposed approach contributes to advancing texture analysis, modeling, and editing techniques, and opens up new possibilities for creating visually appealing images with controllable textures. We will discuss this part in detail in Chapter 4.

1.3 Publications

The dissertation is based on the following peer-reviewed publications.

- *Tu, Peihan, Li-Yi Wei, Koji Yatani, Takeo Igarashi, and Matthias Zwicker. "Continuous curve textures." ACM Transactions on Graphics (TOG) 39, no. 6 (2020): 1-16.*
- *Tu, Peihan, Li-Yi Wei, and Matthias Zwicker. "Clustered vector textures." ACM Transactions on Graphics (TOG) 41, no. 4 (2022): 1-23.*

and under-reviewed manuscript

- *Tu, Peihan, Li-Yi Wei, and Matthias Zwicker. "Compositional Neural Textures." (2024)*

Chapter 2: Continuous Curve Textures

2.1 Introduction

Repetitive patterns are fundamental for a variety of tasks in design [5, 6] and engineering [7, 8, 9, 10, 11]. Manually creating these patterns provides high degrees of individual freedom, but can also require significant technical/artistic expertise and manual labor. These usability barriers can be reduced by automatic methods that can synthesize patterns similar to user-supplied exemplars [1, 12, 13, 14, 15, 5, 6, 16, 17, 18]. However, existing techniques mainly focus on discrete patterns consisting of image pixels or shape elements, and might not apply to general patterns consisting of continuous curves, which can be connected or intersected with one another.

In this chapter, we propose an example-based method that can automatically synthe-

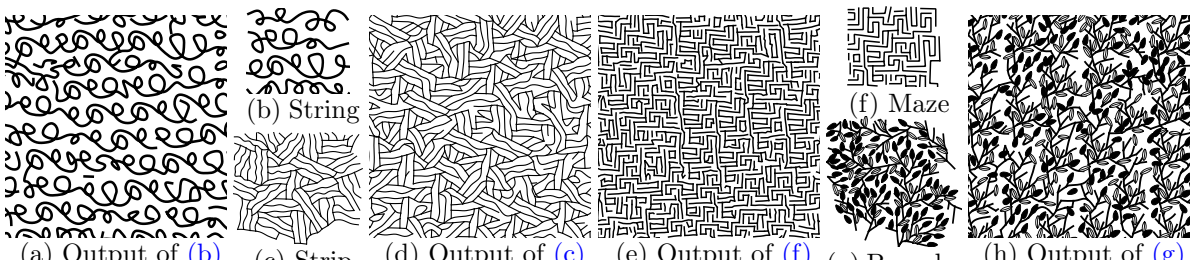
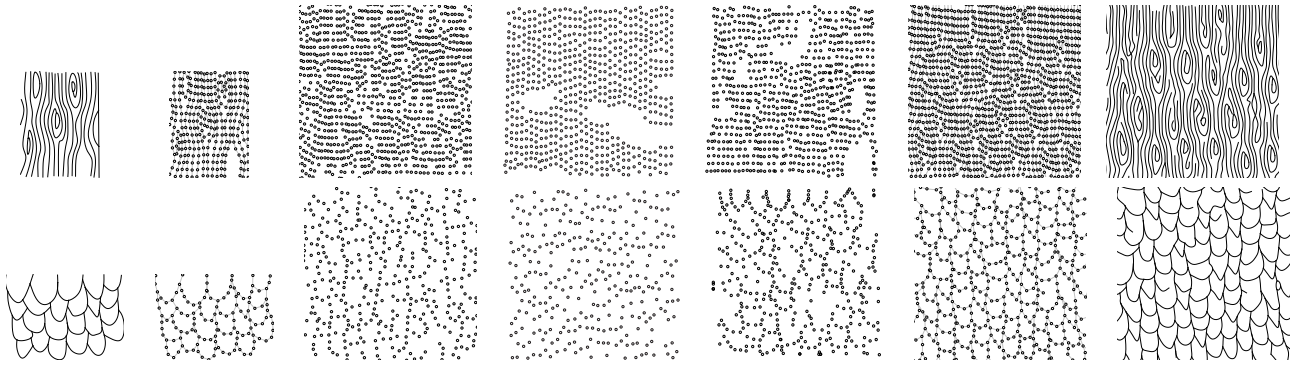


Figure 2.1: *Example inputs and outputs of our method.* Given a small input exemplar, our algorithm can synthesize various types of continuous curve textures. Our results are shown in vector format. Please zoom in to see the details.



(a) Exemplar (b) Samples of (a) (c) [1] (d) [3] (e) [4] (f) Our output samples (g) Reconstruction of (f)

Figure 2.2: *Comparison with prior point/sample synthesis algorithms.* The methods in [1, 3, 4] generate samples without considering their connections in the exemplars (b). Thus, their results, as shown in (c), (d), and (e), preserve the sample distributions less well than ours in (f). It is also unclear how to reconstruct continuous curve patterns from (c), (d), and (e). (g) shows the curve reconstruction from (f).

size continuous curve patterns from user-supplied exemplars. Similar to prior pixel/sample-based methods [19, 1, 20, 21, 12, 6, 3], users can provide exemplars and have the algorithm automatically produce results in desired sizes and shapes. However, different from previous methods and systems that are restricted to discrete pixels/elements or limited continuous structures, our method can handle both discrete elements and continuous curves in a variety of patterns (Figure 4.1).

Our main idea is to extend prior sample-based element synthesis methods [1, 5, 12, 17] to consider not only sample positions (geometry) but also their connections (topology) in all major algorithm components, including pattern representation, neighborhood similarity, and synthesis optimization consisting of search and assignment steps. Our algorithm uses a graph representation for both topology synthesis and geometric path reconstruction for general continuous patterns, in contrast to the graph representations in [17, 18] that only apply to discrete elements. Since continuous patterns can exhibit higher complexity than

discrete elements, we propose robust, hierarchical synthesis [22, 23] to enhance output quality.

Automatically generated outputs, although convenient, might not have sufficient quality or fit what users have in mind for their particular applications. To facilitate further editing and customization, we also propose an interactive autocomplete authoring interface [24, 18] built upon our synthesis algorithm components. Similar to existing design tools, users can create various free-style patterns. When they have sufficient exemplars and would like to reduce further manual repetitions, they can specify an output domain to be automatically filled [5, 24]. The synthesized patterns resemble and seamlessly connect with what has already been drawn. If not satisfied, users can accept or modify the predictions, or ask for re-synthesis to maintain full control. They can further designate specific source regions for cloning to target regions.

We analyze our algorithm via ablation studies, compare it with alternative methods, and demonstrate the quality and accessibility of our system via pattern design results.

In sum, the contributions of this chapter are:

- A hierarchical representation and a synthesis method for both geometry and topology of continuous and discrete patterns.
- An interactive authoring system with autocomplete functions to reduce manual workloads and facilitate user control.

2.2 Related Work

Our work is inspired by prior art in vector patterns, image textures, and interactive workflows. Procedural methods [25, 26, 27] can produce intricate structures, but are limited in scope and difficult to generalize for different types of patterns. Example-based methods are general, but existing work predominantly focuses on image textures [19, 6, 28] rather than vector patterns. Below, we survey methods most related to our work.

2.2.1 Example-based Pattern Generation

Example-based methods are designed to generate large patterns from small exemplars with an optional control provided by the users [13, 1, 12, 14, 15, 21, 3, 17, 4, 29, 30, 31]. However, these methods target discrete elements or samples [13, 1, 14, 21, 17, 18, 4] and treat continuous structures as special cases via curve/surface reconstruction from point samples [1, 3]. [3] reconstructs the output surface from synthesized point samples via their associated surface normals without considering sample connections, and thus can only be applied to surfaces relatively smooth to the underlying sampling density. [4] extends neural point synthesis by treating a graph edge as a line of points, which is essentially point synthesis. The neural optimization method does not provide the same flexibility and efficiency as in our method and it is unstable when the points contain attributes beyond positions. Relatively few works focus on curves, such as enriching details of given coarse curves [32] or growing L-system-like curves [33]. Our system is inspired by these prior sample-based algorithms [1, 3, 18], but we explicitly incorporate both samples and their connectivity into our representation and optimization to synthesize more general continuous

structures, as shown in Figure 2.2.

2.2.2 Interactive Authoring

Workflow analysis has been investigated to assist various content creation task [34]. Examples includes static and animated sketches [24, 35], 3D sculpting [36, 37], texture design [16], hand-writing beautification [38], and image editing [39, 40]. Our work is inspired by prior autocomplete [24, 35, 36, 16] and interactive systems [5, 41, 18]. However, these systems can automate only relatively simple patterns (e.g., repetitive hatches or strokes). Our method can automatically generate diverse and complex continuous curve patterns to facilitate iterative design with reduced input workload.

2.3 User Interface

Our system can be used for both automatic synthesis and interactive editing. Similar to prior work like traditional texture synthesis [19], the user provides an exemplar pattern and lets our system automatically produce the output with desired size and shape. The exemplar is represented via Bézier curves. Inputs in other formats can be converted to Bézier curves (for example, by vectorizing a raster image).

Since the automatic synthesis results might not be what the users want and they might need to create new patterns manually, we also provide an interface built upon our automatic synthesis algorithms for users to author patterns interactively. Through the interface, users can specify an output domain in desired size and shape (Figure 2.4a) and let our system predict patterns that resemble what the users have already drawn (*autocomplete* mode,

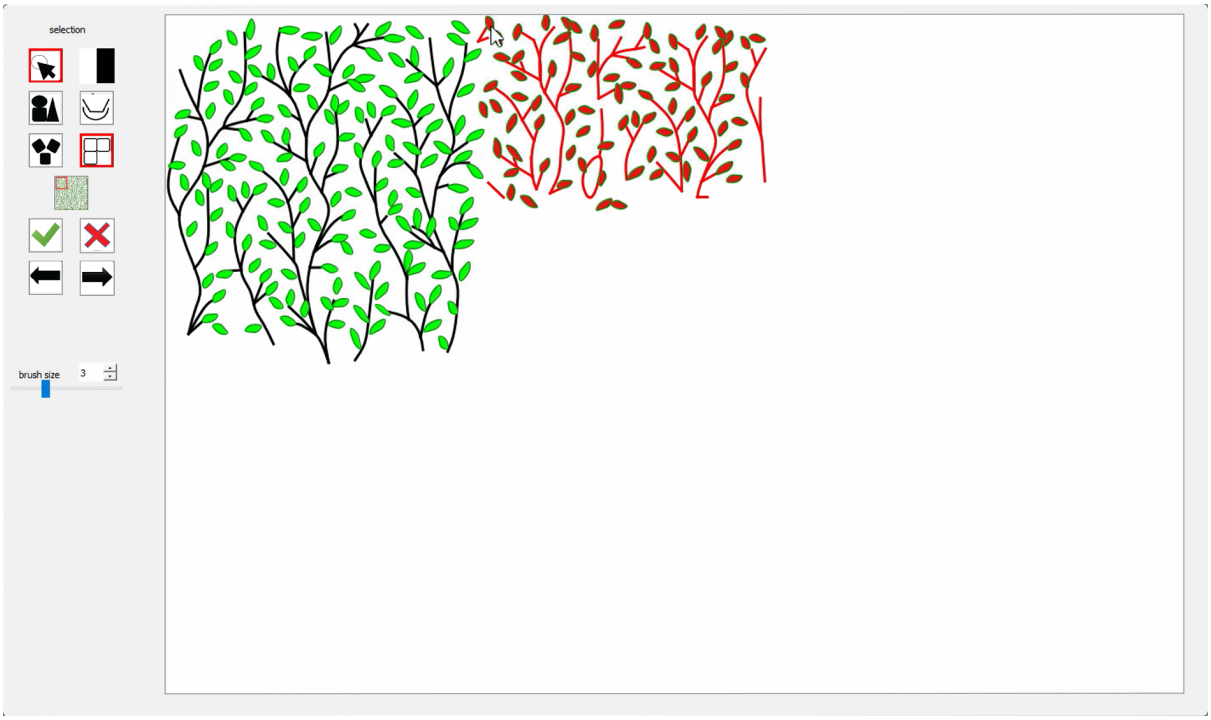
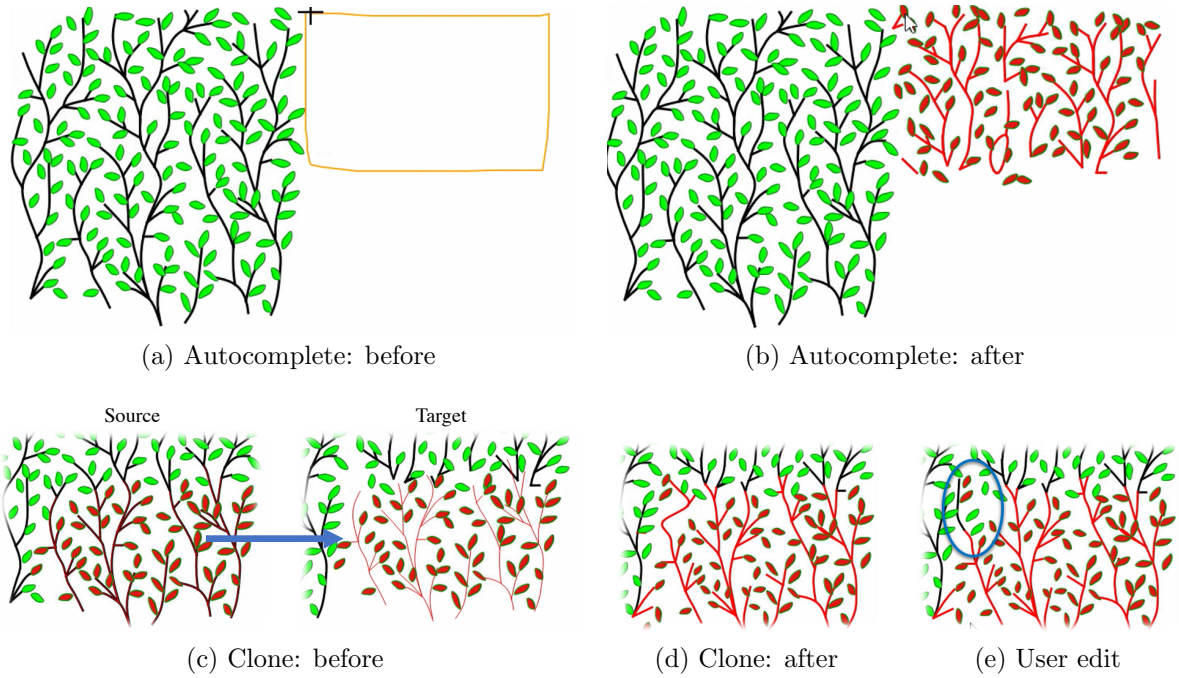


Figure 2.3: *User Interface.* Our interface has a widget panel and a canvas. The widget panel provides basic tools, such as selection in a vector editor, as well as controls to the graphics parameters, including color and pen width, and modes unique to our autocomplete system.

Figures 2.4a and 2.4b). The users can also explicitly control the prediction by copying-pasting from an input region to an output region (*clone* mode, Figures 2.4c and 2.4d). They can accept, partially accept, or reject the predictions via keyboard shortcuts and mouse selections. They can also perform further edits, such as selecting regions for re-synthesis or adding paths in the predictions. Please refer to the supplementary video on our project page for live actions. Since continuous patterns often contain complex structures beyond fine-grained autocomplete of individual strokes [24], we design our current interface to focus on autocomplete pattern regions instead of strokes.



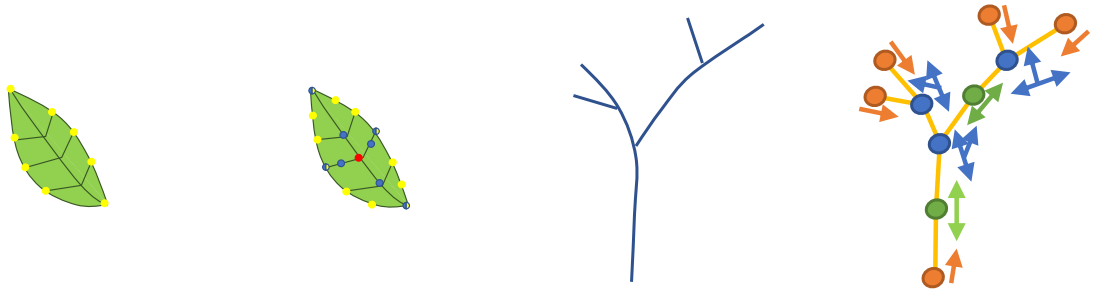
(a) Autocomplete: before (b) Autocomplete: after

(c) Clone: before (d) Clone: after (e) User edit

Figure 2.4: *Autocomplete and clone.* In the autocomplete mode, the user can specify an output region (shown in yellow) (a) and let our system generate predicted patterns (b). In the clone mode, the user can specify a source region (in red) and clone it to a target region (c). Our system can generate predictions adaptive to the existing patterns (d), upon which users can perform further refinements (e). (e) is generated by editing the top left corner (in blue) of the predictions, including 1) partially rejecting several paths, 2) copy-pasting two elements, and 3) adding a path.

2.4 Method

Our method extends the sample-based element texture synthesis method in [1] to consider not only individual point samples but also their curved connections via graphs [17, 18]. We describe our pattern representation in Section 2.4.1, similarity measures in Section 2.4.2, and the corresponding synthesis and reconstruction algorithms in Sections 2.4.3 and 2.4.4. Our method can handle discrete elements, continuous structures, and their combinations. We will describe when and how our algorithms treat them similarly or differently.



(a) Single-res (b) Multi-res (c) Pattern (d) Graph

Figure 2.5: *Pattern representation.* Existing algorithms [1] only use a single-resolution element representation (a). We propose a hierarchical element representation to improve the synthesis quality (b). Our synthesis proceeds from red, blue, to yellow samples in coarse to fine levels. A pattern (c) is represented by a graph (d), where we record connections (yellow edges) and local path orientations (arrows) in addition to point samples. The color of a sample indicates the number of connections $|\boldsymbol{\mathcal{E}}(s)|$ associated with it; orange, green, and blue indicate 1, 2, and 3.

2.4.1 Representation

We represent patterns and elements via point samples and graphs (Figure 2.5) [1, 3, 17]. Each sample s records its position $\mathbf{p}(s)$, attributes $\mathbf{a}(s)$ (Table 2.1), and $i(s) \in [0, 1]$ to indicate the confidence of its existence to optimize the number of samples:

$$\mathbf{u}(s) = (\mathbf{p}(s), \mathbf{a}(s), i(s)) \,. \tag{2.1}$$

Table 2.1: *Sample attributes.* Each sample s records its attributes $\mathbf{a}(s)$ that may vary in terms of types of patterns (discrete element or continuous structure). $q(s) \geq 0$ indicates uniqueness of a sample relative to other samples within a discrete element, and $q(s) = -1$ for continuous structures. $\boldsymbol{\mathcal{E}}(s) = \{e_{ss'}\}$ records all edges associated with s, where $e_{ss'}$ is an edge between s and s'. $\mathbf{o}(s)$ records the local orientations of the paths intersecting at sample s.

Attributes	Sample id	Connectivity	Orientation
$\mathbf{a}(s)$	$q(s)$	$\boldsymbol{\mathcal{E}}(s) = \{e_{ss'}\}$	$\mathbf{o}(s)$

Discrete elements. Following [1, 12], for discrete elements, sample attributes include a sample id $q(s)$ that indicates the uniqueness of s to other samples within its containing element.

Continuous structures. In discrete element synthesis, it is sufficient to use only samples with id $q(s)$ (Figures 2.5a and 2.5b) to encode element shape because every element has the same topology [1]. On the other hand, continuous structures are composed of paths and have more flexibilities. In this chapter, we represent paths by linear and quadratic Bézier curves, even though other vector curve formats can be easily added. The paths may be connected to each other with complex topologies (Figure 4.1). Thus, samples alone are not sufficient to disambiguate matching and reconstruction of continuous structures. Therefore, we also consider connectivity among samples, leading to a graph-based representation (Figure 2.5d). Note the method in [17, 18] also adopts a graph-based representation, but it is for discrete elements only without explicitly modeling paths in a continuous pattern.

Specifically, we record the connectivity $\boldsymbol{\mathcal{E}}(s)$ for s within $\mathbf{a}(s)$. $\boldsymbol{\mathcal{E}}(s) = \{e_{ss'}\}$ is the set of edges associated with s, where $e_{ss'}$ represents the edge between the two samples s and s'. We use $i(e) \in [0\ 1]$ to indicate the confidence of an edge existence; $i(e_{ss'}) = 1/0$ indicates the presence/absence of an edge between s and s'. We will relax the binary $i(e_{ss'})$ to be within the range $[0\ 1]$ during optimization-based synthesis. While $\boldsymbol{\mathcal{E}}(s)$ records pattern topology, we also record the tangent angles at s on a path via an orientation attribute $\mathbf{o} \in \mathcal{R}^{N_\mathbf{o}}$ as part of $\mathbf{a}(s)$, where $N_\mathbf{o}$ is the number of entries in $\mathbf{o}(s)$. Each entry o of $\mathbf{o}$ is within $[0, 2\pi)$. We record $\mathbf{o}$ to facilitate pattern reconstruction from graphs (Section 2.4.4.2 and fig. 2.15). Note that, for any input samples s_i, we always have $|\boldsymbol{\mathcal{E}}(s_i)| = N_\mathbf{o}(s_i)$ (Figure 2.5d), where

$|\boldsymbol{\mathcal{E}}|$ is the size of the edge set $\boldsymbol{\mathcal{E}}$. However, this strict constraint is relaxed for the output during the synthesis to facilitate faster convergence.

2.4.1.1 Hierarchical Pattern Sampling

We adopt a multi-resolution representation of sample graphs to handle patterns with complex structures, analogous to prior multi-resolution algorithms for color texture synthesis [22, 23]. The representation is sparser with less samples at coarser resolutions and becomes denser with more samples at finer resolutions. By default, we use three level of hierarchies, which suffice in our experiments. Users can decide to use fewer levels if needed.

Discrete elements. We generate element samples using a simple approach (Figures 2.5a and 2.5b). The finest level of samples are generated by sampling the element polygon. The coarest level contains only one sample centered at each element. The middle level of samples are located at the midpoints of each downsampled finest-level samples and the coarest-level element centers.

Continuous structures. For continuous patterns (Figure 2.5c), we sample the intersections (blue samples in Figure 2.5d) and ends of paths (orange samples) and uniformly place samples (green samples) along paths with spacing δ. We discuss the parameter δ values in Section 3.7.

2.4.2 Similarity Measure

A core part of pattern synthesis is a measure of similarity between local regions [19, 1, 3]. Here, we describe our similarity measure for continuous patterns via their sample-graph representation (Section 2.4.1), which, in turn, will form the basis for our synthesis optimization (Section 3.6).

2.4.2.1 Sample Similarity

The difference between two samples s and s', which includes the differences in the global position $\mathbf{p}$ and attributes $\mathbf{a}$, is defined as follows:

$$\hat{\mathbf{p}}(s, s') = \mathbf{p}(s) - \mathbf{p}(s'), \tag{2.2}$$

$$\hat{\mathbf{a}}(s, s') = \left(\hat{q}(s, s'), \hat{\boldsymbol{\mathcal{E}}}(s, s')\right) \tag{2.3}$$

The differences in sample id attribute q is

$$\hat{q}(s, s') = \mathbb{1}\left\{q(s) \neq q(s')\right\}, \tag{2.4}$$

where $\mathbb{1}(\cdot)$ is an indicator function that equals to one if its condition $\cdot$ holds, and zero otherwise. The edge set difference is

$$\hat{\boldsymbol{\mathcal{E}}}(s, s') = \left(\sum_{e_{s\hat{s}} \in \boldsymbol{\mathcal{E}}(s)} dist\left(e_{s\hat{s}}, e_{s'\hat{s}'}\right)\right) + \beta \left|\left|\boldsymbol{\mathcal{E}}(s)\right| - \left|\boldsymbol{\mathcal{E}}(s')\right|\right|, \tag{2.5}$$

where $dist\left(e_{s\hat{s}}, e_{s'\hat{s}'}\right) = \|\hat{\mathbf{p}}(s,\hat{s}) - \hat{\mathbf{p}}(s',\hat{s}')\|$ is the difference between $e_{s\hat{s}}$ and $e_{s'\hat{s}'}$, and $e_{s'\hat{s}'} = m_e(e_{s\hat{s}}) \in \boldsymbol{\mathcal{E}}(s')$ is the matching edge for $e_{s\hat{s}}$ via the Hungarian algorithm (which solves the one-to-one matching relationship between edges) [42] to minimize the first term in Equation (2.5) (m indicates matching relationship). β is a weighting parameter set to sampling distance δ (Section 2.4.1.1) in our experiments.

For newly added output samples that do not have any edges, either by initialization or existence assignment (Section 3.6), we want them to be useful and connected to existing output samples. To this end, they should be encouraged (via lower cost) to match with input samples during the search step (Section 2.4.3.3). We set $\beta = 0$ for these samples and thus Equation (2.5) becomes 0, as the first term is also 0 since newly created samples do not have edges.

We do not include $\mathbf{o}$ within the attribute similarity term (Equation (2.3)) since $\boldsymbol{\mathcal{E}}$ already contain similar information in $\mathbf{o}$. However, we still need to update $\mathbf{o}$ during synthesis (assignment step, Section 2.4.3.4) and reconstruction (Section 2.4.4). This requires us to match orientation entries o within $\mathbf{o}$ from s and s' respectively. The matching $o(s') = m_o\left(o(s)\right) \in \mathbf{o}(s')$ is computed via the Hungarian algorithm [42] by minimizing the sum of smallest absolute differences between matched $o(s)$ and $o(s')$

$$\hat{\mathbf{o}}(s,s') = \sum_{o(s)\in\mathbf{o}(s)} \hat{o}(s,s'), \tag{2.6}$$

where $\hat{o}(s,s') = \min\left(|o(s) - m_o\left(o(s)\right)|, 2\pi - |o(s) - m_o\left(o(s)\right)|\right)$.

We also have not found it necessary to include i in the sample similarity measure (Equation (2.3)). Instead, i will be used to optimize the number of samples.

2.4.2.2 Neighborhood Similarity

We define $\mathbf{n}(s)$, the neighborhood of s, as a set of samples around s's spatial vicinity within a certain radius r. The neighborhood similarity is defined as

$$\|\mathbf{n}(s_o) - \mathbf{n}(s_i)\| = \sum_{s'_o \in m_n(\mathbf{n}(s_i))} \hat{\mathbf{u}}_{s_o s_i}(s'_o, s'_i) + \sum_{s'_o \in \mathbf{n}(s_o) \ominus m_n} c(s'_o), \tag{2.7}$$

where

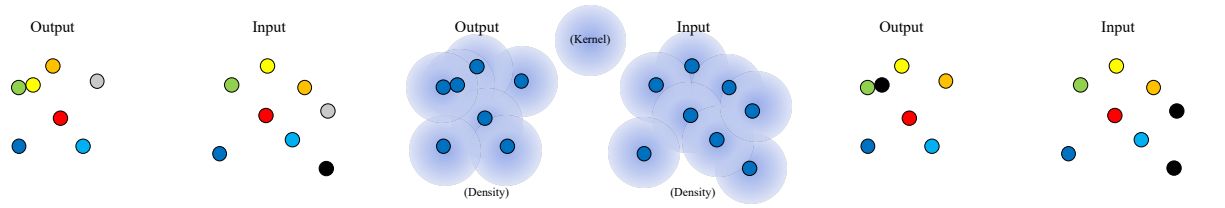
$$\hat{\mathbf{u}}_{s_o s_i}(s'_o, s'_i) = \|\hat{\mathbf{p}}(s_o, s'_o) - \hat{\mathbf{p}}(s_i, s'_i)\| + \gamma \|\hat{\mathbf{a}}(s'_o, s'_i)\| \tag{2.8}$$

is the sample similarity between s'_o and s'_i within neighborhoods centered at s_o and s_i respectively. $s'_i = m_s(s'_o)$ is the matching input sample for s'_o. We discuss how to match samples within $\mathbf{n}(s_o)$ and $\mathbf{n}(s_i)$ in Section 2.4.2.3. The positional differences $\|\hat{\mathbf{p}}(s_o, s'_o) - \hat{\mathbf{p}}(s_i, s'_i)\|$ are computed in local neighborhood coordinate systems centered at s_o or s_i. The two terms in Equation (2.7) partition $\mathbf{n}(s_o)$ into two sets. In the first term of Equation (2.7), $m_n(\mathbf{n}(s_i))$ is the subset of $\mathbf{n}(s_o)$ matched with samples within $\mathbf{n}(s_i)$. In the second term of Equation (2.7), $c(s'_o)$ is the cost resulting from unmatched output samples s'_o. In our implementation, $\gamma = 0.5$. Equation (2.7) is designed for our robust neighborhood matching, described next.

2.4.2.3 Robust Neighborhood Matching

In [1], each output sample is forced to match with another sample in the input, which could be problematic since some output samples are outliers and should not be matched to any input samples. Some output samples might be missing in the current iteration of

optimization. But this forced matching allows to easily define sample similarity for various sample attributes, as shown in Section 2.4.2.1, since we have one-to-one sample correspondence. We call this *hard neighborhood matching* (Figure 2.6a). In [3], the neighborhoods are matched via comparing their density fields estimated with Gaussian kernels. This similarity criterion is computed with the neighborhoods as a whole. *There is no one-to-one correspondence between samples.* We call it *soft neighborhood matching* (Figure 2.6b). The method [3] "smears the sample attributes into their neighborhood" by encoding them as the height of the density kernel, which could unnecessarily couple the position and attribute information. It is not easy to integrate soft matching with various sample attributes, which can include edges.



(a) Hard neighborhood matching (b) Soft neighborhood matching (c) Robust neighborhood matching

Figure 2.6: *Different neighborhood matching methods.* In (a) each sample in the output (left) is forced to match with another one in the input (right), which could be problematic since some output samples are outliers and should not be matched to any input samples, and some output samples might be missing in the current iteration of optimization. Matched samples are with the same color. Black indicates unmatched. In (b) samples are not explicitly matched but the neighborhood is matched as a whole, by transforming the samples with density kernels [3]. There is no one-to-one correspondence between samples, and thus it is not easy to integrate with various sample attributes. Instead, we propose robust neighborhood matching (c) to generate high-quality sample distributions to accommodate various types of patterns. We allow output samples to be unmatched if it can result in high matching cost increase. In the above example, the yellow output sample in (a) destroys the hard neighborhood matching and forces other (orange, and gray) samples to be less matched, while our robust matching leaves the bad sample unmatched.

Instead, we propose to use a robust neighborhood matching that explicitly considers

outliers in the output to address these issues (Figure 2.6c). An output sample is either matched with an input sample or unmatched as an outlier with additional cost c. We apply the Hungarian algorithm to compute the matchings between input $\mathbf{n}(s_i)$ and output neighborhoods $\mathbf{n}(s_o)$. The input of the Hungarian algorithm is a cost matrix where each entry indicates the matching cost between an output and an input sample. Inspired by [43], we define our cost matrix $\mathcal{C} \in \mathbb{R}^{N_{\mathbf{n}o} \times (N_{\mathbf{n}i} + N_{\mathbf{n}o})}$ as:

$$\mathcal{C} = \left[\begin{array}{c|c} \mathcal{C}_m & \mathcal{C}_u \end{array} \right] \tag{2.9}$$

$$\mathcal{C}_m = \begin{bmatrix} \hat{\mathbf{u}}_{s_o s_i}(s_o'^1, s_i'^1) & \hat{\mathbf{u}}_{s_o s_i}(s_o'^1, s_i'^2) & \cdots & \hat{\mathbf{u}}_{s_o s_i}(s_o'^1, s_i'^{N_{\mathbf{n}i}}) \\ \hat{\mathbf{u}}_{s_o s_i}(s_o'^2, s_i'^1) & \hat{\mathbf{u}}_{s_o s_i}(s_o'^2, s_i'^2) & \cdots & \hat{\mathbf{u}}_{s_o s_i}(s_o'^2, s_i'^{N_{\mathbf{n}i}}) \\ \vdots & \vdots & \ddots & \vdots \\ \hat{\mathbf{u}}_{s_o s_i}(s_o'^{N_{\mathbf{n}o}}, s_i'^1) & \hat{\mathbf{u}}_{s_o s_i}(s_o'^{N_{\mathbf{n}o}}, s_i'^2) & \cdots & \hat{\mathbf{u}}_{s_o s_i}(s_o'^{N_{\mathbf{n}o}}, s_i'^{N_{\mathbf{n}i}}) \end{bmatrix} \tag{2.10}$$

$$\mathcal{C}_u = \begin{bmatrix} c_1 & c_1 & \cdots & c_1 \\ c_2 & c_2 & \cdots & c_2 \\ \vdots & \vdots & \vdots & \vdots \\ c_{N_{\mathbf{n}o}} & c_{N_{\mathbf{n}o}} & \cdots & c_{N_{\mathbf{n}o}} \end{bmatrix}, \tag{2.11}$$

where the superscripts of s_o' or s_i' represent the index of a sample within $\mathbf{n}(s_o)$ or $\mathbf{n}(s_i)$. There are $N_{\mathbf{n}i}$ and $N_{\mathbf{n}o}$ samples within $\mathbf{n}(s_i)$ and $\mathbf{n}(s_o)$, respectively. In [12], the sample matching is computed via only the $\mathcal{C}_m$ part of $\mathcal{C}$, in which case every output sample should be matched. Our cost matrix is augmented with the $\mathcal{C}_u$ side, where each entry represents the cost of unmatched outliers in $\mathbf{n}(s_o)$. We make sure there are enough samples in the input neighborhood so that an output sample would not be matched only when it is an

outlier that would result in a high cost increase in matching. For the same reason, we do not take missing output samples into account in the cost matrix formulation.

In our implementation, c_k is set as $\min(2, 1.2 + 0.4|\boldsymbol{\mathcal{E}}(s_o^k)|)\delta$ if the output sample s_o^k is from a continuous pattern, and $1.5\times$ average nearest neighbor distance if s_o^k is from a discrete element. In an interactive system, we may synthesize predictions near the provided exemplars. If s_o^k is from the exemplars, $c_k = \infty$ because none of the samples from the exemplars are outliers and all of them should be matched.

In a neighborhood, there might be samples from both discrete elements and continuous structures. We only match samples from the same type of patterns and with the same id, i.e. continuous structures only match with continuous structures (negative id), and discrete elements only match the same discrete elements and their samples with the same (non-negative) ids.

2.4.3 Pattern Synthesis

Based on our pattern representation (Section 2.4.1) and similarity measures (Section 2.4.2), we now describe how to synthesize an output similar to a given input.

2.4.3.1 Optimization Objective

We synthesize output predictions $\mathcal{O}$ via optimizing the following objective:

$$E(\mathcal{O}) = \sum_{s_i = m(s_o), s_o \in \mathcal{O}} \|\mathbf{n}(s_o) - \mathbf{n}(s_i)\| + \Theta(\mathcal{O}, \mathcal{D}). \tag{2.12}$$

where s_o is matched with s_i. This energy sums up the similarity between every $\mathbf{n}(s_o)$ in $\mathcal{O}$ and its most similar $\mathbf{n}(s_i)$ via Equation (2.7). $\Theta(\mathcal{O}, \mathcal{D})$ is the domain constraint term [44] to encourage the synthesized samples to stay within the user-specified domain $\mathcal{D}$.

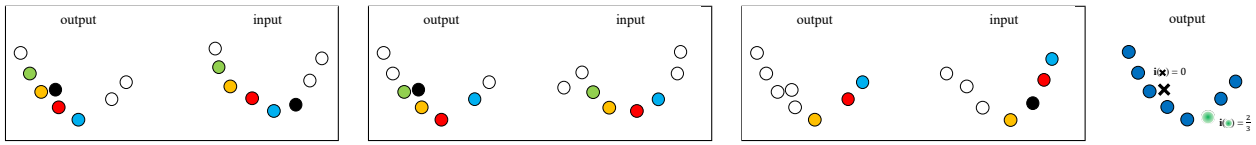
The pattern optimization framework adopts an EM-like strategy to minimize Equation (2.12), by iterating the search and assignment steps as detailed below.

2.4.3.2 Initialization

Similar to prior patch-based texture synthesis methods [45, 46], we copy new patches one-by-one with similar boundary patterns to existing patches for initialization. Each next patch is selected to ensure high similarity (as evaluated by Equation (2.7)) in the overlapped boundary regions with existing patches. In the overlapping regions, we only copy samples unmatched with any sample in the existing patches. We make sure that the initialized samples are within the output domain $\mathcal{D}$ by removing samples outside it. We copy discrete elements in wholes like [1]. In Section 2.5, we will show the robustness of our method to random sample initializations (Figure 2.14). But patch-based initialization makes the algorithm converge faster, contributing to the responsiveness of the interface. For simplicity, we do not copy edges in the initialization step.

2.4.3.3 Search Step

We adopt PatchMatch [47, 48] to compute approximate nearest neighbors (ANN) for each output sample. The standard PatchMatch algorithm 1) randomly generates the initial nearest neighbor field, and 2) alternates between propagation and search steps by

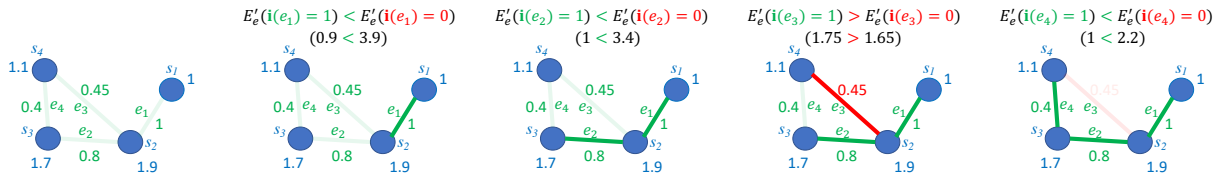


(a) Matched neighbor. pair 1 (b) Matched neighbor. pair 2 (c) Matched neighbor. pair 3 (d) Assign existence

Figure 2.7: *Existence assignment example.* We visualize how to compute confidences of existence of output samples $i(s_o)$ (Equation (2.15) and algorithm 1) from a set of matched input and output neighborhoods. (a) (b) (c) show three pairs of matched input and output neighborhoods centered at different samples (shown in red) over the same set of samples. Matched samples are in the same color. Empty black circles indicate samples outside a neighborhood. Solid black circles indicate samples within a neighborhood but unmatched. The black cross sample in (d) has $i(s_o) = 0$, since it is unmatched with any s_i ($i(s_i) = 0$) in (a) and (b). The green sample in (d) has confidence $i(s_o) = \frac{2}{3}$: in the three pairs of neighborhoods, there are two pairs (a) and (c) where each has an unmatched input sample, which indicates there could be a missing sample in the output located at approximately the same location relative to its neighborhood center; the two unmatched input samples are merged to generate the green output sample in (d).

traversing the regular image grid in a scanline order. Initially, we generate the ANN by randomly assigning an output sample to an input sample (with identical sample id for discrete elements). One issue is how to choose a sample traversal order. We follow the steps from [48] which works on meshes. We build a simple graph by connecting each sample with its k-nearest neighbors ($k = 8$ in our implemenetation), and perform breadth-first search. In the next iteration, the traversal starts from the last sample in the most recent sequence.

In our implementation, for the random search step, the maximum window size is 150, and the minimum size is 25, and the search window is exponentially decreased with factor 2. In each pattern optimization step, we need to compute an ANN. In two consecutive steps, the output sample distributions are similar. So the previous ANN is used to initialize the subsequent patch match algorithm. Since the initialization is close to the converged ANN, a small number (2) of Patch Match iterations is used, except for the initial step at



(a) Initialization (b) Step 1 (c) Step 2 (d) Step 3 (e) Step 4

Figure 2.8: *Edge assignment example.* This example illustrates how to solve Equation (2.16) for edge assignment by solving Equation (2.21) in a loop. Each step (b)-(e) solves Equation (2.21) once. There are four samples and four potential edges (with $\bar{i}(e) > 0$). The numbers near edges or samples indicate the expected confidence of existence $\bar{i}(e)$ of the edges or the expected number of edges $|\overline{\boldsymbol{\mathcal{E}}}(s)|$ associated with samples (e.g. $\bar{i}(e_2) = 0.8$, $|\overline{\boldsymbol{\mathcal{E}}}(s_2)| = 1.9$). In (a), we initialize all $i(e) = 0$ and sort all potential edges by its $\bar{i}$ from the largest to the smallest: e_1 is the one with the largest $\bar{i}$ and e_4 is the one with the smallest. The optimization loops from e_1 (b) to e_4 (e) in decreasing $\bar{i}(e)$ values. The number below each $E'_e(i(e))$ is its computed value (e.g. $E'_e(i(e_1) = 1) = |i(e_1) - \bar{i}(e_1)| + |\boldsymbol{\mathcal{E}}(s_1) - \overline{\boldsymbol{\mathcal{E}}}(s_1)| + |\boldsymbol{\mathcal{E}}(s_2) - \overline{\boldsymbol{\mathcal{E}}}(s_2)| = |1 - 1| + |1 - 1| + |1 - 1.9| = 0.9$). The light green edges are not optimized with initial values $i(e) = 0$. The dark green edges are optimized with $i(e) = 1$. The red edge is optimized with $i(e) = 0$. The light red indicates there is no edge after optimization.

each level of hierarchical synthesis (Section 2.4.3.5), which uses 5 iterations. Our patch match implementation is parallelized by equally dividing the output domain into regions, the number of which equals to that of threads. The search step consumes most of the computation time needed by the synthesis. The computational complexity of the search step in an optimization step is $O(n_{\mathcal{O}} N_{\mathbf{n}}^3)$, where $n_{\mathcal{O}}$ is the total number of output samples and $N_{\mathbf{n}}$ is the average number of samples within neighborhoods. Please refer to Section 5.3 for more details.

2.4.3.4 Assignment Step

Here, we describe how to determine the values of sample positions $\mathbf{p}$, attributes including edge $\boldsymbol{\mathcal{E}}$ and orientation $\mathbf{o}$, as well as sample existence i. The assignments of these different quantities are extended from the assignment step of pixel colors [49] and sample

positions [1] by taking votes from overlapping output neighborhoods at the same entity (such as sample or edge). In particular, discrete samples only have sample id attributes q, which is used in the search step to make sure only samples with the same q are matched. Thus, only position assignment is deployed for discrete samples.

Position assignment. For each output sample s_o and its neighbor s'_o, there is a set of matched input sample pairs (s_i, s'_i) provided by the previous search step. The estimated distance $\hat{\mathbf{p}}(s_o, s'_o)$ between s_o and s'_o is

$$\hat{\mathbf{p}}(s_o, s'_o) \approx \mathbf{p}(s_i) - \mathbf{p}(s'_i). \tag{2.13}$$

We use least squares [1] to estimate $\mathbf{p}(s_o)$ by

$$\underset{\{\mathbf{p}(s_o)\}}{\arg\min} \sum_{s_o \in \mathcal{O}} \sum_{s'_o \in \mathbf{n}(s_o)} \left\| \hat{\mathbf{p}}(s_o, s'_o) - (\mathbf{p}(s_i) - \mathbf{p}(s'_i)) \right\|^2 + \sum_{s_o \notin \mathcal{D}} \left\| \hat{\mathbf{p}}(s_o, \mathcal{D}) \right\|^2. \tag{2.14}$$

The second term in Equation (6.1) is the domain constraint to encourage output samples to stay within $\mathcal{D}$, where $\hat{\mathbf{p}}(s_o, \mathcal{D})$ is the shortest vector from s_o to the boundary of $\mathcal{D}$, and $s_o \notin \mathcal{D}$ indicates s_o is outside $\mathcal{D}$.

Existence assignment. Our method adjusts the number of samples within local regions during the synthesis process for better quality. The number of samples is optimized via existence i assignment, again via a voting scheme:

$$\underset{i(s_o) \in [0\ \ 1]}{\arg\min} \sum_{s_i \in \{s_i\}} \left| i(s_o) - i(s_i) \right|^2, \tag{2.15}$$

where s_i runs through the set $\{s_i\}$ we collect during neighborhood matching, i.e. the corresponding input samples of an output sample from overlapping input neighborhoods. $i(s_i) = 0$ if s_o is not matched with any s_i in a pair of matched input and output neighborhoods, and $i(s_i) = 1$ otherwise. Equation (2.15) computes the confidence of existence $i(s_o) \in [0\ 1]$ of an output sample s_o. Every iteration , we remove output samples s_o whose $i(s_o) < 0.5$. The above assignment step is applied to samples that are already in the output sample distribution. To add back missing output samples, we first generate candidate samples, merge them as output samples, and pick those with $i(s_o) > 0.5$ as added output samples. The energy in Equation (2.15) is not guaranteed to decrease immediately after sample addition or removal, but it will generally decrease through iterations. Please refer to Figure 2.7 for an example and Section 5.1 for more algorithm details.

Edge assignment. We assign edges by optimizing the following objective:

$$\underset{\{i(e_{s_o s'_o})\in\{0,1\}\}}{\arg\min} \sum_{\{e_{s_o s'_o}\}} \left| i(e_{s_o s'_o}) - \bar{i}(e_{s_o s'_o}) \right| + \sum_{\{s_o\}} \left| |\boldsymbol{\mathcal{E}}(s_o)| - |\overline{\boldsymbol{\mathcal{E}}}(s_o)| \right|, \tag{2.16}$$

where the first term computes the difference between the actual and expected edge confidences $\bar{i}(e_{s_o s'_o}) \in [0\ 1]$. $\bar{i}$ is the vote by overlapping input neighborhoods on the same edge, computed using least squares by replacing samples in Equation (2.15) with edges. $\{e_{s_o s'_o}\}$ is the set of edges that have $\bar{i}(e_{s_o s'_o}) > 0$, and there is no edge between s_o and s'_o if $\bar{i}(e_{s_o s'_o}) = 0$. The second term computes the differences between the optimized number of edges $|\boldsymbol{\mathcal{E}}(s_o)|$ and the expected number of edges $|\overline{\boldsymbol{\mathcal{E}}}(s_o)|$ connected to s_o. $|\overline{\boldsymbol{\mathcal{E}}}(s_o)|$ is similarly computed

by voting from overlapping output neighborhoods on the same sample s_o:

$$\arg\min_{|\overline{\boldsymbol{\mathcal{E}}}(s_o)|} \sum_{s_i \in \{s_i\}} \left| |\overline{\boldsymbol{\mathcal{E}}}(s_o)| - |\boldsymbol{\mathcal{E}}(s_i)| \right|^2. \tag{2.17}$$

Basically, we compute the average of $\{|\boldsymbol{\mathcal{E}}(s_i)|\}$. In sum, the first term is edge-centric while the second is sample-centric.

It is non-trivial to optimize Equation (2.16), where the optimization variables $\{i(e_{s_o s'_o})\}$ are binary. Thus, we solve it in a greedy fashion. We initialize all $i(e_{s_o s'_o}) = 0$. We sort output edges $\{e_{s_o s'_o}\}$ by its expected confidence of existence $\bar{i}(e_{s_o s'_o})$, and optimize $i(e_{s_o s'_o})$ greedily by looping over the sorted $\{e_{s_o s'_o}\}$ in decreasing confidence. For each $e_{s_o s'_o}$, we decide whether $i(e_{s_o s'_o}) = 0$ or 1 by choosing the one that minimizes Equation (2.16). In other words, the multivariate optimization problem (Equation (2.16)) is optimized by solving univariate optimization problems in a loop. By decomposing the optimization variables in Equation (2.16) from a set of edges $\{i(e_{s_o s'_o})\}$ to a single edge $i(e_{s^*_o s^{*\prime}_o})$ to be optimized and the rest, the univariate version of Equation (2.16) can be written as:

$$\arg\min_{i(e_{s^*_o s^{*\prime}_o}) \in \{0,1\}} E'_e + E''_e, \tag{2.18}$$

where

$$E'_e\left(i(e_{s_o^* s_o^{*\prime}})\right) = \left|i(e_{s_o^* s_o^{*\prime}}) - \bar{i}(e_{s_o^* s_o^{*\prime}})\right| + \left|\left|\boldsymbol{\mathcal{E}}(s_o^*)\right| - \left|\overline{\boldsymbol{\mathcal{E}}}(s_o^*)\right|\right| + \left|\left|\boldsymbol{\mathcal{E}}(s_o^{*\prime})\right| - \left|\overline{\boldsymbol{\mathcal{E}}}(s_o^{*\prime})\right|\right|, \tag{2.19}$$

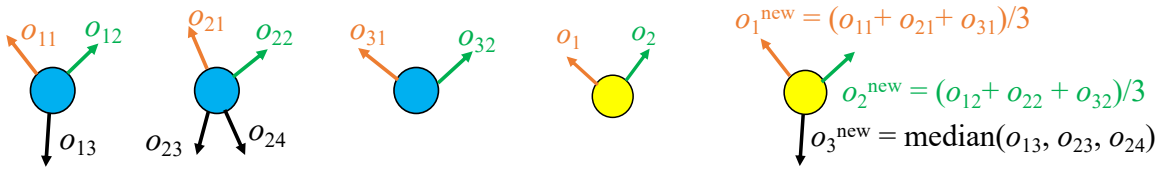
$$E''_e\left(i(e_{s_o^* s_o^{*\prime}})\right) = \sum_{\{e_{s_o s'_o}\} \ominus e_{s_o^* s_o^{*\prime}}} \left|i(e_{s_o s'_o}) - \bar{i}(e_{s_o s'_o})\right| + \sum_{\{s_o\} \ominus \{s_o^*, s_o^{*\prime}\}} \left|\left|\boldsymbol{\mathcal{E}}(s_o)\right| - \left|\overline{\boldsymbol{\mathcal{E}}}(s_o)\right|\right|. \tag{2.20}$$

Since E''_e is a constant in Equation (2.18), it is equivalent to:

$$\underset{i(e_{s_o^* s_o^{*\prime}}) \in \{0,1\}}{\arg\min} \; E'_e. \tag{2.21}$$

Equation (2.21) can be solved with brute-force search. The search space is 2 ($\{0, 1\}$).

Figure 2.8 illustrates how to solve Equation (2.16) by solving Equation (2.21) in a loop.



(a) $\{\mathbf{o}(s_i)\}$ (b) $\mathbf{o}(s_o)$ (c) Updated (b)

Figure 2.9: *Orientation assignment example.* This example illustrates the orientation assignment step when $N_{\mathbf{o}}(s_o)$ is increased from 2 to 3. The output sample (b) is matched with the three input samples (a). Matched orientations o (arrows) are visualized in the same colors. Unmatched inputs o are in black. (c) shows the updated orientations of the output sample. The orange o_1^{new} and green orientations o_2^{new} are updated by averaging matched input orientations. The black output orientation o_3^{new} is newly added by choosing the median from three unmatched orientations (o_{13},o_{23},o_{24}) in (a).

Orientation assignment. In the search step (Section 2.4.3.3), each $\mathbf{o}(s_o)$ is matched with a set of $\{\mathbf{o}(s_i)\}$ associated with input samples coming from different input neighborhoods, and each entry $o(s_o) \in \mathbf{o}(s_o)$ has been matched with a $o(s_i) \in \mathbf{o}(s_i)$. The local orientation attribute $\mathbf{o}(s_o)$ is updated by a voting scheme among $\{\mathbf{o}(s_i)\}$, where $\{\mathbf{o}(s_i)\}$ could have different lengths across different s_i.

We optimize both dimension $N_{\mathbf{o}}$ and value of entries o in order. In the input exemplar, the number of orientation entries $N_{\mathbf{o}}(s_i)$ equals $|\boldsymbol{\mathcal{E}}(s_i)|$. Thus $N_{\mathbf{o}}$ can be computed like in Equation (2.17) and rounding the result as integers. Essentially, we are trying to find an integer $N_{\mathbf{o}}(s_o)$ that is the closest to the arithmetic average of $\{N_{\mathbf{o}}(s_i)\}$.

Similarly, we can update the values $o(s_o)$ in $\mathbf{o}(s_o)$ using the same voting scheme to Equations (2.15) and (2.17). A special case is when $N_{\mathbf{o}}(s_o)$ is updated to a new value (changing $\mathbf{o}(s_o)$ vector length). In this case, we will need to add or remove one or several entries to or from the original $\mathbf{o}(s_o)$. To remove an entry from $\mathbf{o}(s_o)$, we pick the one whose matched set of input votes $\{o(s_i)\}$ has the largest variance. (We have experimented with another strategy that removes $o(s_o)$ whose matched set of input votes $\{o(s_i)\}$ has the least number of entries, but have not found visible differences to the maximum variance strategy above.) To add an entry to $\mathbf{o}(s_o)$, we collect orientation entries $\{o'(s_i)\}$ from the input samples that remain unmatched to any orientation entries $\mathbf{o}(s_o)$ of the matched output sample, and add a new entry $o(s_o)$ into $\mathbf{o}(s_o)$ as the median from the unmatched set $\{o'(s_i)\}$. An example is illustrated in Figure 2.9. In the rare case where we need to add more than one entry to $\mathbf{o}(s_o)$, we randomly choose from $\{o'(s_i)\}$ after the median is used for the first add-on.

2.4.3.5 Hierarchical Synthesis

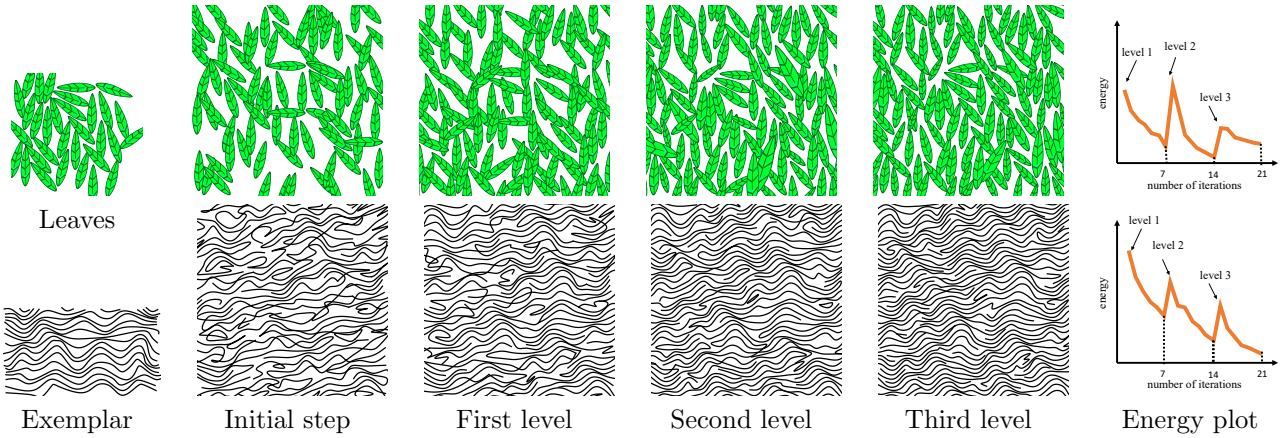


Figure 2.10: *Hierarchical synthesis visualization.* The hierarchical synthesis proceeds from the coarsest (first level) to the finest levels (third level), with upsampling sample representations (Figure 2.5b). The hierarchical synthesis gradually refines the pattern from large to small scale structures. Note that the initialization has fewer elements than required (top row), hence our existence optimization adaptively controls the total number of elements.

Instead of using a single-resolution representation [1], we apply a hierarchical representation (Section 2.4.1.1) for multi-resolution synthesis. We first synthesize the predictions at a coarse level using sparse representation, and then reconstruct the patterns based on sparse samples. We continue this process with a denser and denser pattern representation. Figure 2.5b shows an example of multi-resolution element representation. For continuous structures, the sampling distance δ of continuous pattern is gradually increasing with respect to the level of hierarchy. During synthesis, we use multi-scale neighborhood sizes to keep both large and local structures. The neighborhood size is gradually reduced at different hierarchies. In our implementation, at each hierarchy, there are 7 search-assignment iterations. See Figure 2.10 for an example.

2.4.4 Pattern Reconstruction

The reconstruction step takes a synthesized pattern representation as input to generate output patterns that may consist of discrete elements and continuous structures.

2.4.4.1 Discrete Elements

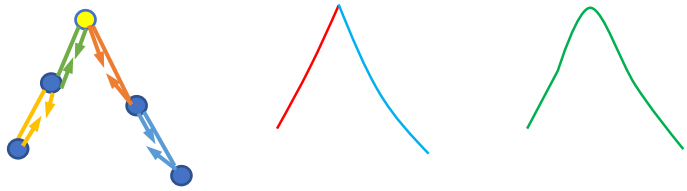
For discrete elements, each sample is uniquely associated with an element. The reconstruction is to transform the element shapes by treating samples as control points. Specifically, we assume similarity transform to reconstruct the elements.

2.4.4.2 Continuous Structures

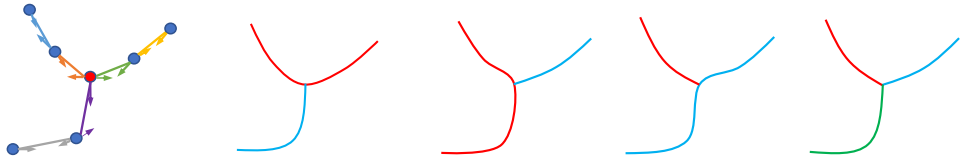
We have synthesized a graph whose sample positions and edge connections represent the topology of the output (Section 3.6). However, these edges are piecewise linear, and they thus capture only connectivity/topology, but not shape/geometry information. The original continuous patterns can be composed of smooth paths (e.g. quadratic Bézier curves). Therefore, we need to reconstruct paths from the graph samples and edges. The samples are used as control points of Bézier curves.

Next, we talk about how to identify which sample and which edge are included within which path. This process relies on the synthesized sample orientation attributes $\mathbf{o}$.

Samples with only one neighbor are unambiguous and thus only included within one path. Samples with only two neighbors could be included in one path (all blue samples with two neighbors in Figure 2.11) as path samples, or two paths as junction samples (e.g. the yellow sample in Figure 2.11a). Samples with more than two neighbors are junction



(a) 2-neighbor s (b) 2 paths from (a) (c) 1 path from (a)



(d) >2 neighbor s (e) Recon from (d) (f) Other possible reconstructions from (d)

Figure 2.11: *Curve reconstruction from a graph using orientation attributes.* If we only consider the different statuses of the yellow sample in (a), there are two possible reconstructions (b)(c), depending on whether it is a junction (b) or path (c) sample. If we only consider the different statuses of the red sample in (d), there are four possible reconstructions (e)(f), depending on the red sample is included within which two or three paths. For the yellow (in (a)) and red samples (in (d)), we decide the reconstruction by examining its associated local orientation attribute and the fact a pair of local orientations of a sample should be opposite if the sample is included within the path. Our algorithm will reconstruct (b) from (a) and (e) from (d).

samples (e.g. the red sample in Figure 2.11d) that are included in multiple potential paths.

To disambiguate these cases, we examine a sample's local orientations $\mathbf{o}(s)$. Since $o \in \mathbf{o}(s)$ should be tangent to the sample's local path, if a sample s has a pair of orientations $o(s)$ that are almost opposite ($8\pi/9 <$ absolute orientation difference $< 10\pi/9$), it will suggest that the sample is included inside a path as opposed to at the ends of a path. Therefore, there are three steps to reconstruct a pattern without ambiguity. First, we identify pairs of local path orientations $o(s)$ (if any) that are opposite (e.g., a pair of arrows associated with 2-neighbor blue samples in Figures 2.11a and 2.11d, or the orange and green ones associated with the red junction sample in Figure 2.11d). Second, we match local orientations $o(s)$ with edges $e \in \boldsymbol{\mathcal{E}}(s)$ connected to the sample using the Hungarian

algorithm by minimizing the sum of absolute difference between local orientation and edge angles. The arrows and graph edges in Figures 2.11a and 2.11d with the same colors are matched. Third, we generate a path by including edges that are connected together and matched with opposite orientations. A Bézier curve is generated by interpolating samples along a path. This reconstruction strategy using $\mathbf{o}$ can help preserving the original curve shapes, as demonstrated in Figure 2.15.

2.5 Evaluation

We evaluate our method with sample results, ablation studies, and comparisons with existing art. We will make our code repository [50] public to facilitate future research.

2.5.1 Results

Our method can automatically synthesize satisfactory results for a variety of patterns without user intervention, as exemplified in Figures 2.12, 2.13 and 4.1 and our (full) results in Figures 2.2, 2.10, 2.14, 2.15 and 3.14. However, like existing techniques, our method might not always produce what users would like to have, and some artifacts can be visible in local regions (such as unfinished or dangling components in Figure 2.1a and Figure 2.2g or inconsistent curvatures in Figure 2.10k bottom) and global structures (such as the regular and warped grids in Figures 2.12h and 2.12p, the rectangular blocks in Figure 2.12l, and the straight lines in Figures 2.1e and 2.12t). For further quality improvement and customization, users can also interactively edit the system suggestions via our system interface, as demonstrated in Figure 2.13. Unless otherwise noted, all our results

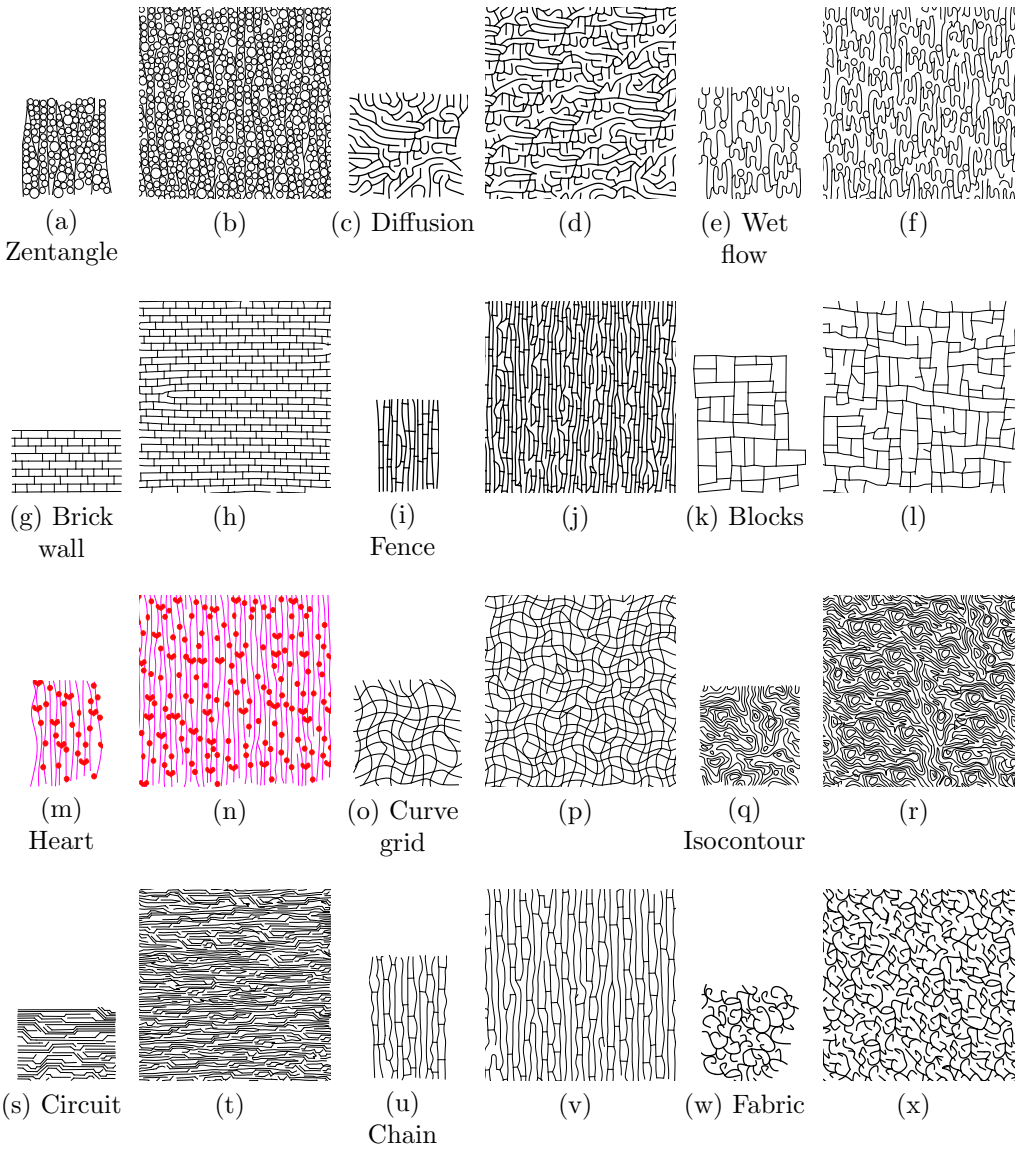


Figure 2.12: *Automatic synthesis results by our method.* Within each pair of images, the input exemplar is smaller and shown on the left, the automatic synthesis result is bigger and shown on the right.

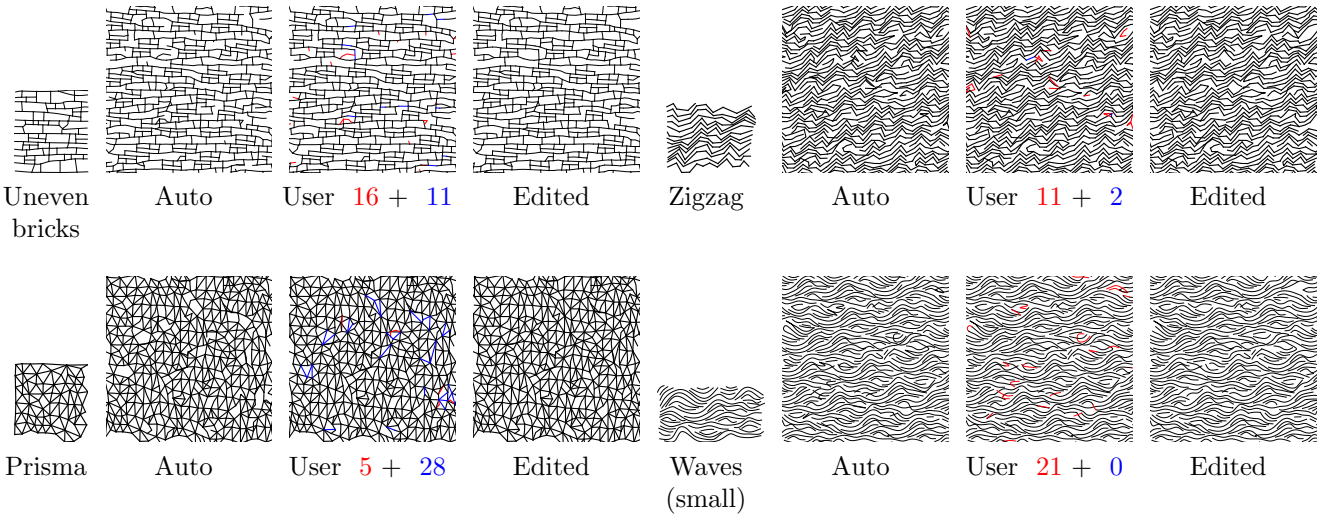


Figure 2.13: *Automatic synthesis and user-assisted results.* We count the number of user operations needed for correcting artifacts. Red indicates the number of rejection and blue the number of manual path drawing.

are produced with three hierarchies using neigborhood radii $r \in \{60, 50, 40\}$ with sampling distance $\delta \in \{40, 30, 25\}$, while the longer side of bounding box of exemplars are varying between 250 and 500. Our method is robust to variations of neighborhood radii. See Section 5.2 for more details about our parameter settings.

2.5.2 Ablation Study

Although we use patch-based methods for initialization in our implementation, our algorithm is robust to different initial conditions (Figure 2.14), even if the initial sample distribution is randomly distributed (white noise). Figure 2.15 is the ablation study for the orientation attribute $\mathbf{o}$. Figure 3.14 shows other components of our algorithm. Without the edge term (Equation (2.5)) or robust matching (Section 2.4.2.3) in the search step, our algorithm produces lower quality results with obvious artifacts. Without existence assignment (the third paragraph in Section 2.4.3.4), the algorithm cannot automatically

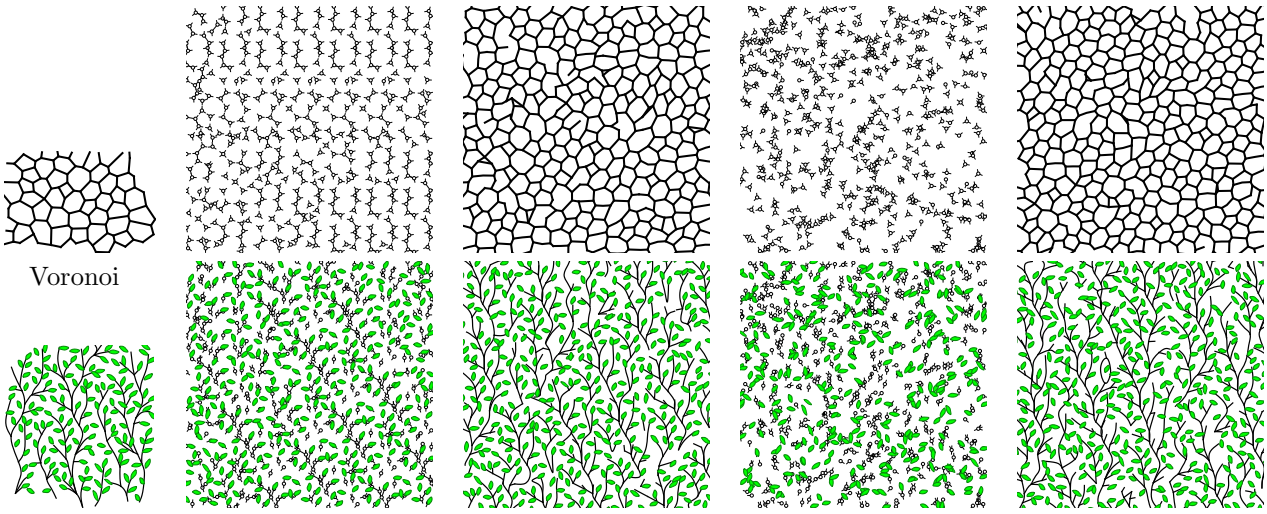


Figure 2.14: *Algorithm robustness to different initialization.* Our algorithm can generate similar results with both patch-based or random initialization.

adjust the number of samples within local regions and can produce empty space or extra broken curves.

2.5.3 Comparison to Previous Methods

To our knowledge, there is no previous example-based method that can generate the types of patterns we target. The sample-based methods in [1, 3, 4] are the most related. We compare against [1, 3] and a state-of-art point distribution synthesis method in [4] which applies convolutional neural networks to preserve both local and global structures. As shown in Figure 2.2, our method can produce better spatial sample distributions than [1, 3, 4]. Note that we compare only sample distributions in Figure 2.2 since it is unclear how to reconstruct continuous curve patterns from synthesized samples without connectivity [1, 3, 4].

We also *enhance* [1] for comparisons, by incorporating it with the sample connectiv-

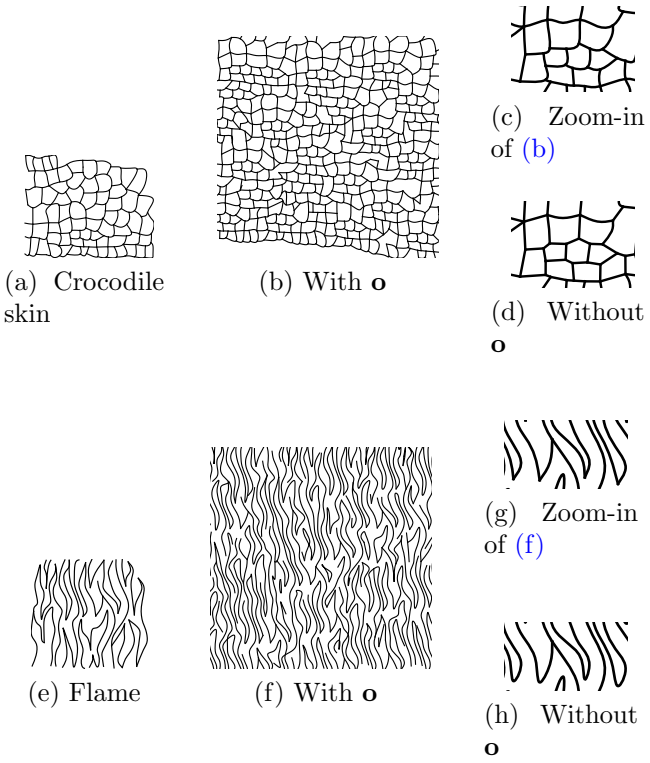


Figure 2.15: *Ablation study for* **o**. The orientation attribute **o** is useful to faithfully recover curve appearance in the exemplars. In the "crocodile skin" example, the curves should be smooth at junction; in the "flame" example, the curves should be sharp at the flame tip.

ity (Figure 2.5d) and edge assignment step (Equation (2.16)), but without the edge set difference (Equation (2.5)) and robust matching (Section 2.4.2.3) in the search step, as well as without the existence assignment step (the third paragraph in Section 2.4.3.4). As shown in Figure 2.17, our method can generate better results than the enhanced version of [1]. Unlike for [1], we are unable to enhance [4, 3] due to the lack of one-to-one sample correspondences which are needed for the edge assignment step.

2.6 Conclusions, Limitations, and Future Work

Repetitive patterns have many applications, whose creation has been a main focus of research in computer graphics and interactive techniques. This work focuses on methods

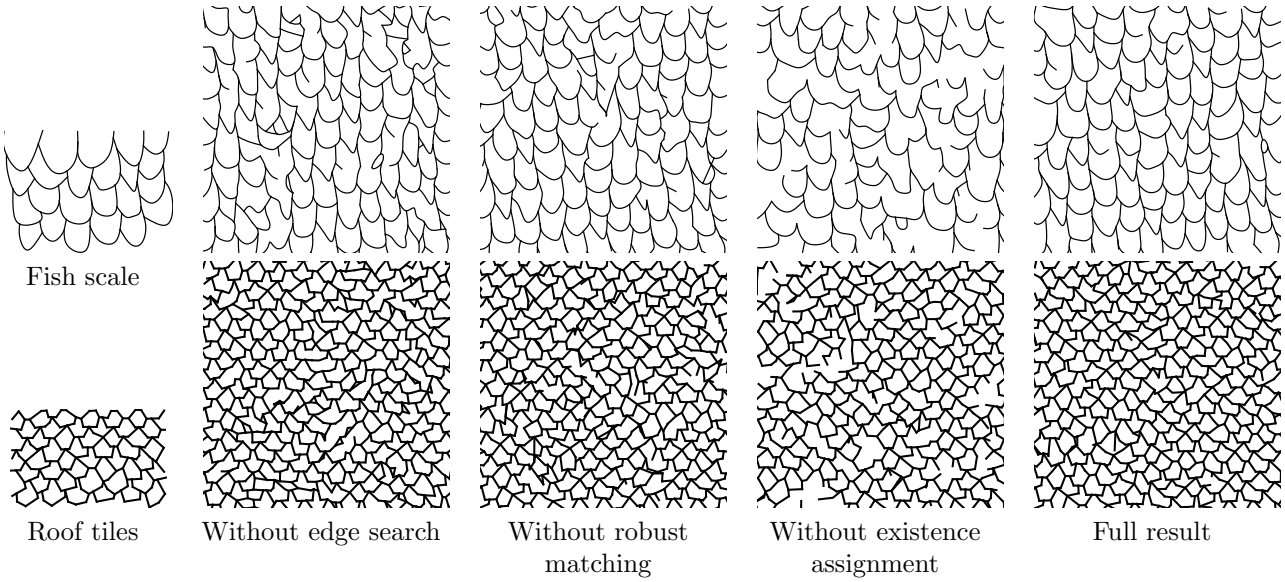


Figure 2.16: *Ablation study.* Without using edges (Equation (2.5)) in the search step (the second column) or robust matching (Section 2.4.2.3) that considers outliers (the third column), the algorithm produces lower quality results. Without existence assignment (the third paragraph in Section 2.4.3.4), the algorithm produces broken curves and empty space due to outliers and missing samples (the forth column). Our results are shown in the last column.

and interfaces to help users author continuous curve patterns. Analysis and results of diverse patterns have demonstrated the promise of our approach.

Like other neighborhood-based texture/pattern synthesis methods, our algorithm also assumes local properties and thus cannot capture global structures and may introduce stochastic variations, such as broken and distorted curves, as shown in Section 3.7. These artifacts can be reduced by other improvements, such as bidirectional similarity, additional feature masks, and smart initialization [51].

Our current reconstruction algorithm is based on Bézier curve interpolation, which might not preserve the exemplar curves. One possibility is to treat each curve segment like a discrete element and reconstruct via sample-based warping [1, 18], while ensuring that curve segments sharing common samples are well connected.

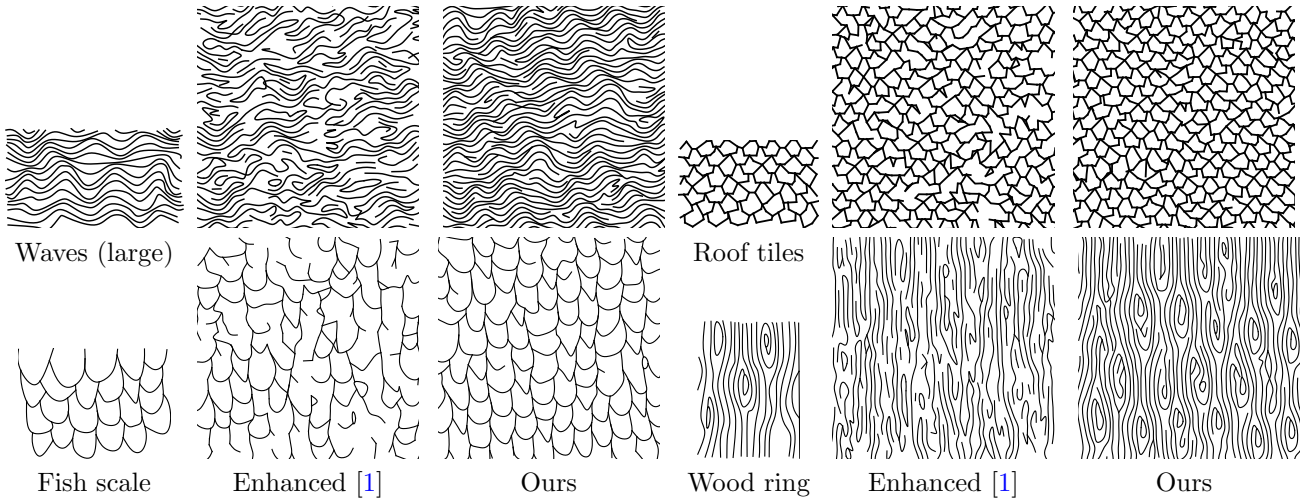
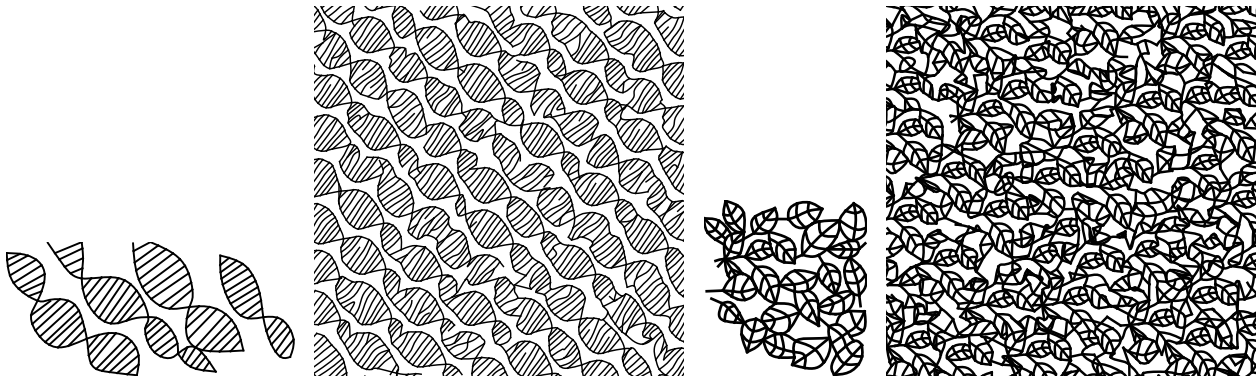


Figure 2.17: *Comparison of our algorithm with enhanced [1].* We compare our methods to an enhanced version of [1] that incorporates our ideas, including the sample connectivity and edge assignment.

Our current algorithm treats discrete elements and continuous structures separately and thus might not preserve identifiable elements within continuous structures, as exemplified in Figure 2.18. A potential future work is to find a unified representation and approach for both discrete and continuous patterns.

The pattern synthesis requires nearest neighborhood searching for output samples, which can become computationally expensive for large outputs. This neighborhood searching process can be readily parallelized [52].

We focus on curves as the first step to handle continuous vector patterns. A next step is to incorporate more vector graphics features as parts of the sample/edge attributes, such as color and thickness, as well as higher dimensional primitives including 2D regions and 3D volumes [53, 54, 55]. More controls can also be added to facilitate more diverse authoring effects such as local variations in scales and orientations [18]. In addition to optimizing pattern appearance as in this work, adding mechanical structures constraints can facilitate



(a) Input (b) Output (c) Input (d) Ouptut

Figure 2.18: *Failure case.* Our algorithm fails to preserve the identifiable DNA-segment and tree-leaf elements within the continuous structures.

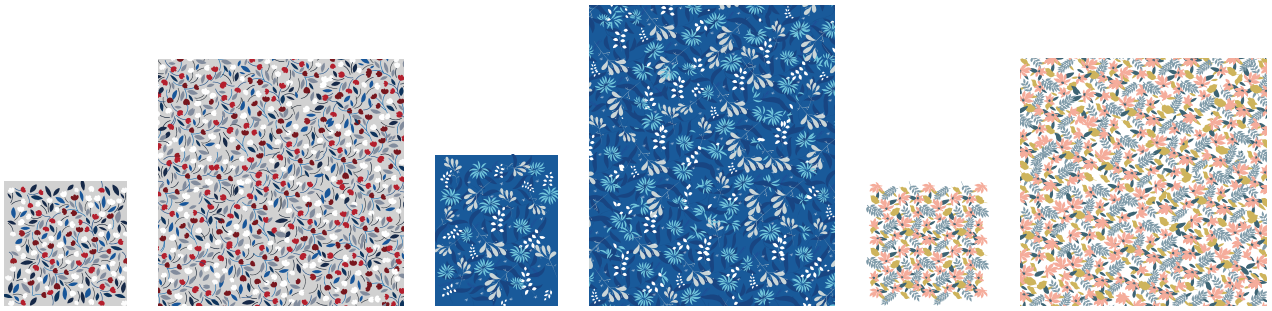
the application of curve structures for rapid manufacturing [9, 56, 10, 7, 41, 57].

Chapter 3: Clustered Vector Textures

3.1 Introduction

Vector patterns are common in design and engineering but can require high expertise and manual efforts to create. To address this issue, significant research has been devoted to automate the synthesis of vector patterns, either fully for batch generation or partially for interactive editing [55, 1, 5, 21, 7, 3, 58, 59, 27, 26, 60, 18, 2, 61, 62]. However, existing methods are more suitable for relatively simple patterns without diverse element shapes or complex, structured interactions (Figure 4.1). Another approach is to rasterize the vector pattern, apply image texture synthesis [19, 63], and vectorize the raster results. However, this process tends to lose the meaning of the original design components such as integral elements and overlapping shapes in different depth layers, often resulting in broken or merged elements.

This chapter proposes an algorithm for generating vector patterns with diverse element shapes and structured local element interactions with potential overlaps (Figure 4.1). Instead of post-vectorizing a synthesized raster output [64, 65], our method directly produces vector patterns by optimizing a sample-based representation [1, 2]. Our main idea is to explicitly optimize *clusters* of samples during the synthesis process. Similar to elements [1] and graphs [2], our clusters help to identify distinguishable pattern components



(a) input (b) output from (a) (c) input (d) output from (c) (e) input (f) output from (e)

Figure 3.1: *Example inputs and outputs of our method.* Given a small input exemplar, our algorithm can synthesize various types of structured vector patterns. All images are shown in vector format; please zoom in for details. Copyrights: (a) Snejana Sityaeva (c) Eva Kali (e) mspoint (stock.adobe.com)

that should remain integral (e.g., not merged, broken apart, or deformed) during systhesis. However, unlike these prior works that just copy element identifications [1] or compute curve identifications (reconstruct curves) from synthesized graphs as post-processing [2], our method optimizes clusters of samples as variables along with other pattern attributes such as sample positions, to facilitate robust optimization and reconstruction of the output vector patterns. The extra optimization of cluster configurations leads to better optimized patterns by expanding the feasible region - the set of all possible outcomes of pattern optimization.

Given a vector input exemplar (Figures 3.2a and 3.2b), our method samples an intermediate clustered representation where each vector element is represented as a cluster of samples (Figure 3.2c), synthesizes the output sample clusters (Figure 3.2d) via *optimization*, and finally reconstructs the output pattern (Figures 3.2e to 3.2g) from the output clusters (Figure 3.2d). Our algorithm *jointly* optimizes the spatial sample distributions and cluster configurations by adding a new clustering step into conventional search-and-assign based pattern synthesis [1, 2]. We design an objective function to measure the quality of clusters

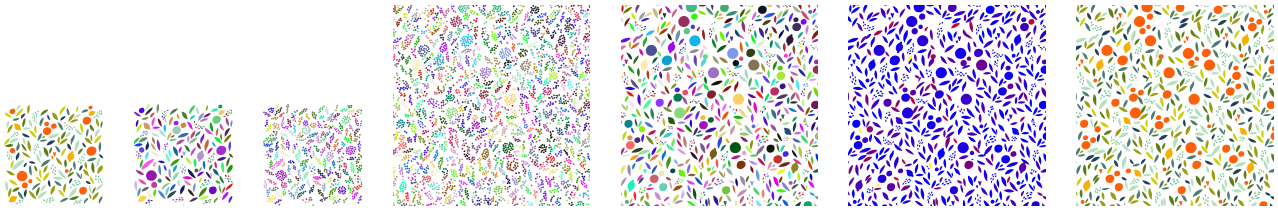


Figure 3.2: *Pattern synthesis with clustered sample representation.* Given a vector input exemplar (a) with elements visualized in different colors in (b), our method first samples a clustered representation (c), synthesizes output sample clusters via *optimization* (d), reconstructs the output elements (e) from (d) with the reconstruction error (f), and generates the final output pattern (g) by filtering out elements in (e) with high reconstruction error (f). Corresponding clusters and elements are visualized in the same colors. In (f), red and blue indicate high and low reconstruction errors; yellow indicates filtered-out elements with errors surpassing a threshold. Copyright: (a) orangemilk (stock.adobe.com)

by taking into account both input-output correspondences and cluster (element) shapes, and an optimization scheme to minimize the clustering objective.

We analyze our algorithms via robustness and ablation studies and comparisons with prior art, which show that our method is robust and can significantly outperform existing methods on synthesizing a variety of structured vector patterns.

3.2 Related Work

Our work is mainly inspired by previous works on example-based and procedural pattern synthesis and authoring. We summarize the most related works below.

Example-based pattern synthesis approaches have been applied to generate geometric patterns with optional control, such as discrete elements [30, 13, 1, 12, 14, 15, 21, 17], continuous structures [2, 3], and their combinations [3, 62, 2]. Existing methods can well handle relatively simple patterns but not complex ones with diverse element shapes and

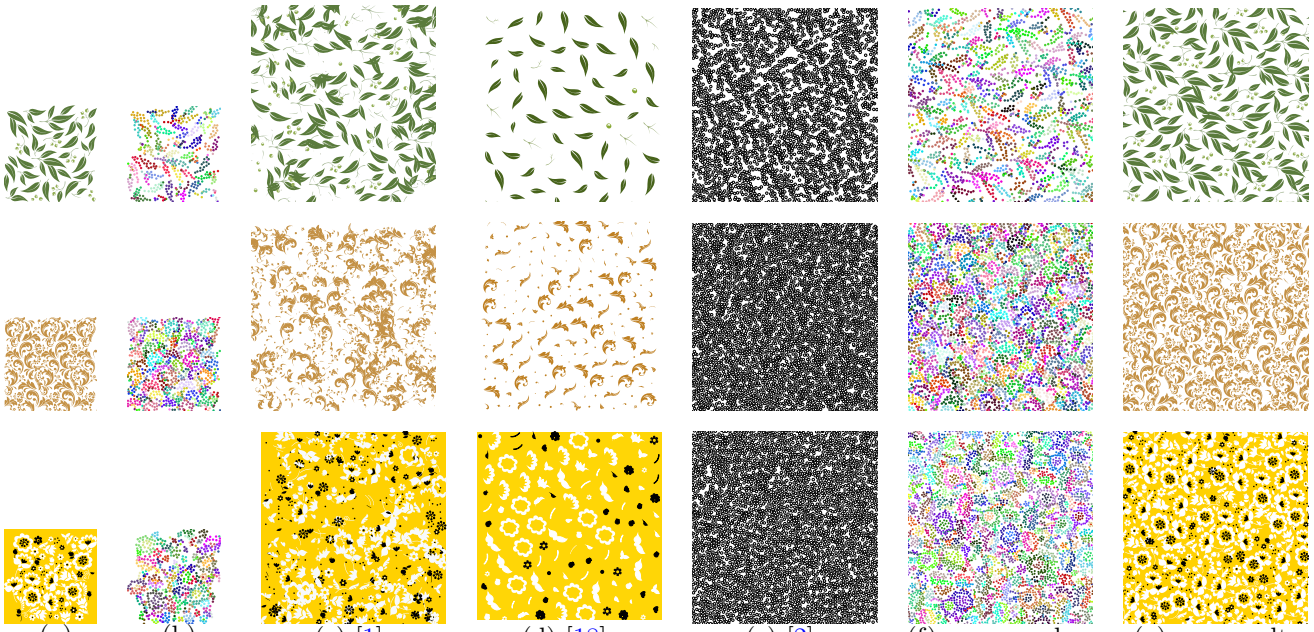


Figure 3.3: *Comparison with previous methods.* Given an input exemplar (a), our method samples a clustered sample distribution (b), synthesizes clustered output samples (f), and reconstructs the final patterns (g). Clusters are visualized in colors. We compare our method against prior vector pattern synthesis methods [1] (c), [18] (d), and [2] (e). As shown, while Ma et al. [1] can generate broken structures and Hsu et al. [18] can only place elements uniformly, our method can preserve diverse structures from the exemplars. Since [2] can only handle Bézier curve patterns via graph synthesis which could not be reconstructed as general shapes, we only show their sample synthesis results here. See Figures 3.13 and 6.9 for more comparisons. (a) top ©photo-nuke, middle ©ELENA, bottom ©galyna_p (stock.adobe.com).

structured local interactions, as compared in Figures 3.3 and 3.13. Our cluster-based representation is inspired by [1] where each vector element is represented by a set of samples with associated "element ids" (equivalent to cluster id defined in Section 3.4). The multi-sample representation is later augmented with graph-based representations [2, 18]. However, the "element ids" in [1, 18] are pre-defined and unoptimized; as each element is associated with a fixed set of samples, these prior methods may not be able to handle complex, structured patterns that require changing sample relationships during synthesis (Figures 3.3c and 3.3d). [2] synthesize graphs and reconstruct curves from graphs by post-processing.

However, graphs could not be applied for representing complex shapes other than curves, such as identifiable elements. Instead, our method *explicitly optimizes sample clusters* (e.g. their sizes and shapes (Equation (3.15))), which can represent diverse element shapes, and cluster configurations (e.g. number of clusters and spatial distribution of clusters) with our proposed neighborhood criteria (Equation (3.2)), and clustering objective (Section 3.6.5.1) and algorithm (Section 3.6.5.2).

Procedural approaches can produce a variety of distributions and patterns including point distributions [66, 67], packed elements [59, 68, 69, 70, 18] or structured patterns [71, 25, 26, 27, 72, 61, 73, 74, 75]. One key advantage of procedural methods is user control [76], even though the range of outputs can be limited by the underlying grammars or procedures.

Components of these automatic methods have been integrated with interactive interfaces to facilitate user control in the form of custom brushes or widgets, via exemplars [6, 20, 5, 77, 78] or procedures [60]. However, existing methods work best for patterns that can be easily supplied by users, and more complex ones still require significant manual authoring. Our algorithm aims to narrow this gap with the ability to generate a variety of structured vector patterns from exemplars, as compared in Figure 3.3.

Our clustering idea also relates to the use of guidance/control maps or channels in prior image texture synthesis methods [79, 80, 51, 81, 82]. However, instead of providing coarse user control, the clusters in our method aim to preserve detailed vector structures. In addition, clustering constitutes a dedicated step in our optimization algorithm.

Unlike raster texture synthesis [19, 63], the output of our method is in vector format and thus can facilitate further editing (Figure 3.12).

3.3 Overview

Our method extends prior optimization-based pattern synthesis methods [1, 2] to represent vector patterns via clusters of samples (Section 3.4 and Figure 3.2c). Analogous to elements in [1] and connected graph edges in [2], our clusters correspond to identifiable pattern components that should remain integral during synthesis, such as avoiding merging, splitting, or deformation. However, our method treats clusters as variables in our optimization, along with other attributes such as sample positions as in [1, 2]. We aims to optimize the similarity between two patterns by maximizing the similarity between local neighborhoods. In Sections 3.4 and 3.5, we explicitly incorporate the cluster information into pattern neighborhoods and their similarity (Equation (3.2)) by associating each sample with a shape context feature [83] computed using its cluster. In Section 3.6, we define the pattern optimization objective which is the sum of pairwise nearest neighorhood similarity (Section 3.6.1) and the optimization strategy including initialization (Section 3.6.2) and iterative search (Section 3.6.3), assignment (Section 3.6.4) and clustering (Section 3.6.5) steps. The search step finds nearest neighborhoods and the assignment and clustering steps update samples. The clustering step, which includes a clustering objective (Section 3.6.5.1) and a greedy optimization-based clustering algorithm (Section 3.6.5.2), is the key new component of our method to better handle complex patterns.

3.4 Representation

3.4.1 Vector pattern

Our input vector pattern is in SVG format [84]. Vector elements are deduced from the SVG file, such as individual paths and ¡g¿ tags that are used to group SVG elements [84]. Optionally, users can manually group or split some input paths to reflect their design intentions. Intuitively, elements are repetitive spatial geometric shapes.

3.4.2 Clustered pattern representation

For pattern synthesis, we use an intermediate clustered representation of vector patterns. We represent each element using a set (cluster) $\{s\}$ of samples s (Figures 3.2b and 3.2c). Specifically, we sample interior regions uniformly using a Poisson disk distribution [85] with disk diameter δ. A smaller δ means a denser sample distribution that can capture more shape details. δ can be chosen proportional to the size of patterns or element shapes.

Similarly to hierarchical image texture synthesis [49, 86], we further synthesize patterns with hierarchical sampling [2, 49]. Increasing hierarchy levels use denser sample distributions (with decreasing δ).

Each sample records the following information:

Spatial parameters $\mathbf{S}$, which include sample position $\mathbf{p} \in \mathbb{R}^2$, cluster id $\mathbf{i} \in \mathbb{Z}^+$, which indicates the cluster C it belongs to, confidence of sample existence $\xi \in [0\ 1]$, which enables the optimization of the number of samples [2], z-index $z \in \mathbb{Z}^+$, which defines

the relative layer order of associated elements appearing in the SVG file [84], and a feature vector $\mathbf{f} \in \mathbb{R}^{N_\mathbf{f}}$ ($N_\mathbf{f}$ is the feature length), which encodes its relative position within its cluster $C \ni s$. We use shape contexts [83], which can be applied to point-based geometry, to compute $\mathbf{f}$ using the cluster C of samples. The output clusters are optimized automatically via our method, as detailed in Section 3.6.5.

Appearance attributes $\mathbf{A}$, such as color, opacity, and gradients.

Together, a sample is represented as

$$\mathbf{U}(s) = (\mathbf{S}(s), \mathbf{A}(s)) \tag{3.1}$$

3.5 Neighborhood and Similarity

Our synthesis method follows the sample-based neigborhood optimization framework [1, 2]. Therefore, one of the core parts of our method is to define neighborhoods and their pairwise similarity criterion.

We define $\mathbf{n}(s)$, the neighborhood of s, as a set of samples around s's spatial vicinity within a certain radius r, including s itself. The distance between an input $\mathbf{n}(s_i)$ (centered at s_i) and an output neighborhood $\mathbf{n}(s_o)$ (centered at s_o) is defined by matching samples $s'_i = m_s(s'_o)$ within the input and output neigborhoods $s'_i \in \mathbf{n}(s_i)$ and $s'_o \in \mathbf{n}(s_o)$, where m_s is the one-to-one sample matching function (i.e. $s'_i = m_s(s'_o)$ means that $s'_i \in \mathbf{n}(s_i)$ is the matched input sample for output sample $s'_o \in \mathbf{n}(s_o)$) defined between two neighborhoods

$m_s : \mathbf{n}(s_o) \rightarrow \mathbf{n}(s_i)$. The distance is written as

$$d_{\mathbf{n}}\left(\mathbf{n}(s_o), \mathbf{n}(s_i)\right) = \min_{\substack{m_s \\ \text{s. t. } s_i = m_s(s_o)}} \sum_{\substack{s'_o \in \mathbf{n}(s_o) \\ s'_i = m_s(s'_o) \in \mathbf{n}(s_i)}} d_s(s'_o, s'_i), \tag{3.2}$$

where $d_s(s'_o, s'_i)$ is the sample similarity between $s'_o \in \mathbf{n}(s_o)$ and $s'_i \in \mathbf{n}(s_i)$. The distance in Equation (3.2) is defined as the minimum over all possible sample matchings m_s. The optimal matching m_s is computed using the Hungarian algorithm [42], subject to that centers s_i and s_o are always matched $s_i = m_s(s_o)$.

The sample similarity $d_s(s'_o, s'_i)$ between $s'_o \in \mathbf{n}(s_o)$ and $s'_i \in \mathbf{n}(s_i)$ is defined as the weighted sum of differences between sample properties $\mathbf{U}(s'_i)$ and $\mathbf{U}(s'_o)$ (as in Equation (3.1)):

$$d_s(s'_o, s'_i) = \|\hat{\mathbf{p}}(s'_o, s_o) - \hat{\mathbf{p}}(s'_i, s_i)\| + \frac{w_{\mathbf{f}}}{2} \sum_{q=1}^{N_{\mathbf{f}}} \frac{\left(\mathbf{f}^q(s'_o) - \mathbf{f}^q(s'_i)\right)^2}{\mathbf{f}^q(s'_o) + \mathbf{f}^q(s'_i)} \tag{3.3}$$

The first term measures the difference on sample positions relative to centers s_o, s_i; $\hat{\mathbf{p}}(s', s) = \mathbf{p}(s') - \mathbf{p}(s)$. The second term, where $\mathbf{f}^q$ is the qth entry in $\mathbf{f}$, measures the difference of shape context features, defined using χ^2 test statistic as in [83]. $w_{\mathbf{f}} = 10$ is the weight for the second term. Since the shape context feature is computed using clusters, *the sample similarity effectively compares both individual samples and their associated clusters.* We do not consider appearance attributes $\mathbf{A}$ in our experiments, as different shapes often have unique colors and thus considering the differences between spatial $\mathbf{S}$ information is sufficient to evaluate the neighborhood similarity. One can define appearance similarity as the weighted sum of differences between the appearance vector $\mathbf{A}(s'_o)$ and $\mathbf{A}(s'_i)$ when

necessary.

3.6 Pattern Synthesis

3.6.1 Optimization objective

We formulate the output pattern synthesis as an optimization problem [49, 1, 2] between input $\mathcal{I} = \{s_i\}$ and output $\mathcal{O} = \{s_o\}$ sample sets:

$$\mathcal{O}^\star = \arg\min_{\mathcal{O},\mu_s} \sum_{\substack{s_o \in \mathcal{O} \\ s_i = \mu_s(s_o)}} d_{\mathbf{n}}\left(\mathbf{n}(s_o), \mathbf{n}(s_i)\right) \tag{3.4}$$

The sum loops over all the local output neighborhoods over $\mathcal{O}$. $s_i = \mu_s(s_o)$ indicates that $\mathbf{n}(s_i)$ is the most similar input neighborhood to $\mathbf{n}(s_o)$. In other words, μ_s is the sample matching function which defines the nearest-neighbor field [47] between input $\mathcal{I}$ and output $\mathcal{O}$. Note μ_s is different from m_s in Equation (3.2), which is the sample matching function within two neighborhoods. Equation (3.4) is minimized by alternating the optimization over $\mathcal{O}$ and μ_s through iterative search-assignment-clustering steps (Sections 3.6.3 to 3.6.5). Essentially, the search step minimizes Equation (3.4) over μ_s by finding the most similar input neighborhood for each output neighborhood. The assignment and clustering steps minimize Equation (3.4) over $\mathcal{O}$ by modifying output sample parameters $\mathbf{U}(s_o)$ $(s_o \in \mathcal{O})$ using μ_s computed during the search step in the current iteration.

3.6.2 Initialization

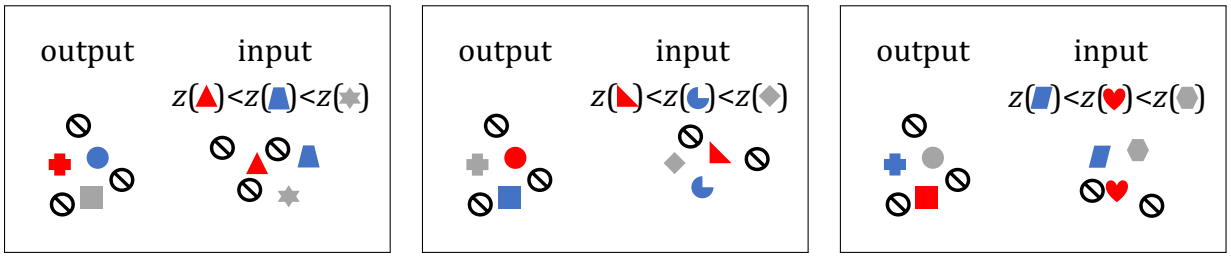
Similar to [1, 2], we randomly copy input patches into the output domain. A patch is a square block within a pattern domain, which can consist of several clusters, and cluster samples outside of a patch boundary are ignored. The patch size is chosen as twice the neighborhood radius $2r$. Our algorithm can also generate similar results with random sample initialization, but patch-based initialization can reach lower optimized objective values using the same number of iterations [1]. Please see Figure 6.1. All attributes $\mathbf{U}(s_o)$ of an output sample s_o are copied from the corresponding input sample s_i attributes $\mathbf{U}(s_i)$ except for 1) the position $\mathbf{p}(s_o)$ which is shifted from $\mathbf{p}(s_i)$ during patch copying, 2) the cluster id $\mathbf{i}(s_o)$ for which i) samples within the same copied patch will receive unique values different from those in other copied patches and ii) samples in a copied patch will share the same $\mathbf{i}(s_o)$ if their source input ids $\mathbf{i}(s_i)$ are the same, and 3) the sample feature $\mathbf{f}$ which is recomputed for broken clusters on input patch boundaries.

3.6.3 Search step

In the search step, for each output sample, we find the input sample with the most similar neighorhood evaluated by Equation (3.2), which is accelerated with the patch match algorithm [47, 2].

3.6.4 Assignment step

In the assignment step, we optimize the output sample parameters by overlapping matched input neighborhoods over the output samples [1, 2]. For each output sample, there



(a) matched neighborhood pair 1 (b) matched neighborhood pair 2 (c) matched neighborhood pair 3

s_o \ s'_o	✚	●	■
✚	0	2/3	2/3
●	1/3	0	2/3
■	1/3	1/3	0

(d) $f_s(s_o, s'_o)$

Figure 3.4: *A toy example of computing the probabilistic pairwise ordering function* $f_s(s_o, s'_o)$. (a), (b) and (c) show three pairs of input and output neighborhoods ($|N_n(s_o, s'_o)| = 3$) where the output neighorhoods are centered at different samples (in red) over the same set of output samples. The three input neighborhoods exhibit different layering relationships among samples. Different samples are visualized in different shapes. For example, output samples in all three pairs of neighborhoods are shown in the same shapes because they are the same set of samples. ⦸ indicates samples that are not used as examples in (d). Matched input and output samples are visualized in the same colors. (d) shows the computed $f_s(s_o, s'_o)$ among three output samples (in solid shape). . For example, $f_s(\bullet, \blacksquare)$ is $\frac{2}{3}$ because in the matched neighborhoods (a), (b) and (c) the matched input samples $m_s(\bullet)$ of $\bullet$ is below ($<$) $m_s(\blacksquare)$ in (a) and (b) but not in (c), so that $|N_z(s_o, s'_o)| = 2$. Note that we only consider the limited three pairs of neighborhoods (where the output contains s_o and s'_o) for illustration purposes.

is a set of input samples that are matched with it. For sample position $\mathbf{p}$, least squares is used to minimize the sum of differences between distances and expected distances between pairs of output samples. We also adaptively optimize the number of samples as in [2]. Please refer to Section 6.2 for more details. The assignment of cluster id $\mathbf{i}$ is a clustering problem, which is significantly different from existing assignment algorithms [2, 1]. We will discuss the id assignment as the clustering step in Section 3.6.5. The sample feature $\mathbf{f}$ is assigned using updated cluster configuration after the clustering step.

Depth z assignment Vector patterns may consist of overlapping elements (e.g. Figure 3.5d), where one with a larger z-index z can cover another with a small z-index. To preserve layer relationship in pattern synthesis, we optimize the layer depth z of output samples as follows.

Since the exact value of z is not important but the ordering among z of different samples, we first compute a probabilistic pairwise ordering function $f_s(s_o, s'_o) \in [0, 1]$. The value of $f_s(s_o, s'_o)$ indicates the probability of s_o below s'_o, which is computed from matched input and output neighborhoods. Intuitively, in a pair of a matched input $\mathbf{n}_i$ and output neighborhood $\mathbf{n}_o$, $z(m_s(s_o)) < z(m_s(s'_o))$ suggests $z(s_o) < z(s'_o)$. Figure 3.4 illustrates the computation of $f_s(s_o, s'_o)$ using a toy example.

Based on the above intuition, we optimize depth as follows:

$$f_s(s_o, s'_o) = \begin{cases} \frac{|N_z(s_o, s'_o)|}{|N_n(s_o, s'_o)|}, & \|\mathbf{p}(s_o) - \mathbf{p}(s'_o)\| < 2\delta \\ \text{undefined}, & \|\mathbf{p}(s_o) - \mathbf{p}(s'_o)\| \geq 2\delta \end{cases} \tag{3.5}$$

$$N_n(s_o, s'_o) = \{(\mathbf{n}_i, \mathbf{n}_o) | (\mathbf{n}_i, \mathbf{n}_o) \in \mathcal{N}, s_o, s'_o \in \mathbf{n}_o\} \tag{3.6}$$

$$\begin{aligned} N_z(s_o, s'_o) = \{(\mathbf{n}_i, \mathbf{n}_o) | (\mathbf{n}_i, \mathbf{n}_o) \in N_n(s_o, s'_o), m_s : \mathbf{n}_o \to \mathbf{n}_i, \\ z\left(m_s(s_o)\right) < z\left(m_s(s'_o)\right)\} \end{aligned} \tag{3.7}$$

where $|\cdot|$ is the size of a set $\cdot$. $\mathcal{N} = \{(\mathbf{n}(s_i), \mathbf{n}(s_o)) \,|s_o \in \mathcal{O}, s_i = \mu_s(s_o)\}$ is the set of matched input and output neighborhoods computed in the search step (Section 3.6.3). $N_n(s_o, s'_o)$ is the set that includes matched neighborhood pairs where the output neighborhoods contain both s_o and s'_o. $N_z(s_o, s'_o)$ is a subset of $N_n(s_o, s'_o)$where the matched input

samples $m_s(s_o)$ is below $m_s(s'_o)$ ($z\left(m_s(s_o)\right) < z\left(m_s(s'_o)\right)$). Since we are only interested in pairs of output samples s_o and s'_o that are sufficiently close for potential overlap, $f_s(s_o, s'_o)$ is only defined when $\|\mathbf{p}(s_o) - \mathbf{p}(s'_o)\| < 2\delta$.

Since we order samples within the same clusters together instead of individually, we compute a pairwise ordering function $f_C(C_i, C_j)$ between sample clusters C_i, C_j:

$$f_C(C_i, C_j) = \frac{1}{Z} \sum_{s_o \in C_i} \sum_{\substack{s'_o \in C_j \\ \|\mathbf{p}(s_o) - \mathbf{p}(s'_o)\| < 2\delta}} f_s(s_o, s'_o) \tag{3.8}$$

where Z is the normalization factor that counts the number of entries within the double summation. Given the pairwise ordering function (Equation (3.8)), the depth order among clusters (or reconstructed elements) is computed using the "Order-By-Preferences" algorithm proposed in [87]. Figure 3.5 shows the effects with and without the z-index assignment.

3.6.5 Clustering step

The clustering step updates sample cluster id $\mathbf{i}$. We address this clustering problem based on two observations:

Sample correlations In a pair of matched input and output neighborhoods $(\mathbf{n}_i, \mathbf{n}_o)$, if there are input samples within the same input cluster, their matched output samples should also be within the same output cluster, and vice versa.

Cluster shape similarity In pattern synthesis, the output element (and thus cluster) shape should be close to an input element (cluster) shape.

These two observations are incorporated into the optimization-based clustering step with

objective E (Equation (3.9)), as two different energy terms: link energy E_l and shape energy E_s. We develop a greedy optimization algorithm to optimize Equation (3.9) via an iterative mechanism based on local operators including sample switching (from one cluster to another), and cluster merge and split (Section 3.6.5.2). While the sample switching gradually adjusts clusters, cluster merge and split update the clusters more drastically which not only optimizes the cluster shapes but also total number of clusters. Details are as follows.

3.6.5.1 Objective

Eventually, we hope to minimize the pattern synthesis objective (Equation (3.4)). Instead of minimizing Equation (3.4) with respect to cluster id $\mathbf{i}$ through brute-force search over all possible cluster configurations, we propose a separate objective defined over output cluster set $\{C_o^k\}$ (k is the index of clusters), for which we develop an efficient greedy optimization algorithm. The objective is as follows.

$$E(\{C_o^k\}) = w_l E_l(\{C_o^k\}) + w_s E_s(\{C_o^k\}) \tag{3.9}$$

where E_l and E_s are the link and shape energies defined over a cluster configuration $\{C_o^k\}$, respectively. w_l, w_s are the weights for E_l and E_s. $w_l = 1$ and $w_s = 4$.

Link Energy The link energy measures how compatible an output cluster configuration is with respect to input clusters based on *sample correlations*. Intuitively, in the set of

matched pairs of input and output neighborhoods $\mathcal{N}$, if there are more pairs of output samples which are 1) in the same output cluster and matched input samples in the same input cluster or 2) in different output clusters and matched input samples in different input clusters, $E_l(\{C_o^k\})$ should be lower and indicates higher compatibility between input and output clusters. Before defining the link energy E_l, we compute the link confidence $l \in [0, 1]$ of two output samples belonging to the same output cluster from $\mathcal{N}$, which will be used to define E_l, as follows

$$l(s_o, s'_o) = \frac{|N_l(s_o, s'_o)|}{|N_n(s_o, s'_o)|} \tag{3.10}$$

$$\begin{aligned} N_l(s_o, s'_o) = \{(\mathbf{n}_i, \mathbf{n}_o) | (\mathbf{n}_i, \mathbf{n}_o) \in N_n(s_o, s'_o), m_s : \mathbf{n}_o \to \mathbf{n}_i, \\ \mathbf{i}(m_s(s_o)) = \mathbf{i}(m_s(s'_o))\}. \end{aligned} \tag{3.11}$$

Intuitively, we compute the confidence l of two output samples s_o, s'_o belonging to the same cluster based on the voting of all input neighborhoods overlapping them. Figure 3.6 illustrate the computation of link confidence l. $N_n(s_o, s'_o)$ is the set of matched input and output neighborhoods where the output neighborhood contain both of s_o and s'_o, as defined in Equation (3.6). $N_l(s_o, s'_o)$ is the subset of $N_n(s_o, s'_o)$ where the matched input samples $m_s(s_o)$ and $m_s(s'_o)$ have the same cluster id $\mathbf{i}(m_s(s_o)) = \mathbf{i}(m_s(s'_o))$. Note that $l(s_o, s'_o)$ is undefined when $|N_n(s_o, s'_o)| = 0$. The larger $l(s_o, s'_o)$ is, the more likely s_o, s'_o are within the same cluster.

The link energy $E_l(\{C_o^k\})$ is defined as

$$E_l(\{C_o^k\}) = \sum_{s_o \in \mathcal{O}} \sum_{\substack{s'_o \in \mathcal{O} \\ \|\mathbf{p}(s_o) - \mathbf{p}(s'_o)\| < \lambda\delta}} E'_l(s_o, s'_o) \tag{3.12}$$

where

$$E'_l(s_o, s'_o) = \begin{cases} 1 - 2l(s_o, s'_o) & \text{if } \mathbf{i}(s_o) = \mathbf{i}(s'_o) \\ 2l(s_o, s'_o) - 1 & \text{if } \mathbf{i}(s_o) \neq \mathbf{i}(s'_o) \end{cases} \tag{3.13}$$

$E'_l(s_o, s'_o) \in [-1, 1]$ is defined over pairs of samples that are spatially close ($\|\mathbf{p}(s_o) - \mathbf{p}(s'_o)\| < \lambda\delta$, λ is a hyperparameter) by two cases based on whether s_o and s'_o are in the same ($\mathbf{i}(s_o) = \mathbf{i}(s'_o)$) or different ($\mathbf{i}(s_o) \neq \mathbf{i}(s'_o)$) clusters. r is the neighborhood radius appearing in Section 3.5. By minimizing Equation (3.12) with respect to $\{C_o^k\}$, which is the sum of Equation (3.13) over pairs of output samples, we encourage s_o, s'_o to be in the same cluster if the link confidence $l(s_o, s'_o)$ is high and be in different clusters if $l(s_o, s'_o)$ is low.

Shape Energy $E_s(\{C_o^k\})$ is the sum of cluster shape energies E'_s, which are defined on individual clusters, over all clusters

$$E_s(\{C_o^k\}) = \sum_k E'_s(C_o^k) \tag{3.14}$$

where $E'_s(C_o)$ measures the differences between an output cluster shape C_o and the corresponding input clusters $\{C_i\}$. A large $E'_s(C_o)$ indicates that the output cluster C_o is dissimilar to the shape of the corresponding input clusters $\{C_i\}$. As output elements

should come from the input, an output cluster is of good shape (low $E'_s(C_o)$) if it is similar to the shape of one of the input clusters. The shape energy $E'_s(C_o)$ is defined as

$$E'_s(C_o) = \min_{C_i \in \mathfrak{C}_i} d_C(C_i, C_o), \tag{3.15}$$

where $\mathfrak{C}_i = \{C_i | s_i \in C_i, s_o \in C_o, s_i = \mu_s(s_o)\}$ is the set of candidate input clusters that have their samples s_i matched with the output samples s_o of the cluster C_o in μ_s computed in the search step (Section 3.6.3). $d_C(C_i, C_o)$ is the shape distance between an input cluster C_i and an output cluster C_o, which is defined as

$$d_C(C_i, C_o) = \min_m \frac{1}{\epsilon} \left(\sum_{\substack{s_o \in C_o \\ s_i = m(s_o) \in C_i}} \| \mathbf{p}(s_o) - \mathcal{T}_r(\mathbf{p}(s_i), C_i, C_o) \| + \epsilon \cdot \text{abs}(|C_o| - |C_i|) \right) \tag{3.16}$$

The first term computes the sum of distances between the translated input cluster samples $s_i \in C_i$ by $\mathcal{T}_r$ and output cluster samples $s_o \in C_o$. $\epsilon = 2\delta$ is the extra cost induced by the difference between the numbers of input $|C_i|$ and output cluster samples $|C_o|$. m is the sample matching function in computing shape distance (Equation (3.16)) between two clusters of samples. The whole equation is normalized using $\epsilon = 2\delta$ to offset the scaling effect of δ (and thus the size of input exemplar/shapes) on the shape distance. $\mathcal{T}_r$ is used to translate the input cluster samples $s_i \in C_i$ to the position of the output cluster samples $s_o \in C_o$ so that we can evaluate the shape similarity based on distances between samples,

which is defined as

$$\mathcal{T}_r(\mathbf{p}(s_i), C_i, C_o) = \mathbf{p}(s_i) + \frac{1}{|\mathcal{S}(C_i, C_o)|} \sum_{(s_i', s_o') \in \mathcal{S}(C_i, C_o)} (\mathbf{p}(s_o') - \mathbf{p}(s_i')) \,, \tag{3.17}$$

where $\mathcal{S}(C_i, C_o) = \{(s_i, s_o) | s_i \in C_i, s_o = \mu_s(s_i) \in C_o\}$ is a set of matched sample pairs where the input and output samples are associated with the input C_i and output clusters C_o respectively. Intuitively, the translation is the average of all translations between sample pairs within $\mathcal{S}$. Figure 3.7 illustrates the computation of Equations (3.15) to (3.17).

3.6.5.2 Optimization

To optimize Equation (3.9) with respect to $\{C_o^k\}$, inspired by previous image segmentation algorithms [88, 89] we develop a greedy algorithm that efficiently explores a large solution space via an iterative mechanism based on local operators, as shown in Figure 3.9, which 1) switch samples from one cluster to another nearby cluster, 2) split one cluster into two clusters, and 3) merge two clusters into one. The energy variations induced by these operators are inserted and sorted into priority queues, with the front operator reducing the energy (Equation (3.9)) most. Since we will use different weights in computing sample switching and cluster operators, two queues P_s, P_C are used: P_s is for all sample switching operators and P_C is for all cluster splitting and merging operators. We first process the sample switching queue P_s and then cluster operator queue P_C. Given a priority queue P of operators, we update the cluster configurations $\{C_o^k\}$ with the front operator, update queues by removing the current operator and other affected operators, and insert new operators due to the update of $\{C_o^k\}$. This process iterates until a stopping criterion is met.

Figure 3.8 shows an example of the objective function (Equation (3.9)) during optimization using the clustering operators.

Sample switching Only boundary samples s_o of an output cluster can switch to its adjacent clusters $\mathcal{A}^C(s_o)$ (Figures 3.9a and 3.9b). We extract the boundary samples of a cluster using the concave hull. A cluster C_o is adjacent to a sample s_o ($C_o \in \mathcal{A}^C(s_o)$) if there is any samples in the cluster C_o that has distance to s_o less than 2δ. All the possible sample switching operators are collected and added into P_s.

Since there are a lot more candidate sample switches than cluster splits and merges (Table 6.1 in Chapter 7), for which the shape energy E_s is expensive to compute, we only use the link energy E_l for computing the induced energy variation during sample switching.

Cluster merging Similarly, for a pair of clusters C_o^i, C_o^j, if there is any pair of samples from the two clusters that are adjacent with distance less than 2δ, then $C_o^j \in \mathcal{A}^C(C_o^i)$ ($C_o^i \in \mathcal{A}^C(C_o^j)$). For a pair of adjacent clusters, we merge them into one cluster (Figures 3.9c and 3.9d) and collect such operators in the priority queue P_C.

Cluster splitting For each cluster, we split it into two clusters (Figures 3.9e and 3.9f) using normalized cuts [91], where the graph weights are the link confidence l. These operators are collected and added into P_C.

Update priority queue After the cluster configuration $\{C_o^k\}$ is modified with an operator, we first remove the current operator and the affected operators related to the modified clusters from the priority queue. Then we add new operators related to new clusters and

insert them into the corresponding sorted priority queue. Since all operations are local, the update of P is relatively efficient.

Stopping criterion Once the front operator in P does not reduce the energy (Equation (3.9)) or the number of iterations reaches a pre-specified threshold $i_{\max}$ (which equals the number of output samples for P_s or initial number of clusters for P_C), the iteration is stopped.

3.6.6 Reconstruction

For each cluster, we reconstruct the final shape by translating the associated element of the candidate input cluster C_i that minimizes the shape energy in Equation (3.15). In addition, we filter out elements with high shape energy (error) ($E_s(C_o) > |C_o|$).

3.6.7 Hierarchical optimization

At each hierarchy level, the optimization iterates the search, assignment and clustering steps. The next level is initialized with samples from the optimized pattern from the previous level. With increasing hierarchy levels we use denser sample distributions with decreasing Poisson disk diameter δ. Hierarchical optimization allows the optimization to use less samples and larger neighborhood size, which can capture larger spatial structure with less computation. Figure 3.10 visualizes a hierarchical optimization process. See Figures 6.2 to 6.5 in Chapter 7 as well as the supplementary video for more visualizations of pattern optimization.

3.7 Results

Our method can automatically synthesize satisfactory results for a variety of vector patterns, as exemplified in Figures 3.3, 3.11, 3.14 and 4.1. Since our synthesis outputs are in vector format, users can edit them in any vector graphics editor for further quality improvement and customization, as demonstrated in Figure 3.12.

3.7.1 Ablation studies

We analyze the components of our method via ablation studies. Without the clustering step (Section 3.6.5), the generated samples resemble these by [2] in Figure 3.3, which cannot robustly reconstruct shapes from unclustered samples. Without using cluster information in the search step (i.e., only the first term in Equation (3.3) is used), the result patterns tend to be more random as shown in the second column in Figure 3.14. Without the link/shape energy E_l/E_s in the clustering objective (Equation (3.9)), the results tend to have empty and broken regions as shown in the third/forth column in Figure 3.14. Our full results are shown in the last column in Figure 3.14. We include ablation studies on the clustering operators in Figure 6.8.

3.7.2 Comparison to previous methods

We compare our method against prior methods in vector element synthesis [1, 18] (Figure 3.3) and vector curve synthesis [2]. [2] can generate Bézier curve patterns by reconstructing curves from graphs, which cannot be applied to the varied structured input patterns that have elements other than open Bézier curves. In order to compare with [2] in

reconstructed patterns instead of just samples as in Figure 3.3, we add a basic clustering step [90] to each iteration in the optimization process of [2]. We use spectral clustering [90] as the basic clustering algorithm; each input element is represented as a fully connected graph, and the output edge confidences in [2] are used as the weights for the weighted similarity graphs in [90]; the number of output clusters is estimated as the product of the ratio between numbers of output and input samples and the number of input clusters (elements) $\frac{|\mathcal{O}|}{|\mathcal{I}|} \cdot |\{C_i\}|$, while our method can automatically determine the number of output clusters. In addition, the cluster informaton is used in the neighborhood similarity, as in Equations (3.2) and (3.3). The final shapes are reconstructed as described in Section 3.6.6. As shown in Figures 3.3 and 3.13, our method can produce better results for a wider range of patterns than these existing methods. More comparisons are shown in Figure 6.9.

3.7.3 Parameters

Our method is robust to chosen parameters and initial conditions. Please see Section 6.1 for a robustness analysis of our algorithm. We use two or three levels of hierarchies. In each hierarchy, we use 7 iterations of search-assign-clustering steps. The neighborhood radius r is $\frac{1}{6}$, $\frac{1}{8}$, $\frac{1}{12}$ of the exemplar size (mean of width and height). The sampling distances δ are $\frac{1}{20}$, $\frac{1}{30}$, $\frac{1}{40}$ of the exemplar size. $\lambda = 4$ (in Equation (3.12) and $\lambda\delta \approx r$). All results shown in this chapter are produced under a single parameter setting, except for Figure 3.1c, which uses $\lambda = 2$, Figure 3.11 (row 3, col 1), which uses one hierarchy (r, δ are $\frac{1}{12}$ and $\frac{1}{40}$ of the exemplar size), and Figure 3.2 where samples are denser for illustration purpose.

3.8 Conclusions, Limitations and Future Work

This chapter proposes a clustering method to synthesize vector patterns with complex structures that have yet to be well handled by existing techniques, as demonstrated in our results and analysis.

Vector patterns with regular or non-local structures or insufficient repetitions might not be well handled by our method (Figure 3.15) due to the inherent limitation of the neighborhood-based measures and local neighborhood optimization. Machine learning along with hierarchical synthesis or differentiability could be applied to address these issues [92, 93, 94, 95]. The proposed pattern optimization framework could incorporate further improvement developed in prior texture optimization literatures, such as bidirectional similarity [96, 97], spatial uniformality, or advanced initialization [51], for more robust pattern synthesis. For patterns with very elongated elements (e.g. the first pattern in Figure 3.12), our method may produce undesirable overlaps. A possible solution is to represent each elongated element using samples with central skeletal path, and synthesize them together like [2] followed by reconstruction guided by the synthesized skeletons [98]. We leave this as a future work.

The core of our proposed framework is explicit clustering, which is general and could be improved with more advanced clustering methods [99].

Our current method can be extended for pattern interpolation, animated patterns [68], and 3D volumes [53, 55], as well as integrated with more user control [5, 6], such as synthesis over user-specified (irregular) domain and optional toroidal boundary condition (Figure 3.15e). Our unoptimized implementation is slow and only suitable for off-line

synthesis. Please see the supplementary materiel for the running time of each algorithm component. Theoretically, the major computational bottleneck is in the search step (that computes nearest neighbor fields), which could be readily parallelized on a GPU [52]. For complexity analysis of the search step, please see [2]. We plan to further accelerate it to achieve interactive speed in the future, for example, by GPU parallelization, or by improving the sampling method [100] to reduce the number of required samples.

With the development of vector pattern synthesis algorithm which can generate more diverse patterns, it is desirable to use quantitative metrics to evaluate the synthesis quality in the future.

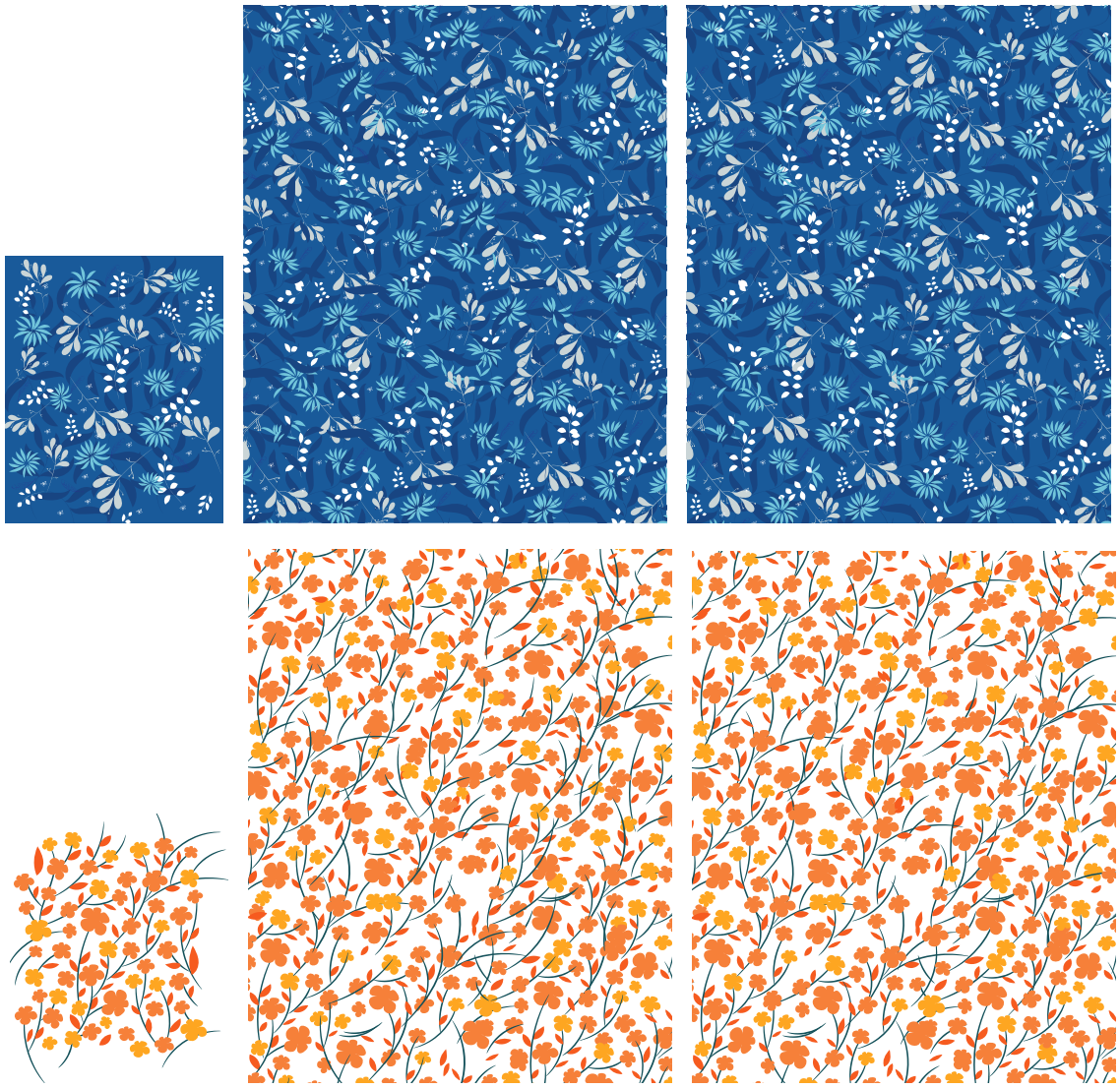
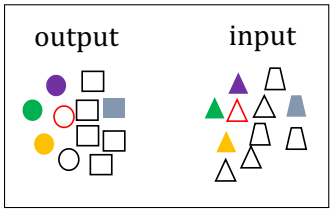
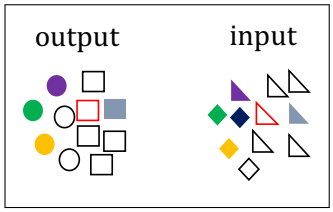


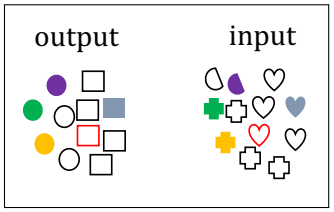
Figure 3.5: *Z-index (depth) assignment.* In the input exemplars, the sky blue petals are above other elements (top) and the branches are below the leaves (bottom). Without depth assignment, this layer relationship is not preserved. With depth assignment, the relationship is mostly preserved. Copyright: exemplars from Eva Kali (top), Snejana Sityaeva (bottom) (stock.adobe.com).



(a) matched neighborhood pair 1



(b) matched neighborhood pair 2



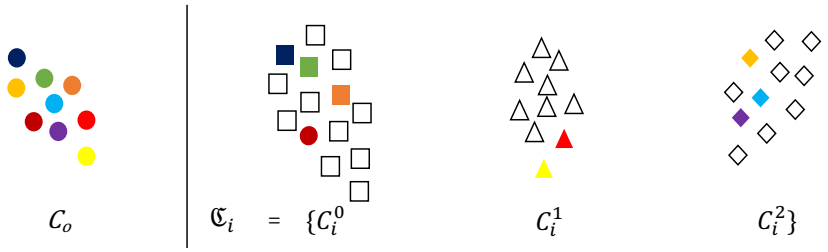
(c) matched neighborhood pair 3

$l(s_o, s'_o)$

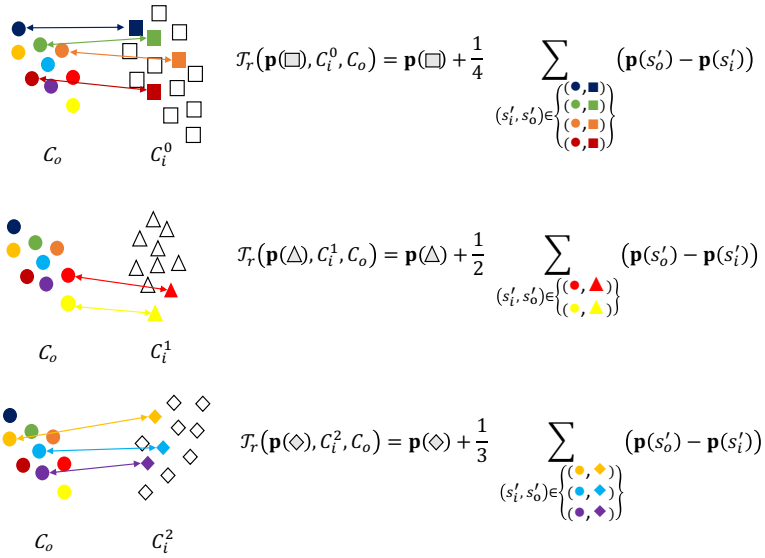
s_o \ s'_o	●	●	●	■
●	-	1/3	1	0
●	1/3	-	1/3	1/3
●	1	1/3	-	0
■	0	1/3	0	-

(d) $l(s_o, s'_o)$

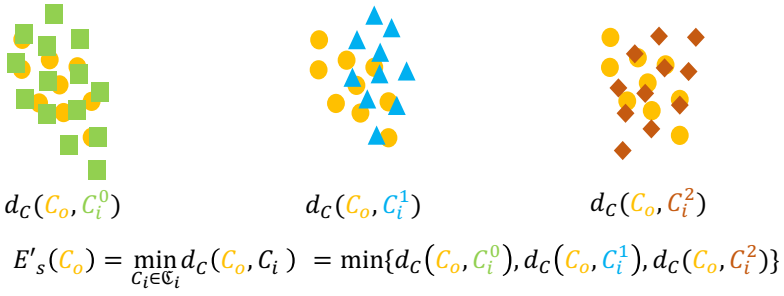
Figure 3.6: *A toy example of link confidence computation.* (a), (b) and (c) show three pairs of matched input and output neighborhoods ($|N_n(s_o, s'_o)| = 3$) where the output neighborhoods are centered at different samples (shown in red). Matched samples are shown in the same colors. Clusters are shown in different sample shapes. For example, output samples/clusters in all three pairs of neighborhoods are shown in the same shapes because they are the same set of samples/clusters. Empty shapes indicate samples that are not used as examples in (d). The computed link confidences among four output samples (in solid shape) from the three neighborhood pairs above are shown in (d). For example, samples ● and ● are matched to the same cluster in (a) but different clusters in (b) and (c), thus $|N_l(s_o, s'_o)| = 1$ and the link $l($●, ●$)$ receive a 1/3 score. Note that we only consider the limited three pairs of neighborhoods (where the output contains s_o and s'_o) for illustration purposes.



(a) Output cluster C_o and its set of matched input clusters $\{C_i\}$



(b) Computing the translation $\mathcal{T}_r$ for each matched input cluster



(c) Shape energy as min. over matched and translated input clusters

Figure 3.7: *Illustration of cluster shape energy computation.* (a) shows an output cluster C_o and its corresponding input clusters $\mathfrak{C}_i = \{C_i\}$. Matched samples visualized in the same colors are computed in the search step (μ_s). Empty shapes indicate input samples from C_i not matched with any samples from C_o. (b) shows the computation of $\mathcal{T}_r$ (Equation (3.17)) which will be used to translate and align input clusters C_i with the output cluster C_o for computing the shape distance $d_C(C_i, C_o)$ between them. In (c), $d_C(C_i, C_o)$ between each pair of input and output clusters is computed via Equation (3.16) using $\mathcal{T}_r$ computed in (b). The shape energy $E'_s(C_o)$ is defined as the minimum of shape distances between the output C_o and input $\{C_i\}$ clusters (Equation (3.15)).

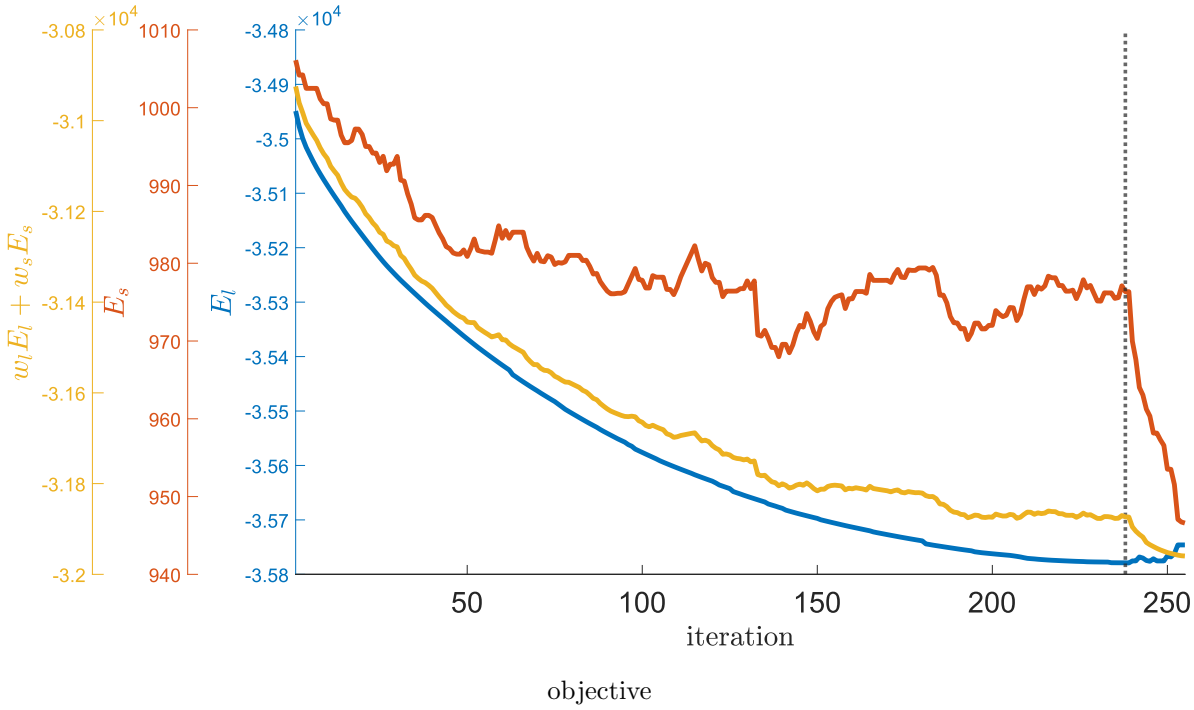
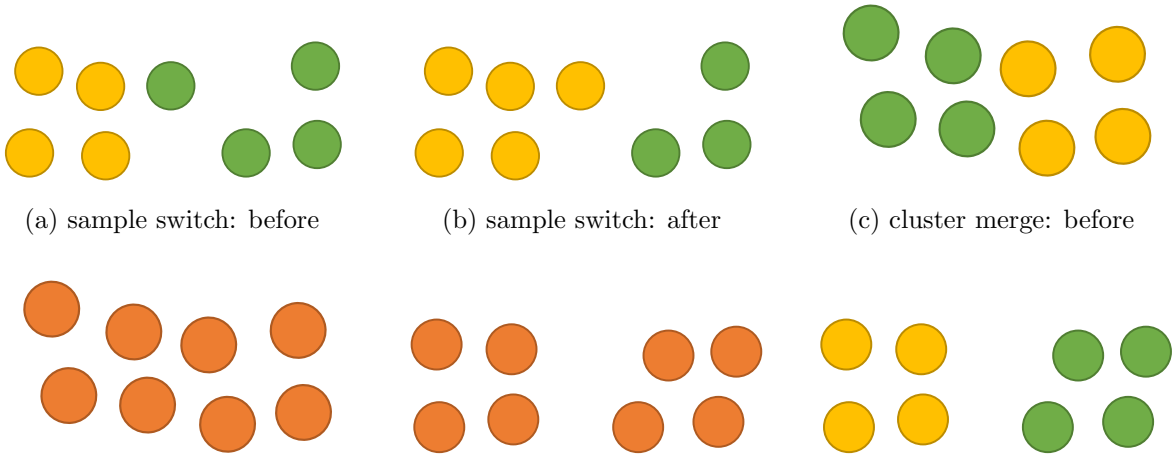


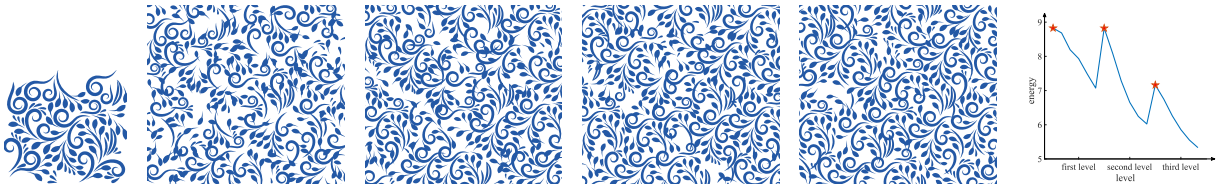
Figure 3.8: *Clustering objective (Equation (3.9)) with respect to number of iterations.* The left side of the vertical dashed line shows the objective optimized by sample switch operators when processing P_s, and the right side shows the cluster split and merge when processing P_C. The blue, dark orange and yellow lines indicate link E_l, shape E_s and total energies $w_l E_l + w_s E_s$.



(a) sample switch: before (b) sample switch: after (c) cluster merge: before

(d) cluster merge: after (e) cluster split: before (f) cluster split: after

Figure 3.9: *Clustering operators.* In each sub-figure, samples from the same cluster are shown in the same color. (a) and (b) show the sample switching operator where a sample in the green cluster is switched to the yellow cluster. (c) and (d) show the cluster merging operator where two clusters (green and yellow) are merged into one (orange). (e) and (f) show the cluster merging operator where one cluster (orange) is splitted into two (green and yellow).



(a) exemplar (b) initial step (c) first level (d) second level (e) third level (f)

Figure 3.10: *Evolution of patterns in the optimization.* In this example, three hierarchies are used. We show the optimized patterns after the very first optimization step (initial step) and the last steps in each hierarchy. The energy plot is shown in the last column where red stars denote the start of each optimization hierarchy. Energies are normalized using the number of samples within an output neighborhood and the total number of matched neigborhood pairs. See Figures 6.2 to 6.5 in Chapter 7 as well as the supplementary video for more visualizations of pattern optimization. Copyright: (a) Meganathan (stock.adobe.com)

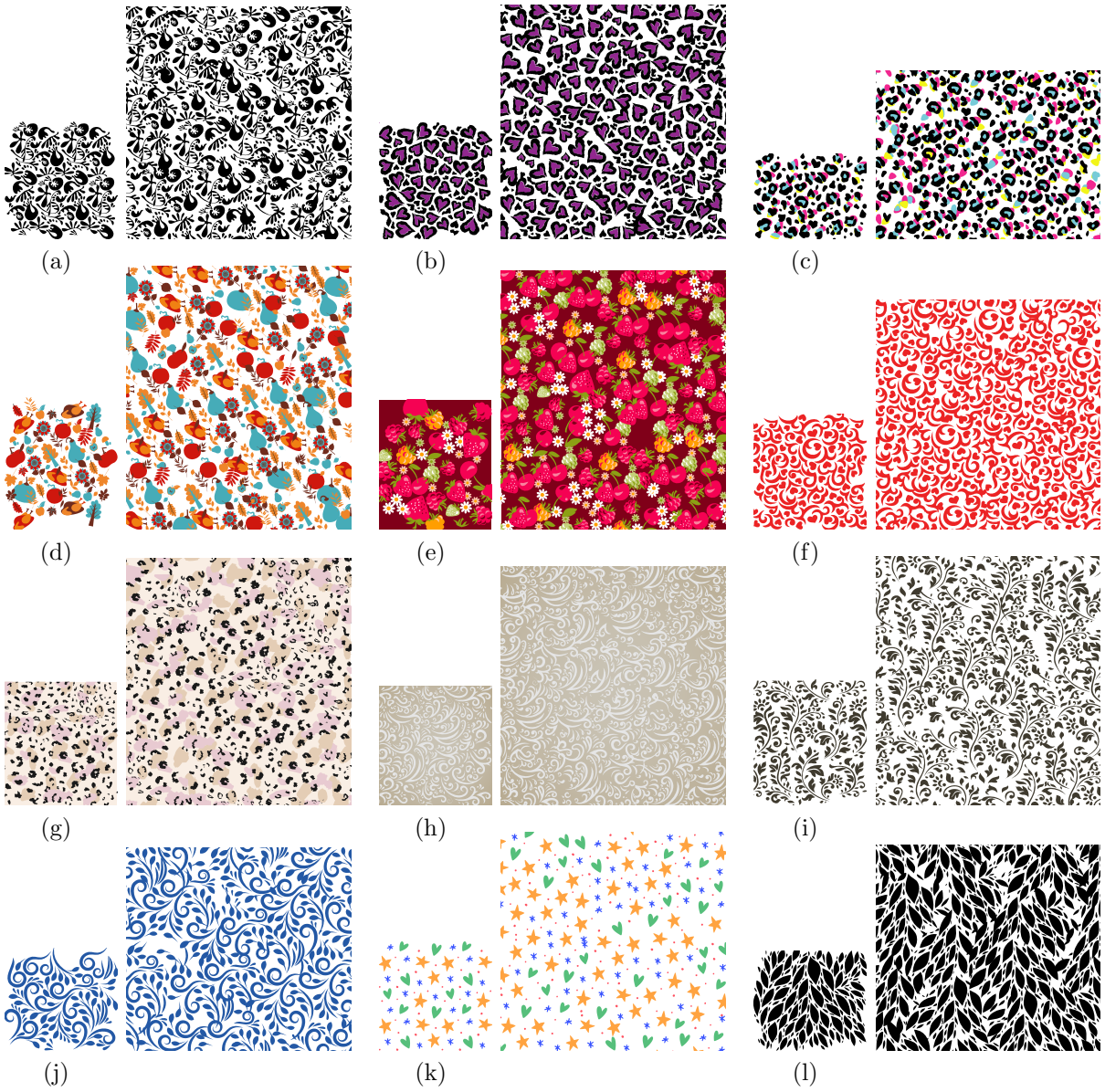


Figure 3.11: *Automatic synthesis results with a variety of exemplar patterns.* Within each group, the input is on the left and our result is on the right. Copyrights: (a) galyna_p (b) leavector (c) Alona Khadzhyoglo (d) olga_milagros (e) galyna_p (f) hoverfly (g) Olgastocker (h) hoverfly (i) hoverfly (j) Meganathan (l) natalyon (stock.adobe.com), and (k) keeeny.am (vecteezy.com)

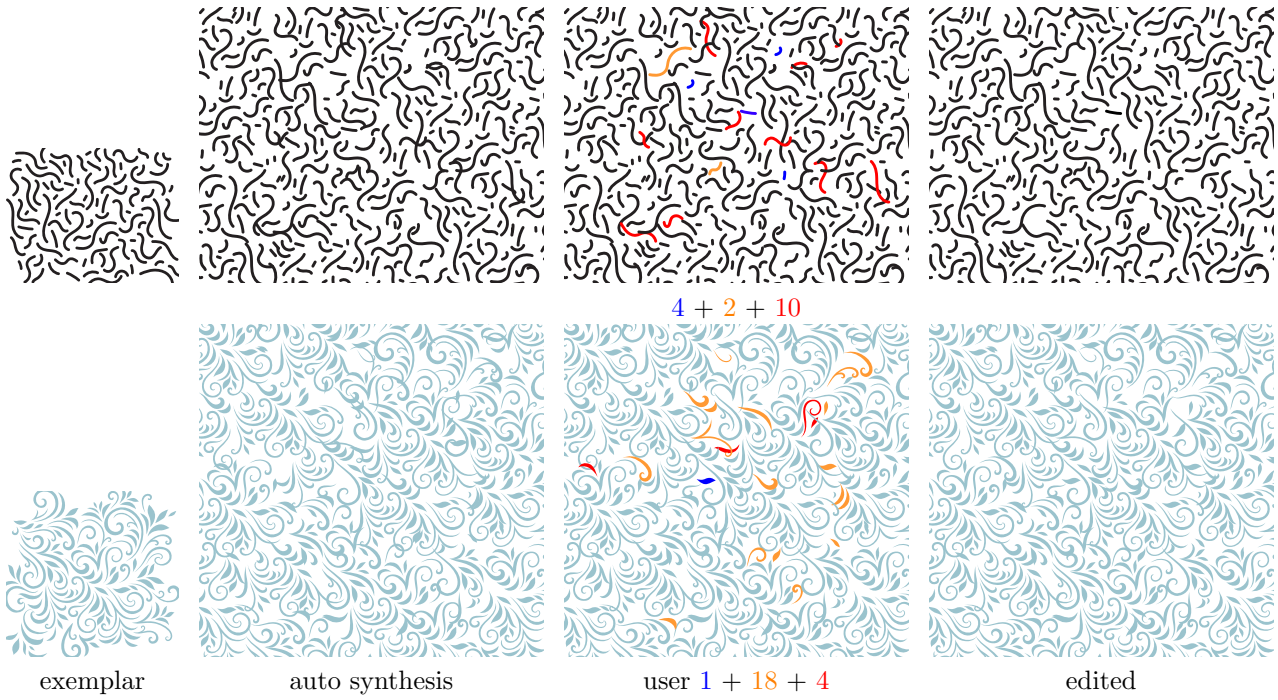


Figure 3.12: *Automatic synthesis with user-editing.* Since the automatic synthesis results are in vector format, users can add (blue), modify (orange), or remove (red) elements from the output patterns. The user editing column also indicates the number of user operations for addition, modification, and removal. Top left ©Frogella.stock, bottom left ©Oleksandra (stock.adobe.com).

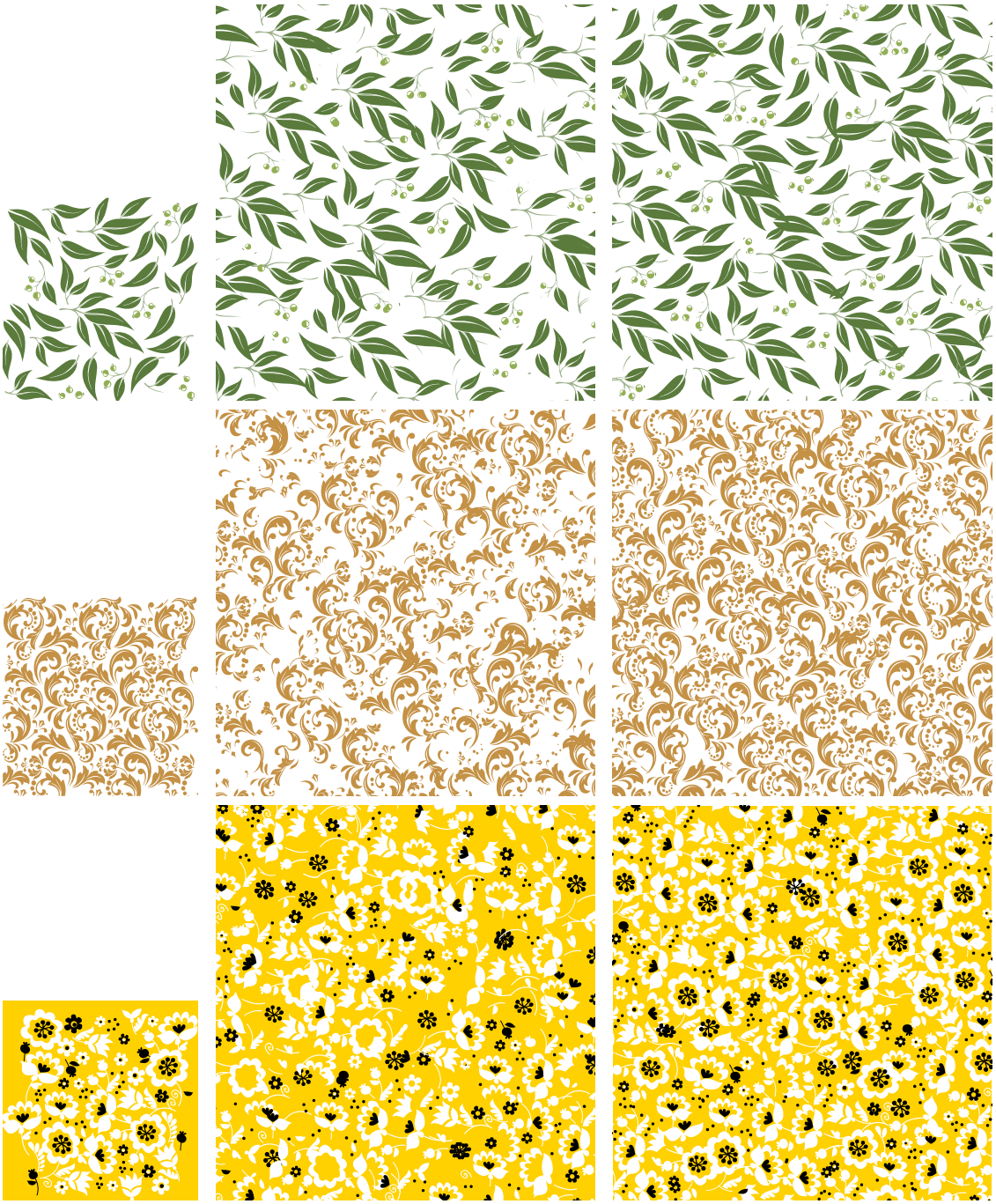


Figure 3.13: *Comparisons against improved [2] with baseline clustering.* The improved version of [2] uses spectral clustering [90] as the basic clustering algorithm; each input element is represented as a fully connected graph, and the output edge confidences in [2] are used as the weights for the weighted similarity graphs in [90]. In addition, clusters are used during neighborhood search, as in Equations (3.2) and (3.3). The final shapes are reconstructed as described in Section 3.6.6.

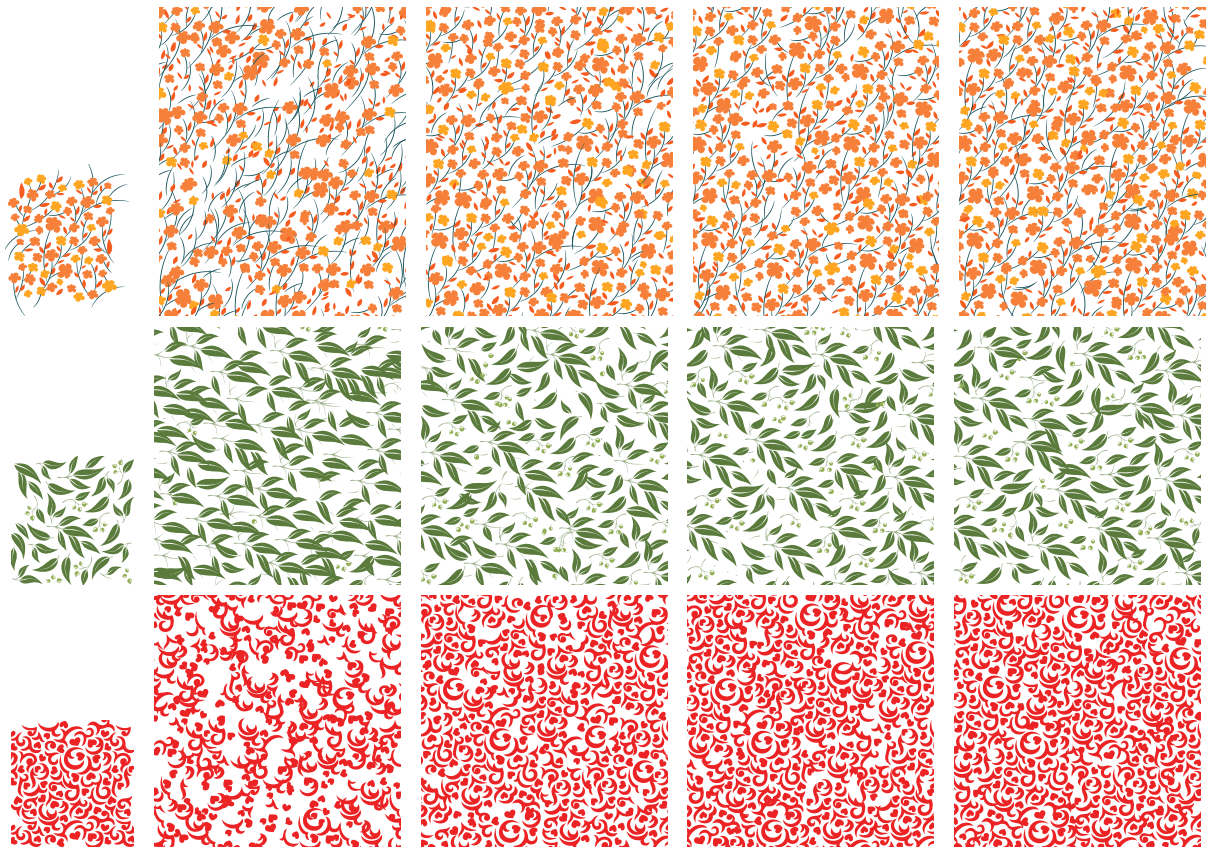


Figure 3.14: *Ablation study.* Without using sample features which contain cluster information ($w_{\mathbf{f}} = 0$) in the search step, the synthesis would generate results that tend to be more random. Without link/shape energy E_l/E_s in the clustering objective (Equation (3.9)), the results tend to have empty and broken regions as shown in the third/forth column. Our full results are shown in the last column.

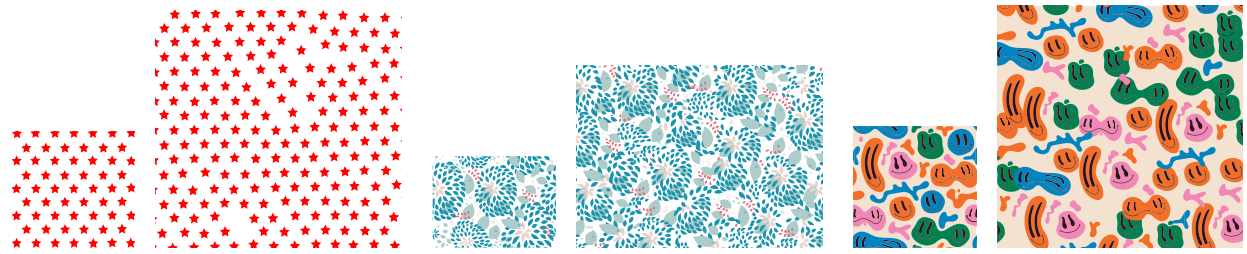


Figure 3.15: *Limitations.* Our method is unable to handle patterns with (a) regular or (c) non-local structures due to the inherent limitations of patch-based synthesis method. Our method requires the input to contain sufficient spatial repetitions and could not handle small tiles with little spatial repetitions (e). Our current implementation has yet to consider toroidal boundary conditions and thus the outputs will not tile seamlessly. Copyrights: (c) Eva Kali, (e) Dariia (stock.adobe.com).

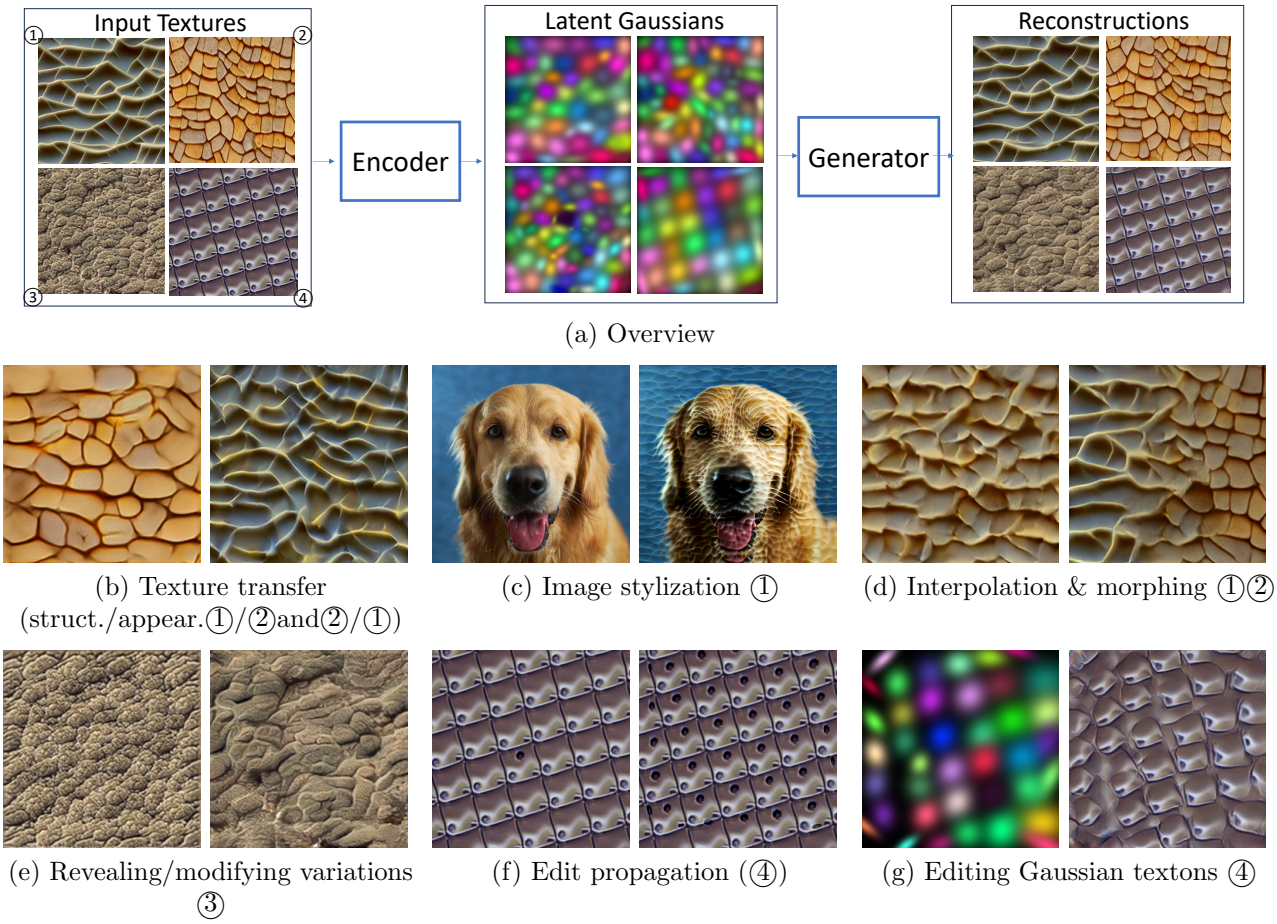
Chapter 4: Compositional Neural Textures

4.1 Introduction

Texture plays a vital role in enhancing the visual richness and overall appeal of various forms of visual content, including real photographs and computer-generated imagery. Textures provide intricate details, patterns, and surface characteristics that contribute to the overall perception and aesthetics of an image. In practice, users often need to create or adjust textures in an image, yet texture editing can be a challenging and time-consuming process, often requiring laborious manual adjustments due to the repetitive details.

Previous research in texture manipulation has explored various applications, ranging from global control [79, 80, 51], to regional variations [101, 102, 103, 104, 105], and individual textons [106]. These approaches rely on domain-specific priors for different applications or categories of textures. Ideally, we would like to have a general-purpose approach that can be applied to a wide range of textures and editing tasks across different scales.

In this chapter, we introduce a texture representation that decomposes textures into a composition of recurring neural elements (Figure 4.1a), disentangling the geometry and appearance of each element. We obtain this representation by developing an autoencoder architecture that is trained in a fully unsupervised manner. Our representation is human-intelligible, applicable to a wide range of textures that exhibit sufficient self-repetitions,



(a) Overview

(b) Texture transfer (struct./appear.①/②and②/①)

(c) Image stylization ①

(d) Interpolation & morphing ①②

(e) Revealing/modifying variations ③

(f) Edit propagation (④)

(g) Editing Gaussian textons ④

Figure 4.1: *Compositional neural representation for texture editing.* (a) We present a neural representation that decomposes textures into geometric elements and appearance features in a fully unsupervised manner. Geometric elements are represented by Gaussian functions computed from an intermediate segmentation corresponding to individual textons, and appearance is represented by an appearance feature vector for each Gaussian. Our network architecture and objectives are designed to encourage extraction of recurring texture elements (textons) and separation of geometry/structure and appearance, while preserving faithful reconstruction of the original texture. This facilitates a variety of editing operations and synthesis applications, such as transferring appearance from one texture towards the structure of another texture (b) or image (c), texture interpolation and morphing (d), manipulating the variations within a texture (e), edit propagation (f), and direct manipulation (e.g. moving and transforming) of the textons (g).

and enhancing a variety of editing techniques (Figure 4.1). Textures can be edited by manipulating the compositional neural representations in deep latent space, and edited textures can be efficiently generated by feeding the manipulated representations through a generator (decoder) network in a feed-forward manner. The supported editing operations include, but are not limited to, transferring appearance from an image texture to another image (Figures 4.1b and 4.1c), diversifying textures (Figure 4.3), texture interpolation and morphing (Figure 4.1d), revealing/modifying texture variations (Figure 4.1e), edit propagation (Figure 4.1f), texture animation (Figure 4.13), and direct manipulation of textons (Figure 4.1g).

Our main idea of representing textures using a compositional neural model is rooted in the understanding that textures are characterized by their recurring patterns, which can vary from regular to irregular formations. Within each texture, basic recurring patterns are known as textons, acting as building blocks that define the texture's unique appearance and structure. However, extracting textons from diverse textures poses considerable challenges due to the ambiguous nature of the texton concept, let alone defining them rigorously in mathematical terms. One possible texton extraction approach is to use supervised learning, for example, by training image segmentation networks. While supervised approaches have achieved state-of-the-art performance for extracting well-defined concepts from images [107, 108] by learning from massive, human-annotated datasets, they are not suitable for extracting textons. Because it is unclear how to consistently annotate textons within a texture due to their ambiguous definition, it is hard to obtain high-quality datasets required for supervised training. Furthermore, the process of annotating any dataset can be both costly and time-consuming. Therefore, we develop a fully unsupervised learning

approach to extract neural textons.

First, we propose a set of desired properties of (neural) textons detailed in Section 4.3.1, which we leverage to effectively train the autoencoder (Figure 4.1a), including 1) discrete objectness, 2) spatial compactness, 3) reshuffling invariance and 4) transformation equivariance. These poperties enable the textons to represent diverse sets of textures while maintaining human interpretability for intuitive texture editing applications. Second, we represent each neural texton as a 2D Gaussian with an associated appearance feature. A Gaussian approximates a texton's shape via its spatial support, and the Gaussian appearance feature encodes the texton's exact appearance. This seperation facilitates the enforcement of geometric transformation equivariance. Further, the separate encoding of rough spatial and detailed appearance information in Gaussian covariances and associated features offers comprehensibility and intuitve user control over textons [109, 110]. Third, we develop unsupervised training methods to instill the desired properties in neural textons. For each desired property we construct a corresponding loss function that, when minimized, ensures the incorporation of that property. In particular, the reshuffling invariance property dictates that randomly reshuffling the appearances of textons, as encoded by Gaussian appearance features, should not modify the texture's overall appearance. In line with this, we specifically design one of the training losses to assess the texture similarity between the input texture, encoded as Gaussians, and an image reconstructed from the Gaussians with reshuffled appearance features. Moreover, we observe that the training losses may present contradictory behaviors, where minimizing one could lead to explosion of another, and address this issue by carefully adjusting the training schedules to achieve a balanced trade-off between these losses.

We evaluate our proposed compositional neural texture representation using a variety of texture editing applications, ablation studies, and comparative analysis against existing methods. The results demonstrate that our approach is versatile in various texture editing applications and surpasses existing methods in texture analysis, modeling, and specific texture editing tasks.

4.2 Related Work

4.2.1 Neural Texture Representations

Recent advancements in deep texture representations have yielded impressive results in various applications, including texture classification [111, 112, 113], segmentation [108, 114], synthesis [115, 92], interpolation [103, 116, 117] and rectification [118]. However, these representations, while engineered to encode a broad spectrum of appearance information, exhibit very limited or no disentanglement when applied to perception and uncontrolled synthesis tasks. This entanglement considerably limits their practical utility in texture editing tasks.

Texture interpolation, a global texture editing task, necessitates a representation that effectively disentangles local and global texture appearances. Bergmann et al. [116] introduce a periodic spatial GAN (PSGAN). PSGAN maps noise to a texture through noise dimensions that are disentangled into global, local, and spatially periodic components. Yu et al. [103] develop an autoencoder to encode an image into a disentangled latent space. This space is divided into components that separately represent local and global aspects of texture appearance. Specifically, the local components of these representations

encode the stochastic variations within a texture, while the global components capture the overall appearance. This separation makes these representations particularly effective for texture interpolation tasks. It facilitates the interpolation of the overall texture appearance, without compromising the inherent stochasticity within textures. Henzler et al. [117] model stochastic textures using noise fields and an MLP decoder. Similarity to [116, 103], this approach effectively disentangles local stochastic and global appearance components. It achieves this by encoding stochasticity with noise fields and the overall appearance with a global texture code. This disentanglement facilitates applications in texture interpolation. However, the level of disentanglement in the representations currently used for interpolation is still inadequate, limiting their extension to other applications across a diverse range of textures.

To address this shortfall and broaden the scope in texture editing, a more effectively disentangled representation is essential. We propose a compositional neural texture representation by modeling textures as neural textons in deep latent space with disentangled structure and appearance. The proposed compositional representation is applicable to a number of texture editing applications and a wide variety of textures.

4.2.2 Texture Editing

Previous research in texture manipulation has explored various applications, such as controlled synthesis [79, 80, 51], interpolation [101, 102, 103, 104], revealing/modifying variations [105], edit propagation [106] and texture weathering [104]. These approaches rely on domain-specific priors for specific applications or categories of textures. They leverage

image smoothness [119] or patch similarity priors [120, 121], or procedural noise [117] defined in image pixel space [120, 106], hand-crafted feature space [86, 122, 101, 104], or learned feature spaces [102, 103, 116, 117, 121], for local [19], stochastic [117], regular/near-regular [122] or non-stationary textures [92].

These texture editing methods can be classified into three categories: non-parametric [119, 105, 19, 86], parametric [101, 122] and more recent deep learning-based approaches [103, 117]. Non-parametric methods, known for their flexibility, can adapt to a diverse range of texture types. However, their editing results are less predictable, and these methods are typically confined to specific applications [19, 104, 106, 105].

In contrast, parametric methods are anchored in well-defined mathematical models, which makes them more interpretable, predictable, and controllable. However, these methods encounter difficulties when dealing with textures that are rich in complexity, encompassing intricate details and sophisticated spatial arrangements [101, 122]. Recent deep learning-based methods employ parametric neural networks, which excel at representing textures, yet they often fall short in providing the interpretability and control crucial for various user editing operations [103, 116, 117].

Another line of active research focuses on transforming image textures into parametric, node-graph-based procedural textures [123, 124, 125, 126]. These procedural textures, once created, offer a professional-grade level of control and quality over texture appearances. However, current methods are limited to a narrow range of texture categories and encounter difficulties in synthesizing graph structures. Moreover, the accuracy of the generated graphs and their parameters falls short, failing to accurately and faithfully reproduce the appearance features of the input image textures.

Consequently, existing approaches are either constrained to a limited set of texture editing capabilities, or their applicability tends to be confined to specific types of textures. Our proposed approach utilizes an autoencoder network characterized by a disentangled, compositional latent space. This latent space offers interpretability and expressiveness, unlocking a wide range of editing capabilities for diverse sets of textures.

4.2.3 Compositional Neural Representations

In certain data domains such as programming languages, vector graphics, social networks, and engineering designs, the underlying compositional structure is either inherently present or can be parsed using specific rules. However, in other domains like 2D images and 3D reconstructions, extracting and representing compositional structures remains a challenging task due to its elusive nature.

To address this, compositional neural representations have been proposed, utilizing a neural latent space to represent various data types compositionally. This approach has been explored in several raster data types, such as 2D images [110, 127, 128, 129], videos [130], 3D shapes [109, 131], and scenes [132, 133]. These representations are derived using various methodologies, ranging from unsupervised [110, 127, 128, 129] and semi-supervised [130], to fully supervised methods [131]. The structures of these neural representations include disentangled object slots [127, 128], spatial points [129], graphs [134, 135], Gaussians [109, 110], and image segments [136].

In this work, we introduce a compositional neural representation for image textures. This representation is characterized by compositions of Gaussians within a disentangled

neural latent space, and is learned without external supervision.

4.3 Method

4.3.1 Overview

At the core of our approach is a texture autoencoder with a disentangled, compositional latent space, trained in a fully unsupervised manner. The latent space consists of 2D Gaussians with neural appearance features, each representing a texton. The Gaussian mean and covariance encode the location and approximate (elliptical) shape of a texton. In addition, each Gaussian has an associated neural feature vector that encodes the detailed appearance of a texton. Users can edit textures by manipulating individual textons represented as Gaussians in the latent space, such as editing their centers, covariances, and appearance features.

Figure 4.2 shows an overview of our autoencoder and unsupervised training strategy, consisting of three major components: autoencoding, appearance reshuffling, and disentangling. In autoencoding (Figure 4.2 autoencoding), the encoder takes an input texture I and generates its latent Gaussians $\{\mathbf{g}\}$. Then, the decoder employs a Gaussian splatting layer h to convert latent Gaussians into a feature map [110] that is fed into the generator $\mathcal{G}$ to reconstruct the input texture I_r. However, autoencoding is not sufficient to reliably learn the extraction of textons as well as separation of texton geometry and appearance in a fully unsupervised manner. While textons are not well-defined both conceptually and mathematically, our unsupervised approach defines and extracts textons based on a set of four desired properties. We then design the appearance reshuffling and disentangling

components to maintain these properties.

Property 1 (Discrete Objectness)**.** *A texton embodies a distinct element within the image that can be identified as either present or absent in its entirety, and a texture can be fully described by a discrete set of textons. This discrete set representation facilitates intuitive and interpretable image editing, allowing users to engage in a more natural and user-friendly editing experience.*

Property 2 (Spatial Compactness)**.** *A neural texton describes the spatial extent and appearance of a* ***local*** *region within a texture.*

Property 3 (Reshuffling Invarance)**.** *Textures inherently possess a stochastic characteristic that exhibits spatial stationarity. Random reshuffling of appearances among a set of given textons should span the stochastic space of a texture without modifying the texture's overall appearance. That is, reshuffled texton appearances should result in a different version of the original texture.*

Property 4 (Transformation Equivariance)**.** *When neural textons undergo a spatial transformation, the resulting image should reflect a consistent and corresponding transformation, preserving the spatial relationships and patterns of the textons.*

Our approach promotes Property 1 (Discrete Objectness) and Property 2 (Spatial Compactness) by applying corresponding compactness L_C and entropy L_E losses in the autoencoding part (Figure 4.2 autoencoding). In addition to the losses, we employ the inherently compact Gaussians to represent textons. Aligned with Property 3 (Reshuffling Invarance), the reshuffling component modifies the Gaussian appearance features (Figure 4.2 reshuffling). These features are associated with input latent Gaussians $\{\mathbf{g}\}$, which are derived by encoding the input texture. The generator $\mathcal{G}$ generates a new

image $\tilde{I}_r$ from the reshuffled latent Gaussians $\{\tilde{\mathbf{g}}\}$. The generated image $\tilde{I}_r$ is expected to closely resemble the input texture I in terms of texture similarity. The disentangling component (Figure 4.2 disentangling) is to promote PROPERTY 4 (TRANSFORMATION EQUIVARIANCE). We transform the input texture and encode the tranformed texture $\hat{I}$ as latent Gaussians $\{\hat{\mathbf{g}}\}$. The same transformation T is applied directly to input latent Gaussians $\{\mathbf{g}\}$. In accordance with PROPERTY 4 (TRANSFORMATION EQUIVARIANCE), we expect that the Gaussians $\{\hat{\mathbf{g}}\}$ derived from transformed input textures should closely match the transformed Gaussians $T(\{\mathbf{g}\})$ from the original input textures.

We next detail autoencoding including encoding textures as Gaussians (Section 4.3.2), Gaussian splatting (Section 4.3.3), and decoding splatted Gaussians as textures (Section 4.3.4). In Section 4.3.5, we then introduce appearance reshuffling (Section 4.3.5.1) and disentangling (Section 4.3.5.2).

4.3.2 Encoding Textures as Gaussians

4.3.2.1 Latent Gaussians

The encoder $\mathcal{E}$ maps an input texture $I \in \mathbb{R}^{H \times W \times 3}$ into our structured latent representation $\mathbf{z}$, a composition of Gaussians $\{\mathbf{g}_i\}_{i=1}^{n}$ where each Gaussian $\mathbf{g}_i$ represents a texton and n is the (maximum) number of Gaussians. To facilitate learning the optimal number of Gaussians required for a texture, we assign a weight parameter to each Gaussian $\delta_i \in [0, 1]$. As we model textures as a discrete set of textons (PROPERTY 1 (DISCRETE OBJECTNESS)), $\mathbf{g}_i$ exists in the set if δ_i is close to 1 and otherwise if δ_i is near 0. Each Gaussian is given by its center $\boldsymbol{\mu}_i$ and covariance matrix $\mathbf{U}_i$ that encodes the approximated shape

of the texton. To enhance the encoding of texton geometry and the subsequent geometry-appearance disentanglement, we include a 2D unit-vector $\nu_i \in \mathbb{R}^2$ ($\|\nu_i\| = 1$) representing the anisotropic direction of textons in addition to Gaussian covariance. Furthermore, $\mathbf{g}_i$ contains an appearance parameter given by a feature vector $\mathbf{f}_i \in \mathbb{R}^{n_f}$, which is used to encode local texton appearance. In summary, our compositional Gaussian representation $\mathbf{z}$ is defined as

$$\mathbf{z} = \{\mathbf{g}_i\}_{i=1}^{n} = \mathcal{E}(I) \tag{4.1}$$

where

$$\mathbf{g}_i = \{\delta_i, \boldsymbol{\mu}_i, \mathbf{U}_i, \nu_i, \mathbf{f}_i\}. \tag{4.2}$$

4.3.2.2 Encoder Network Architecture

To encode image I into latent Gaussians $\mathbf{z} = \{\mathbf{g}_i\}_{i=1}^{n}$, there are two options for generating the Gaussian parameters (Equation (4.2)): the first is to directly predict these parameters via linear or MLP heads; the second, which we choose, is to compute them indirectly from predicted heatmaps or segmentation maps. The second option has demonstrated better performance in previous tasks such as human pose estimation and unsupervised landmark detection [137, 129]. Therefore, the encoder $\mathcal{E}$ includes three components: an image encoder $\mathcal{E}_{FC}$, a segmentation network $\mathcal{E}_s$, and a Gaussian parameter estimation layer $\mathcal{E}_g$. The encoder $\mathcal{E}_{FC}$ is a shallow, fully convolutional network (FCNN) in order to avoid encoding non-local structure information. $\mathcal{E}_{FC}$ takes input images I to generate fea-

ture maps, and has two branches: the appearance branch produces an appearance feature map $\mathbf{F}_a \in \mathbb{R}^{H\times W\times d_a}$ and the geometry branch produces a direction map $\mathbf{V} \in \mathbb{R}^{H\times W\times 2}$. The Mask2Former architecture is used [138] as $\mathcal{E}_s$ and trained end-to-end with the other networks. $\mathcal{E}_s$ takes the image I and generates pixel-wise, softmax-normalized n segmentation masks $\mathbf{S} \in [0,1]^{n\times H\times W}$. For more details about $\mathcal{E}_{FC}$ and $\mathcal{E}_s$, please refer to Section 7.7.1.

Given the n-segmentation masks $\mathbf{S}$, the appearance feature map $\mathbf{F}_a$, and the directional map $\mathbf{V}$, $\mathcal{E}_g$ computes the composition of Gaussians $\{\mathbf{g}_i\}_{i=1}^n$. The mean $\boldsymbol{\mu}_i$ and covariance $\mathbf{U}_i$ of the ith Gaussian $\mathbf{g}_i$ is computed as the average and covariance of image coordinates $\mathbf{p} = [x, y]$ weighted by the ith segmentation mask $\mathbf{S}_i$,

$$\boldsymbol{\mu}_i = \frac{\sum_{\mathbf{p}} \mathbf{p}\cdot \mathbf{S}_i(\mathbf{p})}{\mathfrak{S}_i} \tag{4.3}$$

$$\mathbf{U}_i = \frac{\sum_{\mathbf{p}} (\mathbf{p}-\boldsymbol{\mu}_i)^T(\mathbf{p}-\boldsymbol{\mu}_i)\cdot \mathbf{S}_i(\mathbf{p})}{\mathfrak{S}_i} \tag{4.4}$$

where $\mathfrak{S}_i = \sum_{\mathbf{p}} \mathbf{S}_i(\mathbf{p})$ is the normalization factor computed from the ith segment.

The Gaussian weight δ_i is used to adaptively adjust the number of total Gaussians by indicating the existence of each Gaussian. While this existence weight should ideally be binary, during model training it is crucial to ensure that gradients are effectively back-propagated through the entire network. Hence, we introduce a differentiable, randomized sampling mechanism for computing $\delta_i \sim B(p_i)$ as a sample from Bernoulli distribution B, which takes the value 1 with probability p_i computed as

$$p_i = \frac{\sum_{\mathbf{p}} \mathbf{S}_i(\mathbf{p})\mathbf{S}_i(\mathbf{p})}{\mathfrak{S}_i}. \tag{4.5}$$

Equation (4.5) computes p_i as the average of mask $\mathbf{S}_i$ weighted by $\mathbf{S}_i$. By computing the average of these mask probabilities weighted by the probabilities themselves, we essentially emphasize the higher probabilities, as higher values will contribute more to the average. If the mask has either high probability values (such as in a local region) or zero values (like those outside the region), which might indicate a generally reliable mask, p_i will be high and $\delta_i \sim B(p_i)$, thus the Gaussian $\mathbf{g}_i$ is more likely to exist ($\delta_i = 1$). Conversely, if many pixels have lower but non-zero probability values, implying uncertainty in the segmentation, p_i will be low and the Gaussian $\mathbf{g}_i$ is more likely not to exist ($\delta_i = 0$).

Because sampling from a (categorical) Bernoulli distribution is not differentiable, during training we apply the Gumbel-softmax trick [139] to approximate the hard sampling process from a categorical distribution. During inference, we round the continuous values of $p_i \in [0, 1]$ to obtain binary values $\delta_i \in \{0, 1\}$, aligning with Property 1 (Discrete Objectness). We find that using the categorical sampling mechanism when computing Gaussian weights (Equation (4.5)) generates more aligned and more reliable texton segmentations and thus higher-quality latent Gaussians. We hypothesize that the sampling mechanism inherently promotes weights to converge towards 0 or 1, thereby ensuring binary outcomes, while simultaneously maintaining exploration capabilities through randomized sampling.

Finally, the appearance feature $\mathbf{f}_i$ and direction ν_i are computed as the weighted average (pooling) of image appearance features $\mathbf{F}_a$ and directional map $\mathbf{V}$ within each

image segment as

$$\mathbf{f}_i = \frac{\sum_{\mathbf{p}} \mathbf{F}_a(\mathbf{p})\mathbf{S}_i(\mathbf{p})}{\mathfrak{S}_i} \tag{4.6}$$

$$\nu_i = \frac{\sum_{\mathbf{p}} \mathbf{V}(\mathbf{p})\mathbf{S}_i(\mathbf{p})}{\mathfrak{S}_i} \tag{4.7}$$

followed by normalization $\mathbf{f}_i = \mathbf{f}_i/\|\mathbf{f}_i\|$, $\nu_i = \nu_i/\|\nu_i\|$.

4.3.2.3 Losses

In the process of encoding textures as Gaussisans $\{\mathbf{g}_i\}_{i=1}^n$, we calculate several losses on the intermediate segmentation masks $\mathbf{S}$ during end-to-end training. This includes the entropy loss L_E, which encourages reliable hard segmentation $\mathbf{S}$ and thus PROPERTY 1 (DISCRETE OBJECTNESS),

$$L_E = -\frac{1}{HW} \sum_{\mathbf{p}} \sum_{i=1}^{n} \mathbf{S}_i(\mathbf{p}) \log(\mathbf{S}_i(\mathbf{p})). \tag{4.8}$$

L_E is minimized when the softmax-normalized probabilities in segmentation mask $\mathbf{S}$ are hard with values either 0 or 1. We also apply a compactness loss L_C [140], especially during the initial phase of training on the segmentation masks so that each segment $\mathbf{S}_i$ and thus $\mathbf{g}_i$ is spatially compact and localized, in accordance to PROPERTY 2 (SPATIAL

COMPACTNESS),

$$\mathbf{p}' = \sum_i \boldsymbol{\mu}_i \cdot \mathbf{S}_i(\mathbf{p}) \tag{4.9}$$

$$L_C = \frac{1}{HW} \sum_{\mathbf{p}} \|\mathbf{p}' - \mathbf{p}\|_2^2. \tag{4.10}$$

4.3.3 Gaussian Splatting

After obtaining latent Gaussians $\{\mathbf{g}_i\}_{i=1}^n$, the Gaussian splatting (GS) layer converts $\{\mathbf{g}_i\}_{i=1}^n$ into a feature grid $\mathbf{F}_{gs}$, which is then fed into a generator to reconstruct the input image as I_r. To define the Gaussian splatting function $h : \{\mathbf{g}_i\}_{i=1}^n \rightarrow \mathbf{F}_{gs}$, we first compute the squared Mahalanobis distance M_i^2 of $\mathbf{p}$ to the Gaussian center/mean $\boldsymbol{\mu}_i$,

$$M_i^2(\mathbf{p}) = (\mathbf{p} - \boldsymbol{\mu}_i)\mathbf{U}_i^{-1}(\mathbf{p} - \boldsymbol{\mu}_i)^T. \tag{4.11}$$

Gaussians are derived from pixel-wise normalized image segmentation masks. This approach inherently suggests that each pixel is exclusively associated with a single segmentation mask, and consequently, with a single latent Gaussian. However, in the context of texture editing applications, we permit modifications to these Gaussians during inference, which may result in potential overlaps between the edited Gaussians. For managing these overlaps, we adopt alpha-compositing in Gaussian splatting, as in [110]. Alpha-compositing takes into account the order of Gaussians in instances of overlap. It is important to note that in cases where there is no overlap, for example, during training, the ordering of the Gaussians does not influence the splatting results.

In the alpha-compositing process, it is necessary to define the opacity o for each Gaussian. $o_i(\mathbf{p})$ at $\mathbf{p}$ for i^{th} Gaussian $\mathbf{g}_i$ is calculated as

$$o_i(\mathbf{p}) = \delta_i \exp(-M_i^2(\mathbf{p})). \tag{4.12}$$

Note that we use $-M^2$ instead of $-M^2/2$ in Equation (4.12) to obtain sharper Gaussian splatting. In addition, the compositional alpha $\alpha_i(\mathbf{p})$ at $\mathbf{p}$ for the ith Gaussian is

$$\alpha_i(\mathbf{p}) = o_i(\mathbf{p}) \prod_{j=i+1}^{n} (1 - o_j(\mathbf{p})). \tag{4.13}$$

Finally, the Gaussian splatting function $h : \{\mathbf{g}_i\}_{i=1}^n \to \mathbf{F}_{gs}$ is defined as

$$\mathbf{F}_{gs}(\mathbf{p}) = \sum_i \alpha_i(\mathbf{p}) \mathsf{concat}(\mathbf{f}_i, \nu_i), \tag{4.14}$$

where the function concat concatenates the feature $\mathbf{f}_i \in \mathbb{R}^{n_f}$ and direction vectors $\nu_i \in \mathbb{R}^2$, and $\mathbf{F}_{gs} \in \mathbb{R}^{H \times W \times (n_f+2)}$ is the alpha-composited result of these Gaussian splatted features.

4.3.4 Decoding Images from Splatted Gaussians

4.3.4.1 Network architecture

We adopt the SPADE generator [141] as the generator $\mathcal{G}$ to reconstruct I_r from $\mathbf{F}_{gs}$. The original SPADE generator takes a learned constant feature map and is injected with resized $\mathbf{F}_{gs}$ at multiple levels. Instead, we replace the learned constant feature map with downscaled $\mathbf{F}_{input} = \max_i \alpha_i$ as input to SPADE so it can generate textures of arbitrary

resolutions. For more details, please refer to Section 7.7.1.

4.3.4.2 Losses

The reconstruction is ensured using a mixed reconstruction loss function

$$L_R = w_1 L_1 + w_{LPIPS} L_{LPIPS} \tag{4.15}$$

where $L_1 = \|I - I_r\|_1$ and L_{LPIPS} denotes LPIPS loss [142] computed between I and I_r. We set $w_1 = 2$ and $w_{LPIPS} = 0.2$. In addition, we wish for the reconstructed images I_r to be realistic by employing a discriminator $\mathcal{D}$. We apply the non-saturating adversarial loss on the reconstructed images to train the encoder $\mathcal{E}$ and generator $\mathcal{G}$

$$L_{GAN} = -\log\left(\mathcal{D}\left(I_r\right)\right). \tag{4.16}$$

For details about the discriminator $\mathcal{D}$ and its losses, please see Section 7.7.

4.3.5 Unsupervised Compositional Textons Learning

We utilize Gaussians to represent textons which we wish for exhibiting four properties, as discussed in Section 4.3.1. We have introduced mechanisms to enforce Property 1 (Discrete Objectness) and Property 2 (Spatial Compactness) in Section 4.3.2. In this section, we introduce our approach to inject Property 3 (Reshuffling Invarance) and Property 4 (Transformation Equivariance) into the neural textons.

4.3.5.1 Reshuffling-invariant neural textons

In accordance with PROPERTY 3 (RESHUFFLING INVARANCE), random reshuffling of texton appearances should maintain the original geometry encoding, resulting in a different version of original texture without altering overall texture appearance. More specifically, we randomly reshuffle, denoted as π, appearance feature $\mathbf{f}_i$ associated with $\mathbf{g}_i$ within $\{\mathbf{g}_i\}$ to obtain $\{\tilde{\mathbf{g}}_i\} = \pi(\{\mathbf{g}_i\})$, which is then reconstructed as image $\tilde{I}_r$ that should resemble the input image I. During inference, when dealing with valid discrete textons where the Gaussian weight δ is rounded to 1, we can describe the reshuffling process using a square binary permutation matrix. In the training phase, however, as the network evolves to determine the ideal number of Gaussians by continuously adjusting the Gaussian weight δ, it is crucial to restrict feature reshuffling exclusively to Gaussians that effectively represent individual textons. For example, when the Gaussian weight/existence δ_i/p_i or the area of its texton segmentation mask $A_i = \sum_{\mathbf{p}} \mathbf{S}_i$ is very small or approaching 0 ($p_i \approx 0$ or $\delta_i \approx 0$ or $A_i \approx 0$), we anticipate that the latent Gaussian $\mathbf{g}_i$ and its associated appearance feature $\mathbf{f}_i$ do not effectively represent a texton and should, therefore, be discouraged from appearance feature reshuffling. In this randomized reshuffling process $\{\tilde{\mathbf{g}}_i\}_{i=1}^n = \pi(\{\mathbf{g}_i\}_{i=1}^n)$, say the appearance feature $\mathbf{f}_i$ in $\mathbf{g}_i$ is exchanged with $\mathbf{f}_j$ in $\mathbf{g}_j$, $\mathbf{g}_i/\mathbf{g}_j$ becomes $\tilde{\mathbf{g}}_i/\tilde{\mathbf{g}}_j$ and the resulting appearance feature $\tilde{\mathbf{f}}_j$ in $\tilde{\mathbf{g}}_j$ is calculated as

$$\tilde{\mathbf{f}}_j = \tau_{ij}\mathbf{f}_i + (1 - \tau_{ij})\mathbf{f}_j \tag{4.17}$$

where

$$\tau_{ij} = \left(\min\left(\max\left(\frac{A_i}{A_j}, \frac{p_i}{p_j}\right), 1\right)\right)^{\gamma}. \tag{4.18}$$

Equation (4.17) and Equation (4.18) define the basic pairwise feature reshuffling operation, namely feature swapping, used in the feature reshuffling π process. $\tau_{ij} \in [0, 1]$ is the swapping coefficient between $\mathbf{f}_i$ and $\mathbf{f}_j$. $\gamma = 0.5$ is a non-negative temperature hyperparameter, which will be further discussed later, controlling the softness of the swapping/reshuffling operation. Since δ_i and p_i are similar concepts, we find that using one of them p_i is sufficient for computing τ_{ij} in Equation (4.18). A Gaussian can either effectively represent a neural texton or not. We consider a Gaussian $\mathbf{g}$ to effectively represent a texton if both associated $A \gg 0$ and $p \gg 0$. Conversely, a Gaussian $\mathbf{g}$ is deemed ineffective in representing a texton if $A \approx 0$ or $p \approx 0$. Accordingly, there are four distinct cases to consider when computing the reshuffling coefficient τ_{ij} (Equation (4.17)) between a pair of Gaussians $\mathbf{g}_i, \mathbf{g}_j$, which are detailed below.

1. If both $\mathbf{g}_i$ ($A_i \gg 0$ and $p_i \gg 0$) and $\mathbf{g}_j$ ($A_j \gg 0$ and $p_j \gg 0$) are effective in representing textons, the swapping is effective with $\tau_{ij} \gg 0$.
2. If $\mathbf{g}_i$ is effective ($A_i \gg 0$ and $p_i \gg 0$), but $\mathbf{g}_j$ is not ($A_j \approx 0$ or $p_j \approx 0$), Equation (4.18) gives us $\tau_{ij} = 1$ and thus the swapping is effective. But if $\tilde{p}_j = p_j \approx 0$ ($\tilde{p}_j \in \tilde{\mathbf{g}}_j$), the swapping has no effect on the reconstructed images and is in fact ineffective, because the Gaussian splatting (Section 4.3.3) can ignore Gaussians with $\tilde{p}_j \approx 0$ (and thus $\tilde{\delta}_j \approx 0$).
3. If $\mathbf{g}_i$ is not effective ($A_i \approx 0$ or $p_i \approx 0$), but $\mathbf{g}_j$ is effective ($A_j \gg 0$ and $p_j \gg 0$), there are three subcases: the swapping is not effective with $\tau_{ij} \approx 0$ when both $A_i \approx 0$ and

$p_i \approx 0$, the swapping is effective with $\tau_{ij} \gg 0$ but has no effect on the reconstruction when $A_i \gg 0$ and $p_i \approx 0$, and otherwise the swapping is effective.

4. If neither $\mathbf{g}_i$ ($A_i \approx 0$ or $p_i \approx 0$) nor $\mathbf{g}_j$ ($A_j \approx 0$ or $p_j \approx 0$) are effective, τ_{ij} can be any values between $[0, 1]$. Even if it is possible that $\tau_{ij} \gg 0$, the swapping may still be ineffective if $\tilde{p}_j = p_j \approx 0$ and thus $\tilde{\delta}_j \approx 0$.

Next, we discuss the effect of γ. When $\gamma = 1$, Equation (4.18) behaves normally; when $\gamma < 1$, reshuffling is harder and tends to include Gaussians that may not effectively represent a texton: an extreme case is when $\gamma = 0$, and the operation is equivalent to hard, discrete reshuffling because we always have $\tau_{ij} = 1$ with $\tilde{\mathbf{f}}_j = \mathbf{f}_i$.

We can describe the randomized reshuffling process π in a matrix form by extending pairwise feature swapping (Equation (4.17)). π can be described as a convex combination of both unshuffled and hard reshuffled (permuted) features with the coefficient matrix τ of size $n \times n$ as

$$\tilde{\mathbf{F}} = \tau\mathbf{F} + (1 - \tau)\mathbf{P}\mathbf{F} \tag{4.19}$$

where $\mathbf{F} = [\mathbf{f}_1^T, \mathbf{f}_2^T, \ldots, \mathbf{f}_n^T]^T$ are the unshuffled features, $\mathbf{P} \in \mathbf{P}_n$ is a random permutation matrix, and $\mathbf{P}_n$ is the set of all permutation matrices of size $n \times n$. The multiplication $\mathbf{P}\mathbf{F}$ yields the hard reshuffled features.

In accordance with Property 3 (Reshuffling Invarance), appearance feature reshuffling should not alter overall texture appearance. To this end, we calculate texture loss L_T between the original input I and image $\tilde{I}_r = \mathcal{G}\left(h(\{\tilde{\mathbf{g}}_i\})\right)$ reconstructed from Gaussians $\{\tilde{\mathbf{g}}_i\}$ with reshuffled features by using the patch-based texture loss defined in [121]. In

addition, we use non-saturating GAN losses on image patches [143] to train $\mathcal{E}$ and $\mathcal{G}$

$$L_{PGAN} = -\mathsf{mean}\left(\log\left(\mathcal{D}_p\left(\mathsf{randcrop}(\tilde{I}_r)|\mathsf{randcrop}(I)\right)\right)\right) \tag{4.20}$$

where the function randcrop indicates random image cropping. randcrop crops random image patches of size $\frac{H}{2} \times \frac{W}{2}$. $\mathcal{D}_p$ is a patch discriminator performing on local patches of $\mathsf{randcrop}(\tilde{I}_r)$. Given real cropped patches $\mathsf{randcrop}(I)$ from input texture I as conditions, $\mathcal{D}_p$ is optimized to discriminate if patches are from the same texture I (real) or not (fake), and $\mathcal{E}$ and $\mathcal{G}$ are optimized to fool $\mathcal{D}_p$ by generating $\tilde{I}_r$ with similar local patches as I. In the implementation, the real patches are concatenated to real/fake patches before feeding into $\mathcal{D}_p$. 8 crops are made for each (real or fake) image. The function mean calculates the mean of a vector because $\mathcal{D}_p$ executes on patches and returns a vector of predictions. Please refer to Section 7.7 for more implementation details.

4.3.5.2 Equivariance of geometric transformation

Per Property 4 (Transformation Equivariance), our objective is to achieve consistent results by first extracting the Gaussians and then applying transformations, as compared to first transforming the image and subsequently transforming the Gaussians. The consistency guarantees that the latent Gaussians effectively separate the encoding of texton geometry and appearance, and maintain robustness against transformations applied to the image. Concretely, a consistency loss L_{Csis} is defined to measure the discrepancies between two sets of Gaussians $T(\mathcal{E}(I))$ and $\mathcal{E}(T(I))$. The first set, $T(\mathcal{E}(I))$, is derived by initially extracting Gaussians from the input texture I using our encoder $\mathcal{E}$, and sub-

sequently applying the transformation T; the second set, $\mathcal{E}(T(I))$, is obtained by first applying the transformation T to the input texture I, and then extracting Gaussians with the encoder $\mathcal{E}$. A matching σ is required to compute the consistency loss between the two sets of Gaussians. Before presenting the formal definition of the consistency loss L_{Csis}, we first outline the procedure for computing σ between, in order to simplify notations, two arbitrary sets of Gaussians $\{\mathbf{g}_i\}_{i=1}^n, \{\mathbf{g}'_j\}_{j=1}^n$.

As in [144], we use the Hungarian algorithm [42] to compute the matching σ. The Hungarian algorithm processes a cost matrix, where each entry specifies the cost associated with matching a Gaussian $\mathbf{g}_i$ from the set $\{\mathbf{g}_i\}_{i=1}^n$ to a Gaussian $\mathbf{g}'_j$ from the other set $\{\mathbf{g}'_j\}_{j=1}^n$. The entry $\mathbf{C}_{ij}$ at i^{th} row, j^{th} column of the cost matrix $\mathbf{C} \in \mathbb{R}^{n\times n}$ is computed from $\mathbf{g}_i$ and $\mathbf{g}'_j$ as

$$\begin{aligned}\mathbf{C}_{ij} =& w_{ij}\beta_{ij}(\lambda_{\boldsymbol{\mu}}\|\boldsymbol{\mu}_i - \boldsymbol{\mu}'_j\|_2 + \lambda_{\mathbf{U}}\|\mathbf{U}_i - \mathbf{U}'_j\|_2 + \lambda_{\nu}\|\nu_i - \nu'_j\|_2 \\ &+\lambda_{\mathbf{f}}\|\mathbf{f}_i - \mathbf{f}'_j\|_2) + \lambda_p\|p_i - p'_j\|_2 + \lambda_{\mathbf{S}}\|\mathbf{S}_i - \mathbf{S}'_j\|_2\end{aligned} \tag{4.21}$$

where $w_{ij} = p_i p'_j$ is employed to ignore low-existence Gaussians when computing differences in "non-existence" Gaussian parameters $\boldsymbol{\mu}, \mathbf{U}, \mathbf{f}, \nu$, and $\beta_{ij} \in [0.1, 1]$ is the weight for reducing the impact of matching lower-accuracy textons near the image boundaries. β_{ij} is lower for pairs of Gaussians located near image boundaries and equals 1 for other Gaussian pairs. The term computing the difference in p encourages the number of Gaussian textons to be consistent across image transformations, and the difference in $\mathbf{S}$, which can increase the matching robustness, promotes equivariant segmentations. Then the matching cost between two sets of Gaussians $\{\mathbf{g}\}$, $\{\mathbf{g}'\}$ is computed as the minimum sum of matching

cost between the two sets

$$\mathcal{C}(\{\mathbf{g}_i\}_{i=1}^n, \{\mathbf{g}'_j\}_{j=1}^n) = \min_{\sigma} \sum_{i,\sigma(i)} \mathbf{C}_{ij}. \tag{4.22}$$

Using the matching cost defined between two arbitrary sets of Gaussians, we define the consistency loss between $T(\mathcal{E}(I))$ and $\mathcal{E}(T(I))$ as

$$L_{Csis} = \mathcal{C}\left(T\left(\mathcal{E}(I)\right), \mathcal{E}\left(T(I)\right)\right). \tag{4.23}$$

We filter out Gaussians and segmentation masks transformed outside of image boundaries prior to applying Equation (4.21). In this chapter, we set $\lambda_{\boldsymbol{\mu}} = 1.2$, $\lambda_{\mathbf{U}} = 2$, $\lambda_{\mathbf{f}} = 10$, $\lambda_{\nu} = 0.01$, $\lambda_p = 4$, $\lambda_{\mathbf{S}} = 200$.

In addition, we calculate the reconstruction loss L'_R, using the same formulation as in Equation (4.15), but between transformed images $\hat{I} = T(I)$ and images reconstructed from $T\left(\mathcal{E}(I)\right)$. Again, we mask out image regions transformed outside image boundaries.

4.3.6 Overall Losses and Training Procedure

To sum up, we train the generation network ($\mathcal{E}, h$ and $\mathcal{G}$) with the following losses

$$\begin{aligned} L = L_R + L'_R + w_E L_E + w_C L_C + w_{Csis} L_{Csis} + w_T L_T + \\ w_{GAN} L_{GAN} + w_{PGAN} L_{PGAN} \end{aligned} \tag{4.24}$$

During training, we fix $w_T = 0.01, w_{GAN} = 0.1, w_{PGAN} = 0.1$. We adjust w_E, w_C, w_{Csis} during training to maintain a balance in the unsupervised training process. This ensures

that the reduction of one loss does not disproportionately occur at the expense of the others.

We train our model using 128×128 images for 950K steps and then fine-tune it using 256×256 images for 75K steps. Since some of the network components, including segmentation network $\mathcal{E}_s$ and the discriminator $\mathcal{D}$, cannot take arbitrary-resolution input, we downsample 256×256 images back to 128×128 before feeding into $\mathcal{E}_s$, randomly crop 128×128 patches before feeding into $\mathcal{D}$ and resize as 96×96 before feeding into $\mathcal{D}_p$. $\mathcal{E}_s$ is freezed during fine-tuning. We want to emphasize $\mathcal{E}_s$ is primarily employed to generate spatial Gaussian parameters. In this context, the resolution of the segmentation is not a critical factor affecting the overall quality of the output. Our results and comparisons are in 256×256 except ablation studies where 128×128 images are sufficient to show the differences. See Section 7.7 for more implementation details about our network architectures, losses, etc.

4.3.7 Dataset

We collect a high-quality texture dataset consisting of 377K synthetic and 1.2K real stock images. The synthetic data is from a commerical text-to-image diffusion model using 2500 detailed texture descriptions generated with a large language model (LLM). We use 90% images for training and 10% images for testing (applications and analysis). The dataset is further augmented with random affine transformations and color histogram matching with a random reference image from the dataset. Please see Section 7.1 for some random data samples.

4.4 Applications

In this section, we showcase a variety of applications for our compositional neural texture representation, including texture diversification (Section 4.4.1), texture transfer and image stylization (Section 4.4.2), modifying texture variations (Section 4.4.3), texture interpolation (Section 4.4.4), direct manipulation of textons (Section 4.4.5), and animated textures (Section 4.4.6). These applications involve manipulating the compositional latent Gaussians, and efficiently generating output textures by feeding the manipulated Gaussians through the generator network in a feed-forward manner. For reference, the feed-forward processing of a 256×256 image using our autoencoder takes 0.16 seonds on a single NVIDIA A10G (24GB) in PyTorch, and this is achieved without any specific optimizations for speed.

4.4.1 Texture Diversification

Textures can be characterized by their inherent randomness, meaning that there are countless variations of the same texture while maintaining its overall look. Our method treats a texture as a combination of neural textons, and the randomness is (partially) encapsulated in the distribution of appearance features. By reshuffling these appearance features among effective textons (with $\delta = 1$), we can create diverse versions of the same texture, as shown in Figures 4.3d to 4.3f. This method validates that our neural textons accurately encapsulate the texture's composition, showing that feature reshuffling creates a subspace of variations of a texture.

4.4.2 Texture Transfer

Texture transfer combines the structure and appearance of two different textures: the first provides the structure while the second provides the appearance. Our compositional texture representations effectively separate the spatial structure and appearance characteristics within a texture, which is key for successful texture transfer. Our transferred results are created by taking the Gaussians from the structure-providing texture, appearance features from the appearance-providing texture, and we adjust the means of Gaussian appearance features of the structure-providing texture to match those of the appearance-providing texture. Figure 4.4 shows a grid of texture transfer results. Note for texture transfer, we can also adjust other statistics beyond the means. For example, we can directly replace appearance features from structure-providing textures with those from appearance-providing textures. This alternative generates transfer results that preserve less appearance information from the structure-providing textures (see Section 7.4 for comparisons and Figure 4.1b (left)). We will leave studying other transfer mechanisms as future works. Unless otherwise noted, we transfer textures by adjusting means of appearance features by default.

In addition, we can stylize a natural image using textures. While our network is not trained on natural images, a small patch of a natural image can be effectively treated as a texture. Initially, we run our encoder $\mathcal{E}$ on overlapping content image patches to generate compositional Gaussians for each patch. These Gaussian patches are merged in overlapping areas to ensure smooth transitions, similarly to [45]. Concretely, we build correspondences between Gaussians from two patches in the overlapping area, and then

we interpolate their parameters based on the correspondences. Subsequently, we adjust the feature mean of these smoothly merged Gaussians to match that of the appearance-providing texture. Finally, the stylized image is reconstructed from these mean-aligned Gaussians, resulting in a harmonious blend of the texture appearance and structure from the natural image. Figure 4.5 shows the image stylization results. Users can also transfer different textures to image regions specified by control masks, which could be later used to create interactive brush tools. In this case, we adjust the means of appearance features within each image region respectively to match those of the corresponding appearance-providing texture. Figure 4.6 shows the transfer of three different textures to three image regions.

Moreover, users can control the scale of transferred textons, as demonstrated in Figure 4.7 and more in Section 7.4. This control is achieved by varying the scale of Gaussians derived from content image patches. To produce Gaussians on a smaller scale for a content image, we initially sample reduced-size patches from the content image. These smaller patches are then upscaled to the standard resolution of 128×128 pixels, which corresponds to the resolution of input images required for the segmentation network $\mathcal{E}_s$. Post-upscaling, these patches are encoded into latent Gaussians. These latent Gaussians are subsequently downscaled and positioned at the corresponding locations within original sampled patches. They are then merged with Gaussians from other patches to maintain smooth transitions. Conversely, to generate Gaussians on a larger scale, the process is reversed, involving the sampling of larger patches and appropriate scaling adjustments.

4.4.3 Revealing and Modifying Texture Variations

We can modify non-local texture variations by editing variations within the Gaussian's appearance and geometry parameters. To quantify the variations, we first calculate the mean of Gaussian appearance features $\mathbf{f}_i$ and covariances $\mathbf{U}_i$ as

$$\overline{\mathbf{f}} = \frac{\sum_i \mathbf{f}_i \delta_i}{\sum_i \delta_i} \tag{4.25}$$

$$\overline{\mathbf{U}} = \frac{\sum_i \mathbf{U}_i \delta_i}{\sum_i \delta_i}, \tag{4.26}$$

where $\delta_i \in \{0, 1\}$ in the inference process. Equation (4.26) ignores ineffective Gaussians when computing the means. The variations in edited appearance features $\mathbf{f}_i^*$ can be controlled by a coefficient $\Delta_{\mathbf{f}} > 0$,

$$\mathbf{f}_i^* = \overline{\mathbf{f}} + \Delta_{\mathbf{f}}(\mathbf{f}_i - \overline{\mathbf{f}}). \tag{4.27}$$

Note when $\Delta_{\mathbf{f}} = 1$, the features are not edited as $\mathbf{f}_i^* = \mathbf{f}_i$ and the input image is reconstructed. While the differences in appearance features can be computed as simple vector differences, we measure the differences between covariances $\mathbf{U}_i$ and $\overline{\mathbf{U}}$ by estimating the transformation W_i between them using Cholesky decomposition, as

$$L_i L_i^T = \mathbf{U}_i,\ \overline{L}\overline{L}^T = \overline{\mathbf{U}} \tag{4.28}$$

and

$$W_i = L_i \overline{L}^{-1}. \tag{4.29}$$

Then the variations in edited covariances $\mathbf{U}_i^*$ can be controlled by $\Delta_{\mathbf{U}} > 0$,

$$\mathbf{U}_i^* = (W_i)^{\Delta_{\mathbf{U}}} \overline{\mathbf{U}} \left((W_i)^{\Delta_{\mathbf{U}}}\right)^T, \tag{4.30}$$

where $(W_i)^{\Delta_{\mathbf{U}}}$ computes the fractional matrix power of W_i. Note when $\Delta_{\mathbf{U}} = 1$, the covariances are not edited as $\mathbf{U}_i^* = \mathbf{U}_i$ and the input image is reconstructed. Figures 4.8 and 4.9 show the edited image when we reduce and increase the variations in appearances features and covariances (shapes) respectively.

4.4.4 Texture Interpolation

Our representation can be applied for interpolating textures by interpolating latent Gaussians. Given two sets of Gaussians $\{\mathbf{g}_i\}_{i=1}^n$ and $\{\mathbf{g}'_j\}_{j=1}^n$, we first build correspondences ϕ between them ($\mathbf{g}_i$ matches $\mathbf{g}'_{\phi(i)}$), by minimizing the sum of differences between their centers and weights using the Hungarian algorithm [42]. The parameters in interpolated Gaussian $\mathbf{g}_i^\dagger$ are the convex blending of those in $\mathbf{g}_i$ and $\mathbf{g}'_{\phi(i)}$ controlled by a coefficient $\eta \in [0, 1]$. For example, interpolated Gaussian center $\boldsymbol{\mu}_i^\dagger$ is defined as

$$\boldsymbol{\mu}_i^\dagger = (1 - \eta)\boldsymbol{\mu}_i + \eta\boldsymbol{\mu}'_{\phi(i)}. \tag{4.31}$$

Since δ_i^* is supposed to be binary, being either 0 or 1 during inference, we treat the interpolated value as a probablity of being 1 in a Bernoulli distribution and δ_i^* is sampled from the distribution. The interpolated image is reconstructed from the interpolated Gaussians $\{\mathbf{g}_i^\dagger\}$. Figure 4.10 shows interpolated images with various η.

Our method can also generate smooth transition of two textures in one image by spatially interpolating Gaussians, as shown in Figure 4.10f, where η linearly increases from 0 (left) to 1 (right).

4.4.5 Texton-centric Editing

Since we extract individual neural textons, our approach can also enable more direct manipulation of individual textons.

4.4.5.1 Editing Gaussians

Users can move, scale and rotate textons by transforming their geometric parameters ($\mathbf{U}_i$ and ν_i), as shown in Figure 4.11.

4.4.5.2 Edit propagation

When users make edits to a specific area of a texture, these changes can be automatically propagated to other similar areas. This process is demonstrated in Figure 4.12. The procedure begins by calculating the difference image between the edited and the original images. The difference image encodes the edits in the pixel space. Next, the edit propagation involves identifying the specific neural Gaussian texton that corresponds to the edited region which is implemented by querying the nearest Gaussian to the edits. To propagate these edits to another similar region or texton, we 1) apply a transformation to the difference image; this transformation aligns the Gaussian of the edited and the target textons; and 2) add the transformed difference images to the corresponding region of the original

image, which creates the final edit-propagated image. The edits can be consistently and accurately replicated across similar textural features, the locations and shapes of which are identified by extracted neural textons, within the image.

Discussions. We observe that propagating edits directly in the pixel space results in clearer and more precise edits compared to using the latent feature space. This distinction arises from two key factors. First, edits like text or other novel elements, which the network might not have encountered during training, can lead to poor reconstruction quality when processed in the latent feature space. This, in turn, affects the quality of edit propagation. Second, even if edits in the image space can be theoretically represented in the appearance feature space, another issue is that the appearance feature space is not explicitly trained to handle robust feature arithmetic, that is, the image-space differences are not guaranteed to be describable and transferable as differences in the appearance feature space. This lack of specialized training can lead to less accurate or predictable edit propagation in feature space when compared to direct pixel space manipulation.

4.4.6 Texture Animation

In Section 4.4.5.1, we describe how our framework can handle image editing at the texton level, with each latent Gaussian aligned with a corresponding texton in the image. This technique also enables animating Gaussians using flow fields [49, 146], which are then reconstructed as animated textures. Figure 4.13 shows various frames from such dynamic visualizations. Please refer to our supplementary video for dynamic experiences and refer to Section 7.7.3 for further details about the flow fields used in these animations.

4.5 Analysis

We compare our method to related approaches and evaluate our approach via ablation studies, both qualitatively and quantitatively.

4.5.1 Comparison to "Segment Anything"

The fundamental aspect of our methodology lies in its ability to autonomously extract compositional textons from textures using unsupervised techniques. In our encoder $\mathcal{E}$, the unsupervised segmentation network $\mathcal{E}_s$ is used to segment textons and convert them into Gaussians in the latent space for subsequent texture editing applications. One obvious alternative approach to extracting textons is by training a supervised segmentation model tailored for textons. However, the challenge with supervised segmentation, which relies on ground-truth labels, is its incompatibility with the ambiguous nature of textons. Since the concept of a texton is not universally agreed upon, achieving consistency in ground-truth texton labels is difficult. Furthermore, labeling textures can be a labor-intensive process, even more so than natural photos.

To evaluate the quality of our extracted textons, we compare our segmentation against the "Segment Anything" (SA) model [145]. This model, noted for its remarkable ability to segment with zero prior knowledge, has been trained on an extensive dataset of 1 billion masks. Figure 4.14 shows the comparisons with the SA model. See Section 7.5 for more comparisons. These results indicate that SA is incapable of reliably detecting and extracting textons, a fundamental aspect that contributes to the versatility of our method across a variety of applications.

4.5.2 Comparisons on Texture Transfer

In this section, we aim to demonstrate our approach's unique capacity in disentangled structure-appearance texture modeling using latent Gaussians based on segmented structures. To this end, we compare our approach with prior methods [147, 148, 149] tailored for combining appearance from one image and content/structure from another image. Figure 4.15 shows comparisons against prior methods. Our method generates more visually appealing results without clear artifacts, effectively capturing both the structure and appearance from the respective structure-providing and appearance-providing textures. Please see Section 7.5 for more comparisons. Note we not only disentangle structure from texture appearance but also explicitly represent them as different Gaussian parameters, which enables a variety of applications (Section 4.4) beyond texture transfer.

4.5.3 Ablation Studies

We evaluate our method via ablation studies to demonstrate the importance of each individual component in our approach, both qualitatively and quantitatively.

4.5.3.1 Quantitative metrics

For quantitative evaluation, first, we evaluate how robust and meaningful the compositional representation is for different transforms/edits. Since there is no ground-truth benchmarks for texton extractions, we consider to evaluate our method based on a cycle consistency (CC) metric. That is, the estimated compositional representation $\mathcal{E}(I^*)$ of the edited images $I^* = \mathcal{G}(\{\mathbf{g}^*\})$ reconstructed from edited Gaussians $\{\mathbf{g}^*\}$ is supposed to be

consistent to the edited Gaussians $\{\mathbf{g}^*\}$. Formally, CC metric is defined as

$$\text{CC} = L_{Csis}(\{\mathbf{g}^*\}, \mathcal{E}(I^*)) \tag{4.32}$$

where L_{Csis}, the consistency loss to measure differences between two sets of Gaussians during training, is reused. Second, we evaluate the quality of our edited images I^* using Fréchet inception distance (FID) by comparing it to unedited input images I. We also compute CC and FID, between input images and their reconstructions for calibrations, in which case, there is no edits applied to $\{\mathbf{g}^*\}$ and $\{\mathbf{g}^*\} = \{\mathbf{g}\}$ when computing CC. In addition, we calculate a diversity (DVC) metric for measuring the diversity of textures generated from randomly reshuffled appearance features (Section 4.4.1). Please see Section 7.2 for more evaluation details.

4.5.3.2 Ablation studies on losses

We illustrate the significance of each loss term in our model by removing each one and observing the effects. Figure 4.16 presents a visual comparison of the results in terms of segmentation, Gaussians, reconstruction, and the texture transfer application. See Section 7.6 for more visual comparisons for the ablation studies. Without compact loss L_C, our model struggles to extract textons, leading to the entire image being represented by a single segment or Gaussian. The absence of entropy loss L_E results in less defined segmentations and blurry reconstructions. Without consistency loss L_{Csis}, the segmentation degenerates and is always grid-like caused by excessive minimization of compact L_C and entropy L_E losses; while its quality of texture reconstruction is similar to that of using

	Ablation studies								Full
	Absent losses						Absent network components		
	L_C	L_E	L_{Csis}	L_T	L_{GAN}	L_{PGAN}	segmentation	latent Gaussians	
CC (R)	0.15	1.29	1.40	0.99	0.54	1.49	0.62	0.05	0.60
CC (TD)	0.15	1.30	7.82	1.20	0.65	1.63	0.60	0.11	0.99
CC (TT/S)	0.16	1.73	3.28	1.20	0.79	1.70	0.63	0.13	0.99
CC (TT/A)	0.15	1.39	3.29	1.27	0.68	1.95	0.61	0.11	1.04
CC (TE)	2.71	1.50	3.59	1.17	0.73	1.78	0.75	-	1.13
CC (TI)	0.08	4.92	10.34	6.11	5.01	6.02	0.99	-	6.13
FID (R) ↓	67.26	129.29	17.40	53.32	69.30	112.93	219.59	11.93	16.95
FID (TD) ↓	67.26	129.97	59.13	63.59	87.81	116.90	239.18	16.87	44.73
FID (TT/S) ↓	67.11	139.14	53.33	70.01	93.17	122.38	221.78	27.44	41.72
FID (TT/A) ↓	67.26	133.75	53.53	100.80	79.43	130.32	220.76	16.64	53.58
FID (TE) ↓	178.31	139.23	50.81	57.13	77.97	116.61	221.94	-	47.18
FID (TI) ↓	97.35	142.84	134.32	96.35	113.20	153.64	246.39	-	70.92
DVS (TD) ↓	0.0	0.69	4.56	2.30	1.47	1.00	0.87	0.12	2.63

Table 4.1: *Quantitative evaluations of ablation studies.* R, TD, TT, TE, TI stand for reconstruction, texture diversification (Section 4.4.1), texture transfer (Section 4.4.2), texton (Gaussian) editing (Section 4.4.5) and texture interpolation (Section 4.4.4). TT/S and TT/A indicate the metrics are computed between the transferred output (TT) and the structure (S) or appearance (A) image. For TI, metrics are computed between the interpolation using $\eta = 0.5$ and one of its source/target images. FID score is better when it is lower. For CC (cycle consistency), a very low value (e.g. w/o L_C) may suggest the extracted neural textons do not vary regardless of the input textures while a high value could suggest the extracted textons lack consistency in response to edit/transform. A moderate CC value is preferred, as it indicates a balanced and reliable extraction of neural textons that are both responsive to different input texture structures and consistent with respect to edits. Similarly, a moderate DVS (diversity) value is preferred. For example, w/o L_C, DVS equals 0 as a texture is represented by one latent Gaussian; w/o L_{Csis}, DVS is the largest as the network fails to extract correct texton compositions, and therefore random reshuffling of appearance features could destroy the texture's overall appearance. In ablaion studies of network components, without latent Gaussians, it's unclear how to manipulate textons and conduct interpolation on segmentations; even if FID scores might be lower, the editing results may not properly capture texture structures, as shown in Figure 4.17.

full losses, applying its extracted latent Gaussian textons for texture transfer applications generates low-quality, blurry results without capturing structure and appearance respectively from two input images. Without texture L_T and patch GAN losses L_{PGAN}, both of which are applied to encourage similarity of textures between images reconstructed from appearance-reshuffled Gaussians and input images for injecting PROPERTY 3 (RESHUFFLING INVARANCE), the extracted textons are less accurate. The reconstruction quality is also degraded as a result of the disrupted equilibrium in unsupervised training. Without GAN loss L_{GAN}, the resulting reconstructions exhibit more blurry artifacts, which interferes with the texton extraction quality. In addition, we provide detailed quantitative analyses of the significance of various loss terms for different texture applications on the left side of Table 4.1.

4.5.3.3 Ablation studies on neural architectures

The segmentation model and the intermediate compositional Gaussian representation are the two pivotal components in our model architectures. We evaluate the importance of both components below. Figure 4.17 shows our results of the ablation studies. The right side of Table 4.1 reports the quantitative evaluation results.

Disabling segmentation module. We replace the segmentation model with a set of linear layers that directly regress the Gaussian parameters. In the encoder, we replace the segmentation model with transformer + MLP to generate Gaussian parameters. With the absense of segmentations, we replace the losses L_C and L_E originally computed on segmentation masks with similar losses but defined on Gaussians directly. As shown in

Figure 4.17, without the segmentation network, we observe that direct generation of Gaussians cannot converge and does not produce meaningful compositional textons. This is in accordance with what is observed in keypoint detection research where heatmap-based approaches outperform regression-based approaches, since direct regression involves more non-linearity and thus is hard to train even in supervised settings [137, 129].

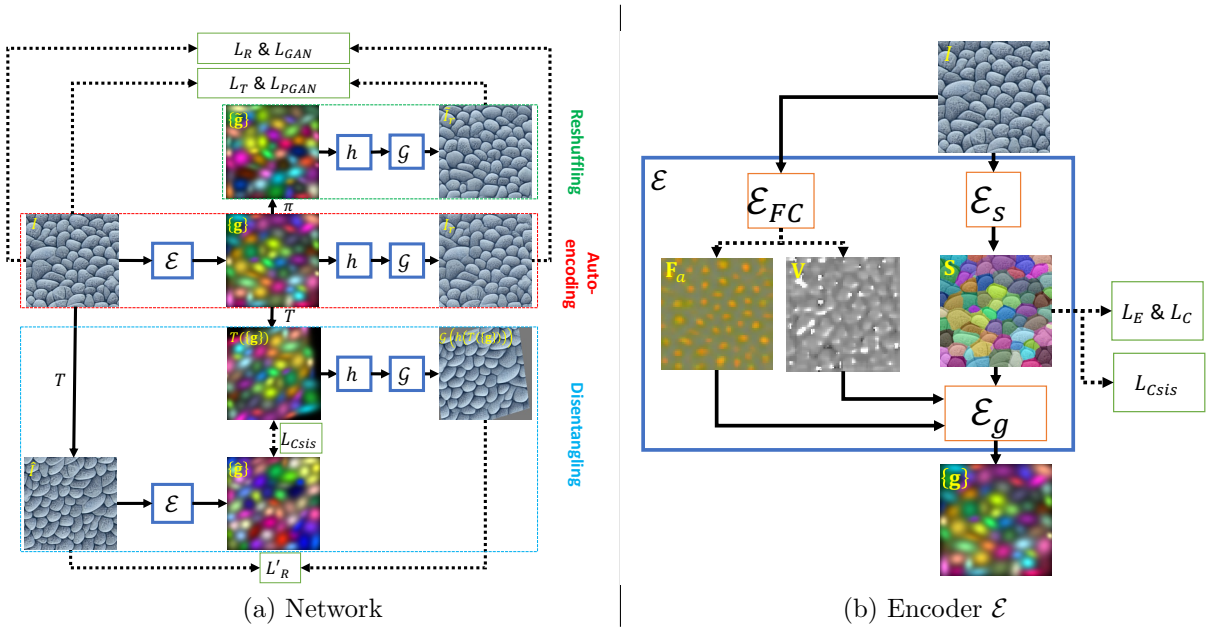
Disabling latent Gaussians. Instead of using the Gaussians, we splat features directly using predicted segmentation masks. Figure 4.17 shows that, without the Gaussian bottleneck, the segmentation network segments as simple grids instead of each individual texton. Also, it is unclear how to directly manipulate textons and conduct interpolation using segmentation masks. Therefore, Table 4.1 does not report the metrics for texton editing and interpolation applications. These results indicate the Gaussian bottleneck not only provides an easy and intuitive way for editing applications, but also serves as an implicit regularization for clean segmentation. We hypothesize this is because without the bottleneck, the model tends to find shortcuts to encode texture appearance information directly via high-dimensional segmentation masks without extracting meaningful texton geometry.

4.6 Limitations and Future Work

Depending on the editing operations used, our current method may struggle with handling patterns with elongated, continuous structures. In these patterns, the notion of textons might not be clear and each elongated structure might be represented by multiple textons in our current implementation as we impose Property 2 (Spatial Compactness) on these learned neural textons. Figure 4.18 shows such an example where texton

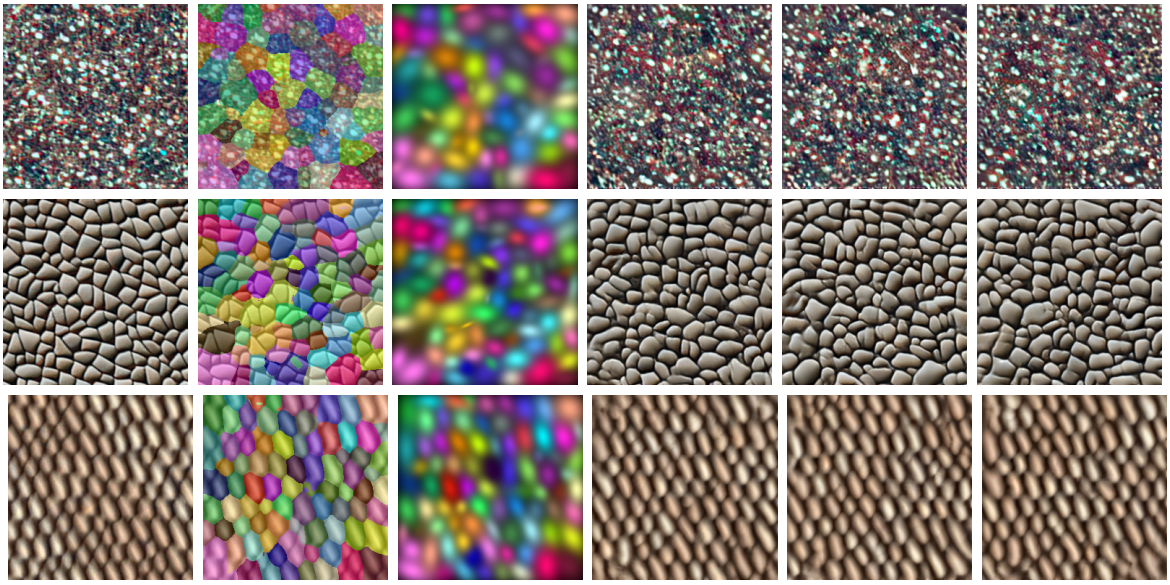
editing can disrupt continuous structures in textures. In addition, our approach relies on the assumption of random reshuffling of texton appearances (PROPERTY 3 (RESHUFFLING INVARANCE)) that can result in a different version of the original texture without altering its overall appearance. Consequently, while our method can extract non-stationary structure of textons within textures, it is suitable for those that exhibit consistent texton appearances throughout the image domain.

In this chapter, we propose a framework for compositional modeling of repetitive patterns in images. Using our framework, we present a broad range of image editing applications using simple, baseline approaches, which could be improved with more dedicated mechanisms in the future. For example, in texture interpolation applications (Section 4.4.4), we could interpolate Gaussians via a softer mechanism using optimal transport instead of via the hard approach by solving assignment problems. Additionally, the potential uses of our approach extend beyond the presented examples, such as texture rectification [118], by analyzing and rectifying Gaussian centers, directions and shapes, and weathering [104], by analyzing and manipulating the appearance features, font stylization by focusing on font shape, and video stylization by imposing intuitive coherence directly on latent Gaussians. Furthermore, our framework could be adapted to model repetitive phenomena in various other data types, such as materials, vector patterns, 3D geometry, and audio.



(a) Network (b) Encoder $\mathcal{E}$

Figure 4.2: *Architecture of the proposed method.* (a) Our network is composed of autoencoding (Sections 4.3.2 to 4.3.4), texton reshuffling (Section 4.3.5.1), and structure-appearance disentangling (Section 4.3.5.2). In the autoencoding part, the encoder $\mathcal{E}$ encodes an image I into latent Gaussians $\{\mathbf{g}\}$. The Gaussians are splatted h onto feature maps, which are fed into the generator $\mathcal{G}$ to reconstruct the original input as I_r. We calculate the reconstruction L_R and GAN loss L_{GAN} between I and I_r to ensure accurate reconstruction. In the texton reshuffling part, we randomly reshuffle, denoted as function π, appearance features within $\{\mathbf{g}\}$ to produce $\{\tilde{\mathbf{g}}\}$, which are subsequently reconstructed as $\tilde{I}_r$. We compute the texture loss L_T and the patch GAN L_{PGAN} between I and $\tilde{I}_r$, both of which are used to measure texture similarity. This is done with the expectation that reshuffling texton appearances will not alter a texture's overall appearance (Property 3 (Reshuffling Invarance)). In the disentangling part, we transform image I to obtain $\hat{I}$ via transformation T. $\{\hat{\mathbf{g}}\}$ is obtained from encoding $\hat{I}$. The same transformation T is applied to $\{\mathbf{g}\}$, obtaining $T(\{\mathbf{g}\})$. We compute the consistency loss L_{Csis} to enforce the similarity between latent Gaussians $\{\hat{\mathbf{g}}\}$ and $T(\{\mathbf{g}\})$, and aim for reconstructing $\hat{I}$ from $T(\{\mathbf{g}\})$ by minimizing reconstruction loss L'_R between $\hat{I}$ and $\mathcal{G}(h(T(\{\mathbf{g}\})))$. disentangling encourages the encoding of image transformation within the transformation in Gaussians (Property 4 (Transformation Equivariance)), thereby promoting the disentanglement of structure and appearance. (b) illustrates a detailed view of the encoder (Section 4.3.2), which comprises an appearance encoder $\mathcal{E}_{FC}$, a segmentation network $\mathcal{E}_s$, and a Gaussian parameter estimation layer $\mathcal{E}_g$. $\mathcal{E}_{FC}$ calculates an appearance feature map $\mathbf{F}_a$ and direction map $\mathbf{V}$, which, along with segmentations $\mathbf{S}$ generated from $\mathcal{E}_s$, are used to compute latent Gaussians $\{\mathbf{g}\}$. In accordance with Property 1 (Discrete Objectness) and Property 2 (Spatial Compactness), entropy L_E and compactness L_C losses are applied to $\mathbf{S}$ to foster hard and localized segmentation masks in order to inject these desired properties into textons. Segmentations $\mathbf{S}$ are also employed in computing L_{Csis}, as detailed in Section 4.3.5.2.



(a) Input (b) Seg. overlay (c) Gaussians (d) Output 1 (e) Output 2 (f) Output 3

Figure 4.3: *Texture diversification.* Given the input (a), our network processes it to extract textons via segmentation (b). Each texton is represented by a Gaussian calculated from segmentations, as illustrated in (c). We can generate different versions (d) (e) (f) of the same texture by randomly reshuffling appearance features in latent Gaussians.

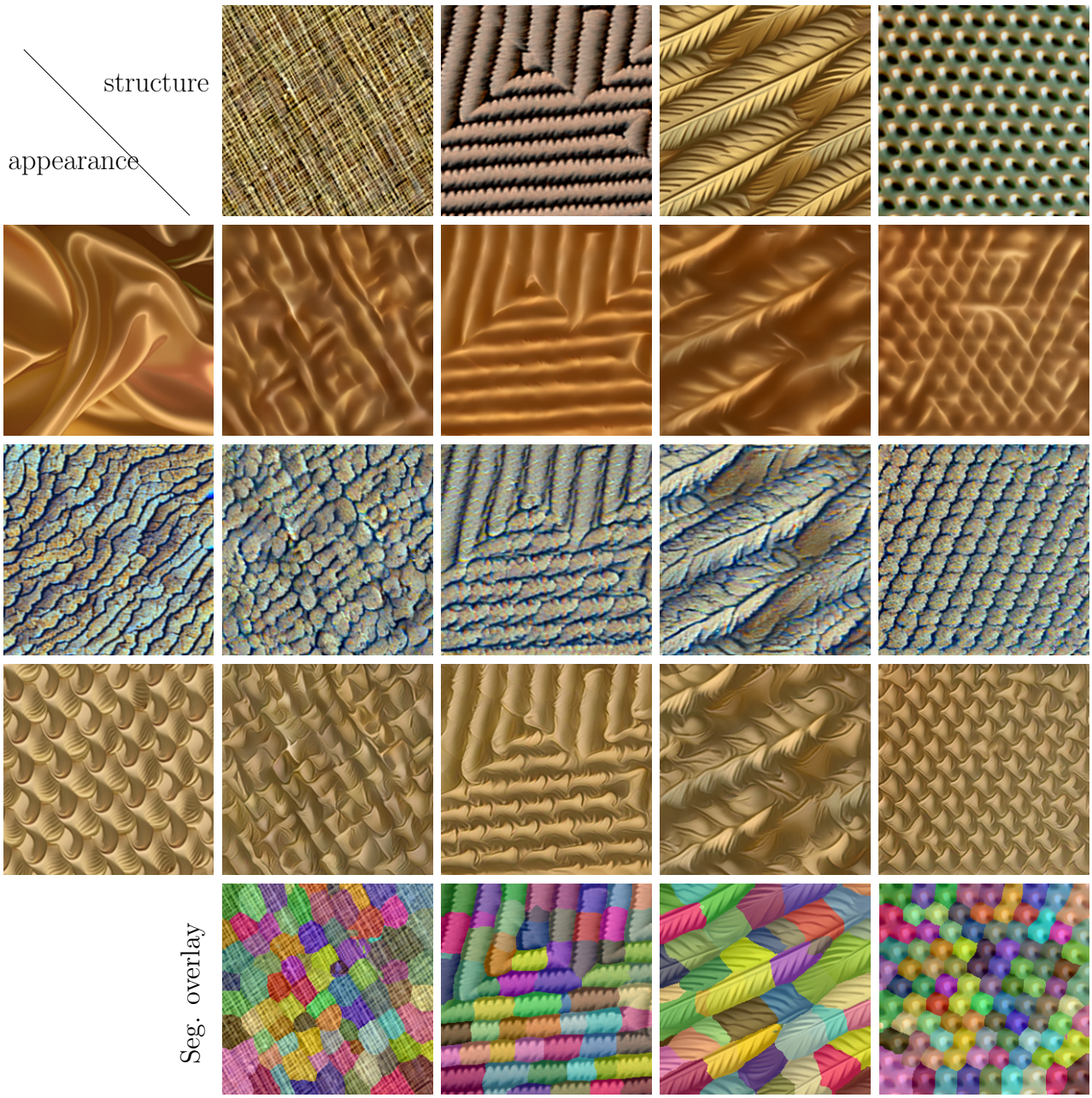


Figure 4.4: *Texture transfer.* This application involves appearance-providing (left column) and structure-providing (top row) textures. The corresponding transfer results are shown at the row-column intersections. We also visualize the segmentation masks **S** of the structure-providing inputs (bottom row).

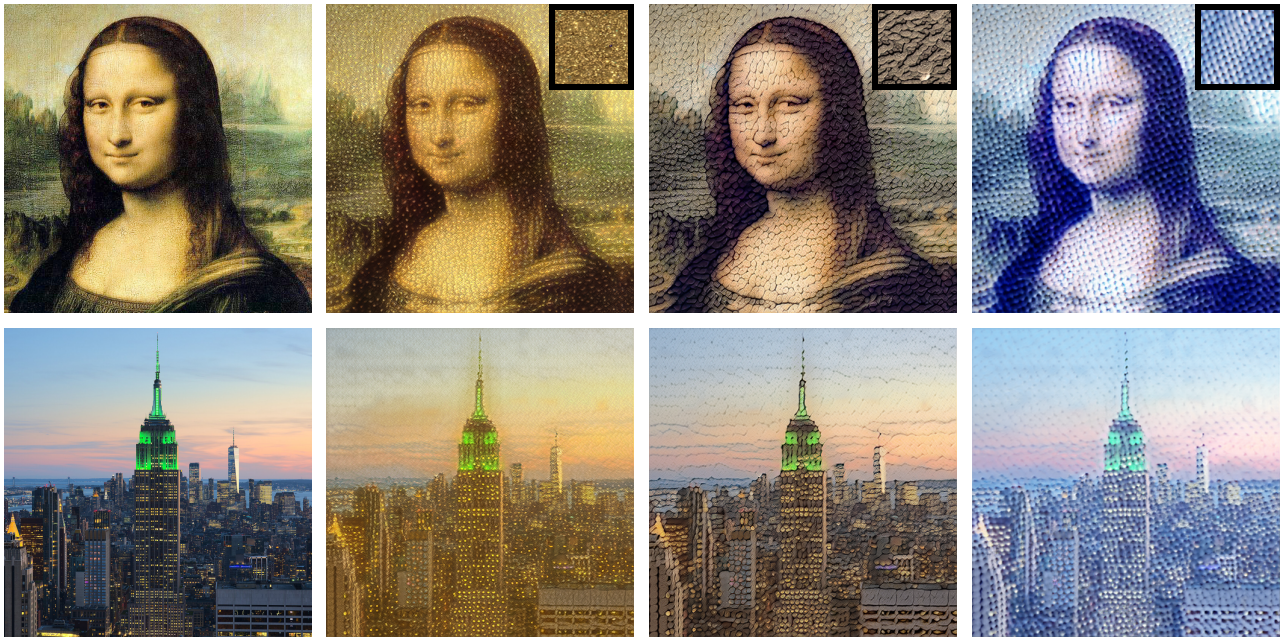


Figure 4.5: *Stylizing an image by tranferring textures.* Our approach is also adaptable to image stylization. In each row of the presented results, the original content is shown on the left, followed by the stylized outputs with the corresponding texture overlays in the top right corners.



Figure 4.6: *Spatial control in texture transfer.* We can apply textures to specific areas of an image as designated by image masks. From left to right: masks, textures, original images, stylized outputs.

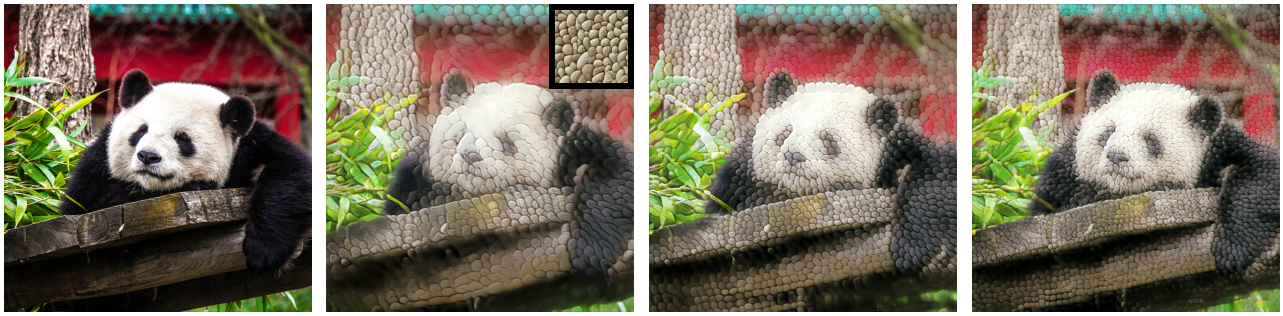


Figure 4.7: *Scale control in texture transfer.* Content image is shown in the first column. From second to last columns, the transferred results with large, median (2/3 of large scale) and small (2/3 of median scale) textons. The texture is shown on top right corner of the second column.

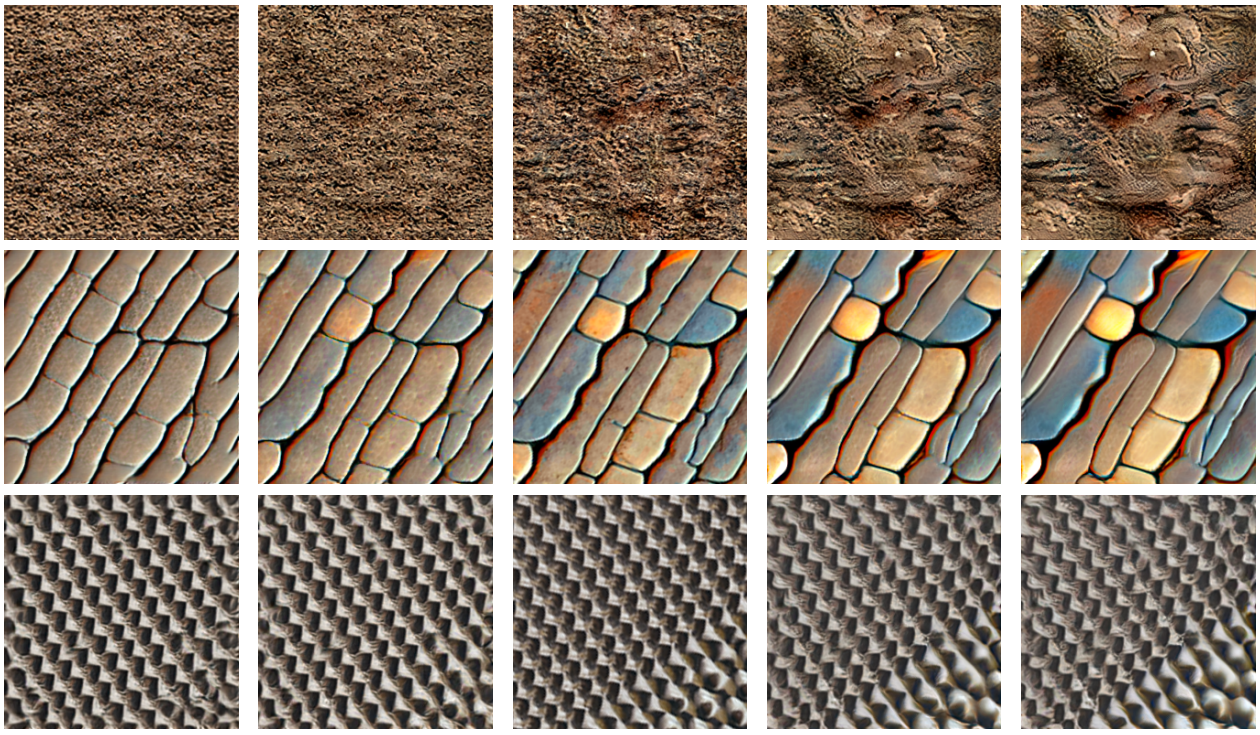


Figure 4.8: *Revealing and modifying appearance variations.* Our method is capable of either reducing or increasing variations in the appearance of a texture by correspondingly changing the appearance feature variations. With the input images shown in the center column, we demonstrate reduced variations towards the left and increased variations towards the right.

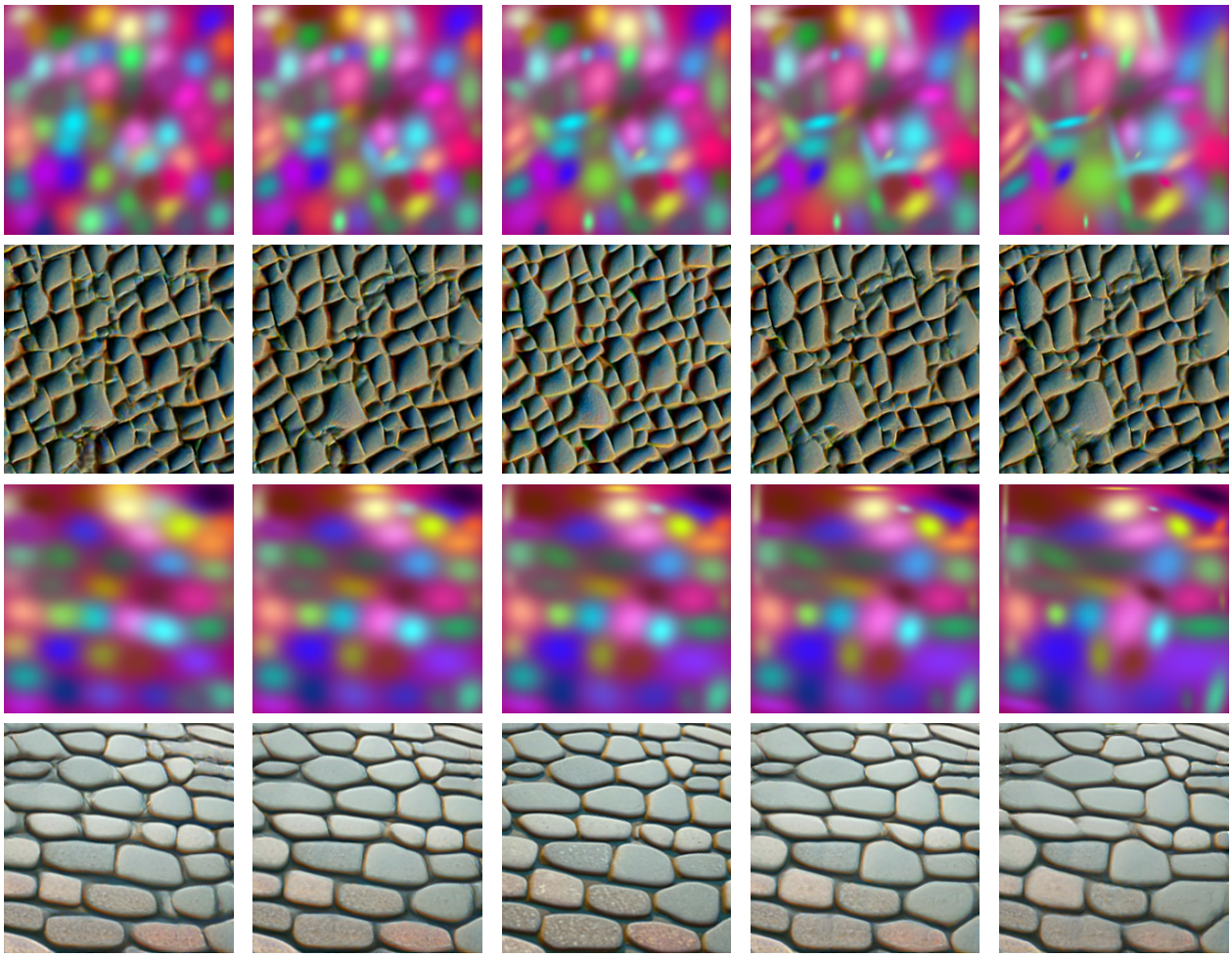


Figure 4.9: *Revealing and modifying geometric variations.* Our method is capable of either decreasing or increasing variations in the geometry of textons by changing the the variations in Gaussian covariances. With the input images and the corresponding visualized Gaussians in the center, we demonstrate reduced and increased variations towards the left and right.



(a) source (b) $\eta = 0.25$ (c) $\eta = 0.5$ (d) $\eta = 0.75$ (e) target (f) morphing

Figure 4.10: *Texture interpolation and morphing.* Given a source texture (a) and a target texture (e), we can generate interpolations at various degrees between the source and the target, controlled by interpolation coefficients of $\eta = 0.25$ (b), $\eta = 0.5$ (c), and $\eta = 0.75$ (d). Furthermore, the interpolation coefficients can vary spatially within one output, such as linearly increasing from left to right. This variation leads to texture morphing, as demonstrated in (f).

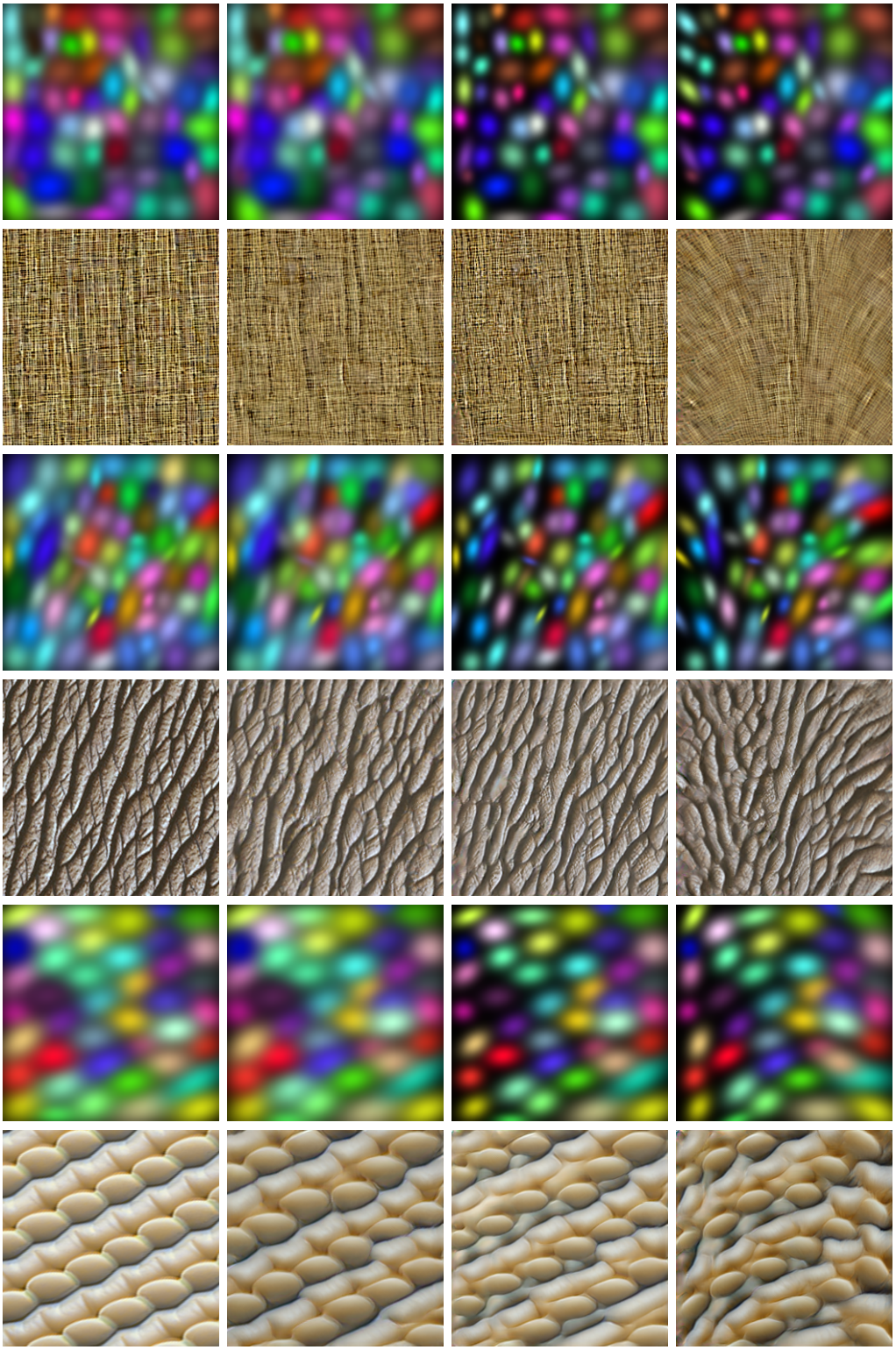
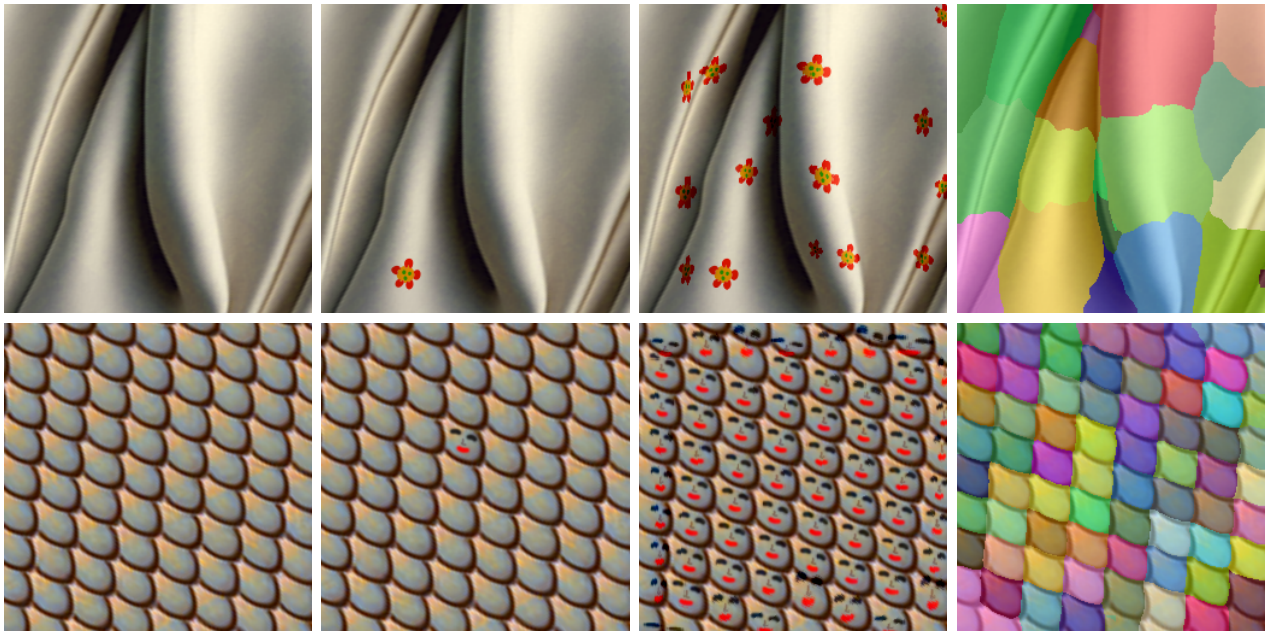


Figure 4.11: *Texton editing.* In order to edit individual textons within a texture, we can apply a sequence of edits (move, scale and rotate) to its latent Gaussians.



(a) Input (b) Edited (c) Propagated edits (d) Seg. overlay

Figure 4.12: *Edit propagation.* Given a texture (a), we can edit it (b) and the edits can be propagated to other similar locations (c) based on extracted textons (d), for both stationary (bottom) and non-stationary (top) structures.

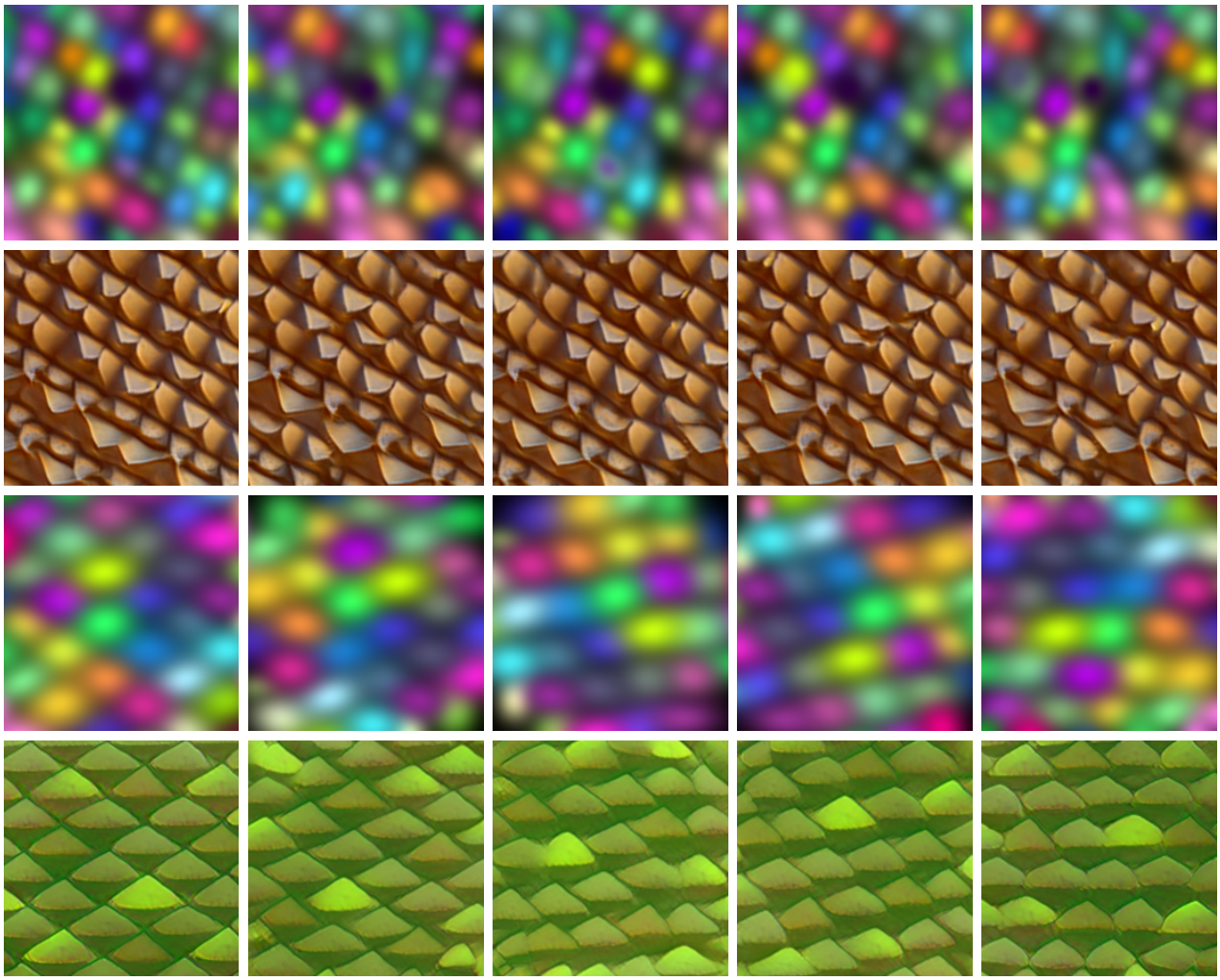


Figure 4.13: *Texture animation.* The top shows the animation created from a texture and perturbed shear flow. The bottom uses vortex flow. Please see the supplementary video for dynamic effects.



Figure 4.14: *Comparison to segment anything (SA).* SA is trained with supervision of 1 billion masks. However, SA is incapable of detecting and extracting textons from textures. Our model can extract textons from textures using fully unsupervised training.

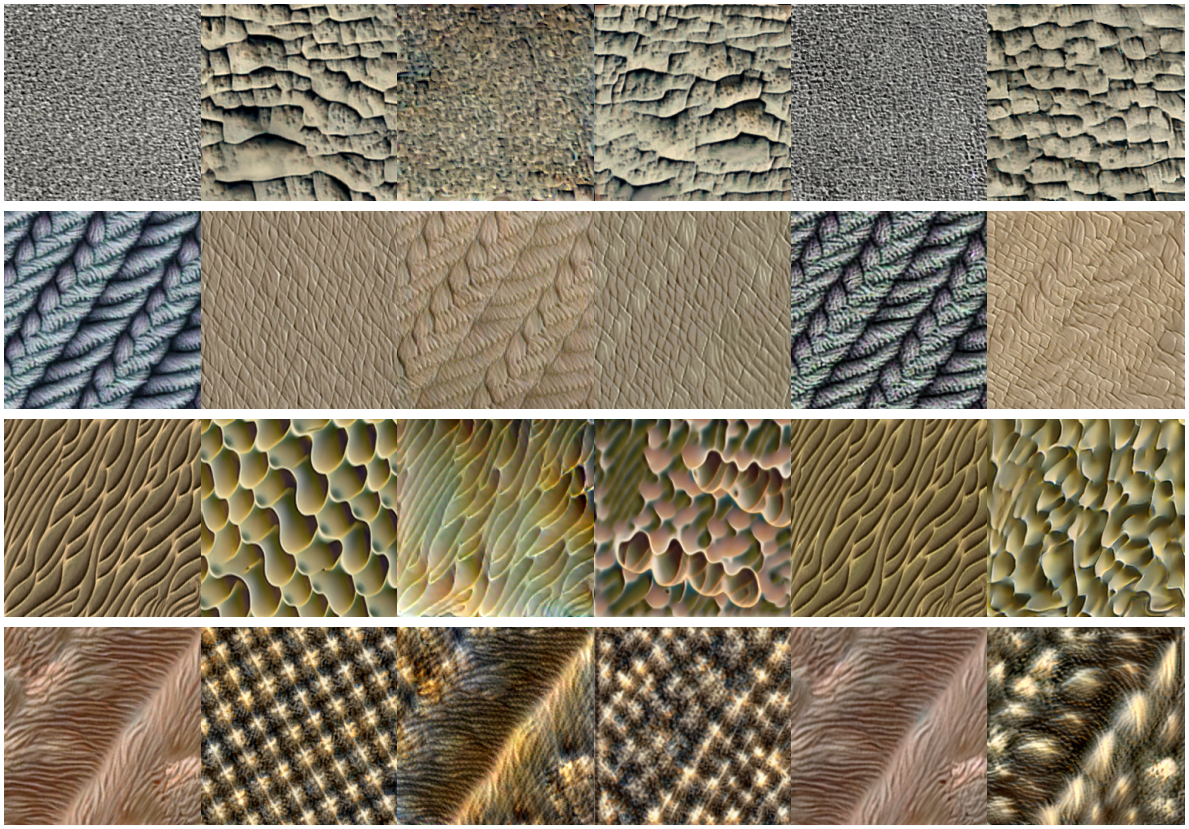


Figure 4.15: *Comparisons with existing approaches on texture transfer.* Our approach can preserve the better structure from the structure-providing texture than [148] and more accurate texton appearance than [147]. Cheng et al. [149] could not transfer the texton appearance to the structure image due to its use of text-to-image models while it might not be suitable to describe detailed textures using abstract human language.

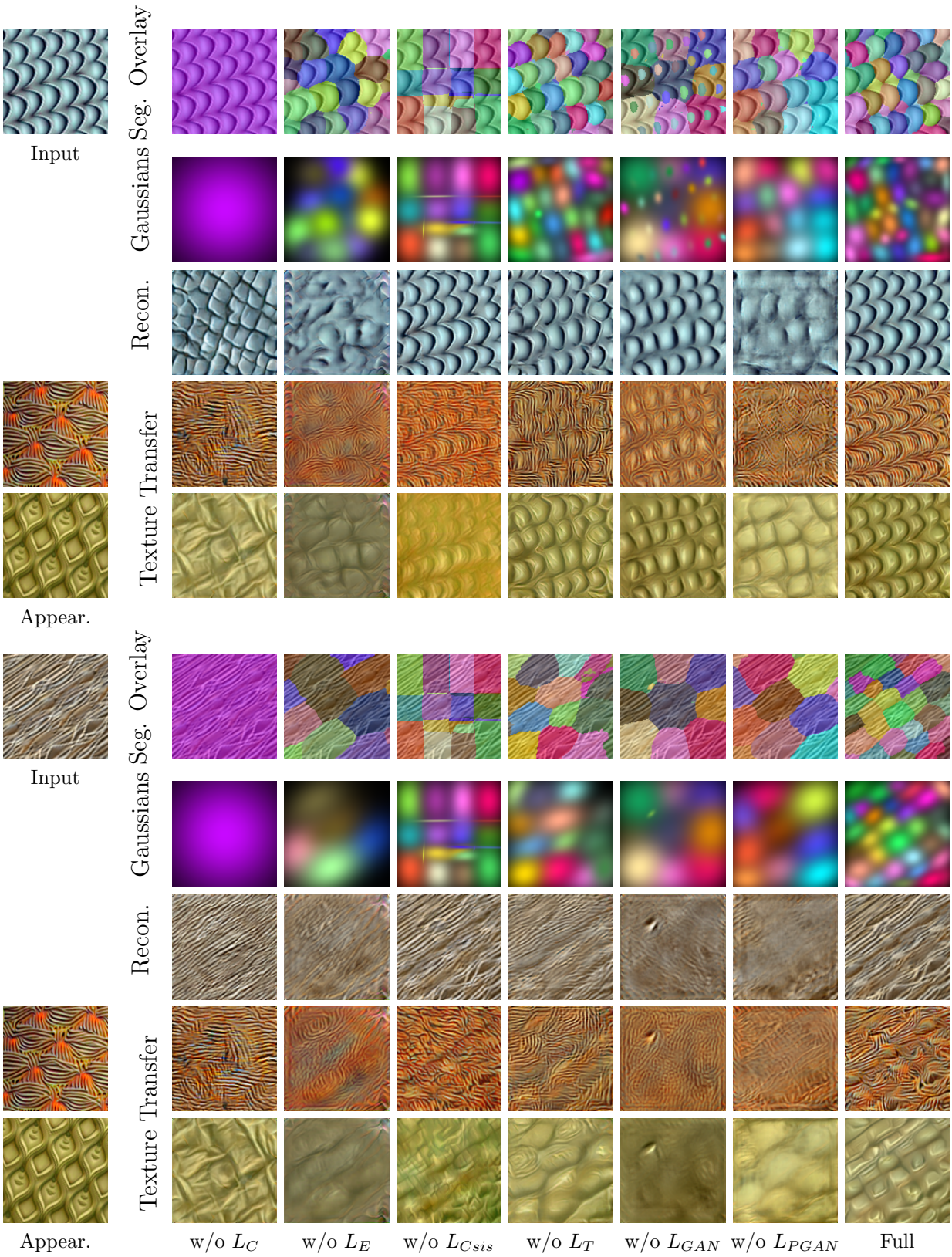


Figure 4.16: *Ablation studies on losses.* We ablate each loss from our implementation and visually compare their results with those using full losses. We visualize the segmentations $\mathbf{S}$, Gaussians $\mathbf{g}$, reconstructions I_r and texture transfer results given appearance-providing textures.

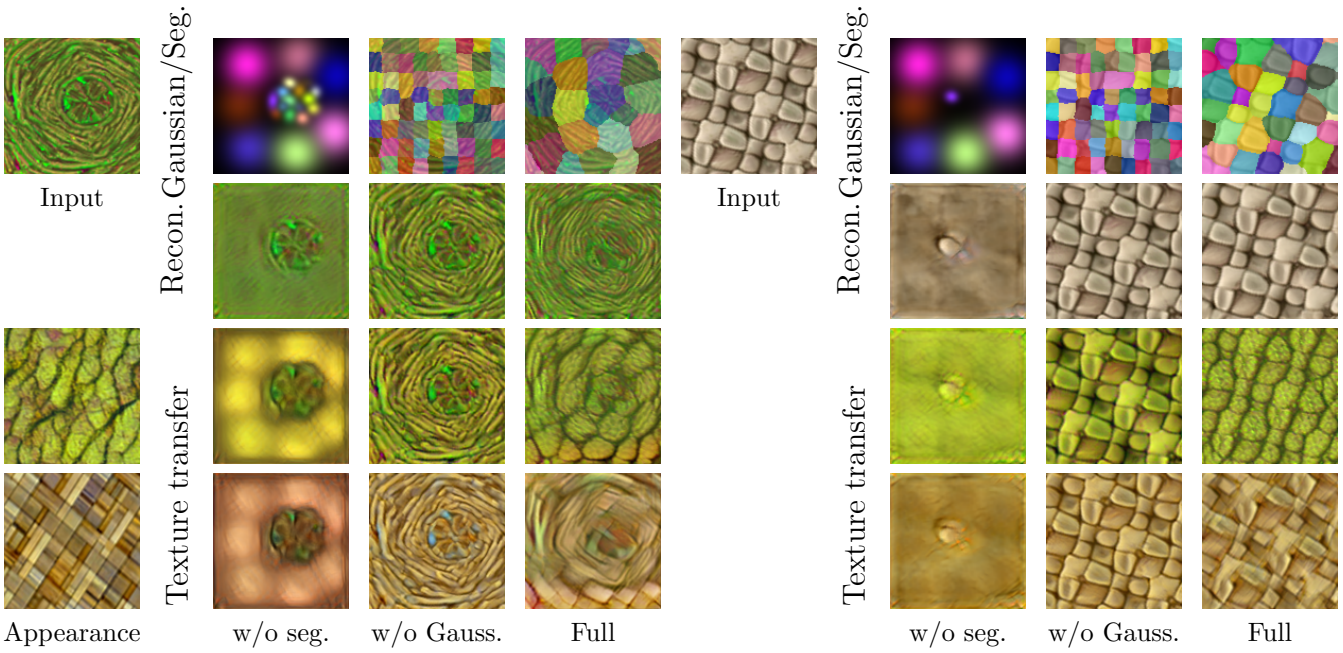


Figure 4.17: *Ablation studies on network components.* We visualize the Gaussians/segmentation masks, reconstruction and texture transfer results given appearance-providing textures.

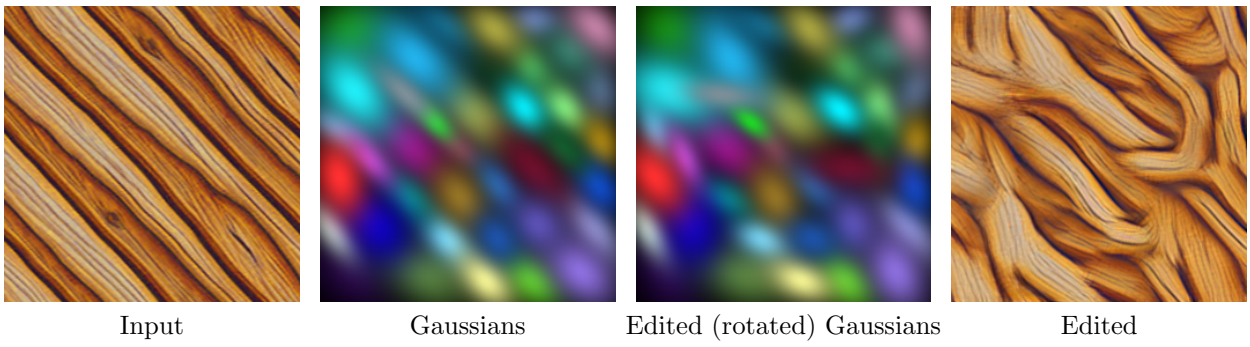


Figure 4.18: *Improper compositional representation for elongated structures.* For elongated, continuous structures, it might be improper to model them as localized textons. Editing individual textons might discrupt the continuous structures.

5: Appendix: Continuous Curve Textures

5.1 Existence Assignment

The algorithm for generating additional samples is shown in Algorithm 1.

The candidate samples are generated from pairs of $\mathbf{n}(s_i)$ and $\mathbf{n}(s_o)$ (lines 5-12 in Algorithm 1). In a pair of $\mathbf{n}(s_i)$ and $\mathbf{n}(s_o)$, if there is an unmatched input sample s_i' in $\mathbf{n}(s_i)$ (line 7), it will indicate the potential lack of an output sample, whose global position is $\mathbf{p}(s_o^c) = \mathbf{p}(s_i') - \mathbf{p}(s_i) + \mathbf{p}(s_o)$ located within $\mathbf{n}(s_o)$, and attributes are the same to s_i' (line 8). The algorithm loops over all pairs of neighborhoods, each neighborhood pair may or may not produce new candidate samples. All these samples s_o^c form a candidate sample set $\{s_o^c\}$. We group $\{s_o^c\}$ into clusters $\{\mathcal{S}\}$ (line 13-24) by assigning a sample to its nearest cluster $\mathcal{S}$ with distance between s_o^c and the center of $\mathcal{S}$ smaller than 0.5δ or otherwise create a new cluster $\mathcal{S}'$ using the sample. For each cluster $\mathcal{S} \in \{\mathcal{S}\}$ (line 25-31), we merge all its candidate samples $\{s_o^c\} \in \mathcal{S}$ as one output sample s_o by averaging their position and attributes using the same way as in the assignment step (Section 2.4.3.4). The existence of s_o is assigned as the ratio of the number of candidates within $\mathcal{S}$ over the number of overlapping $\mathbf{n}$ over the position of s_o. For the sake of explanation, assume $i(s_o) = 1$, it means all $\mathbf{n}$ overlapping over s_o produce one candidate sample on average, which suggests there could be missing samples, around s_o. Finally, s_o with $i > 0.5$ is added into the output

function `GenerateNewOutputSamples`($\{\mathbf{n}(s_i), \mathbf{n}(s_o)\}$)
$\{s_o^c\} \leftarrow \emptyset$ {candidate sample set}
$\{\mathcal{S}\} \leftarrow \emptyset$ {$\mathcal{S}$ is a cluster that contains some candidate samples}
$\{s_o\} \leftarrow \emptyset$ {new output sample set}
for $\mathbf{n}(s_i), \mathbf{n}(s_o) \in \{\mathbf{n}(s_i), \mathbf{n}(s_o)\}$ **do**
  for $s_i' \in \mathbf{n}(s_i)$ **do**
    if s_i' is unmatched **then**
      Generate a candidate sample s_o^c with $\mathbf{p}(s_o^c) = \mathbf{p}(s_i') - \mathbf{p}(s_i) + \mathbf{p}(s_o)$ and other attributes are the same to s_i'
      $\{s_o^c\} \leftarrow \{s_o^c\} \cup s_o^c$
    end if
  end for
end for
for $s_o^c \in \{s_o^c\}$ **do**
  {Generate clusters from the candidate sample set by greedily looping over all candidates; more advanced clustering technique can be applied to replace this step}
  Find the $\mathcal{S}$ within $\{\mathcal{S}\}$ with nearest center to s_o^c
  if `Distance`$(\mathcal{S}, s_o^c)$ ¡ 0.5δ **then**
    {`Distance`$(\mathcal{S}, s_o^c)$ computes the spatial distance between the center of $\mathcal{S}$ and s_o^c}
    $\mathcal{S} \leftarrow \mathcal{S} \cup s_o^c$
  else
    $\mathcal{S}' \leftarrow \emptyset$
    $\mathcal{S}' \leftarrow \mathcal{S}' \cup s_o^c$
    $\{\mathcal{S}\} \leftarrow \{\mathcal{S}\} \cup \mathcal{S}'$
  end if
end for
for $\mathcal{S} \in \{\mathcal{S}\}$ **do**
  create a new s_o with averaged sample positions and attributes by merging all $\{s_o^c\}$ within $\mathcal{S}$
  $i(s_o) \leftarrow \frac{\#\mathcal{S}}{\#\text{overlapping } \mathbf{n} \text{ over } s_o}$ {existence assignment}
  if $i(s_o) > 0.5$ **then**
    $\{s_o\} \leftarrow s_o$
  end if
end for
return $\{s_o\}$

Algorithm 1: *Generating new output samples in existence assignment.*

sample distribution every iteration.

5.2 Parameters

Table 5.1: *Parameters.* From left to right: neighborhood radii and sampling distances from lower to higher hierarchies. The parameters in the bottom part of the table share default values. Input size is bounding box size of input exemplar.

	r	δ	input size
Figure 2.12h	$\{50, 40, 30\}$	$\{20, 15, 10\}$	350×200
Figure 2.12p	$\{60, 50\}$	$\{40, 30\}$	300×300
Figure 2.1e	$\{40, 30, 20\}$	$\{20, 15, 10\}$	250×250
Figure 2.13j	$\{60, 50, 40\}$	$\{40, 30, 30\}$	250×250
Figure 2.12f	$\{50, 40, 30\}$	$\{30, 20, 10\}$	300×350
Figure 2.1h			400×400
Figure 2.16e			400×300
Figure 2.12t	⋮	⋮	500×400
Figure 2.15b			350×300
Figure 2.12d			350×200
Figure 2.12x			300×300
Figure 2.12j	⋮	⋮	200×300
Figure 2.15f			300×300
Figure 2.12n	$\{60, 50, 40\}$	$\{40, 30, 25\}$	200×300
Figure 2.12r			400×400
Figure 2.14h			350×400
Figure 2.16j			400×250
Figure 2.1a			300×250
Figure 2.1d	⋮	⋮	350×300
Figure 2.14c			350×250
Figure 2.17c			500×300
Figure 2.13n			400×200
Figure 2.17l			250×350
Figure 2.12v	⋮	⋮	250×400
Figure 2.12b			300×300
Figure 2.13f			300×250

Table 5.1 lists the parameters for the results shown in Chapter 2.

5.3 Performance

Our current implementation in C++ is unoptimized. It takes about 160 seconds to synthesize a pattern with about 750, 1000, 1300 output samples and 30, 30, 20 samples on average within neighborhoods at each hierarchy, on a desktop with AMD Ryzen 9 3950 X 3.49 GHz 16-core processor and 32 GB RAM. The major computational burden is on the neighborhood searching process (Section 2.4.3.3). The computational complexity of neighborhood searching mainly depends on the number of samples and neighborhood radius. To compute the similarity and sample matching between a pair of neighborhoods, the Hungarian algorithm [42] has complexity $O(N_{\mathbf{n}}^3)$ where $N_{\mathbf{n}}$ is the number of samples within a neighborhoood (assume all input and output neighborhoods have same number of samples). The patch match algorithm [47] is composed of two alternating steps: propagation and random search. In an optimization step, there are the $an_{\mathcal{O}}I_{max}$ neighborhood matching computation where a is a constant (in our implementation $a \approx 10$) which is related to the number of neighboring samples to a sample used in the propagation step and the range of random search, $n_{\mathcal{O}}$ is the total number of output samples, and I_{max} is the maximum number of patch match iteration. The complexity of an optimization step is thus $O(n_{\mathcal{O}}N_{\mathbf{n}}^3)$. Our algorithm has no more than 3 hierarchies and each hierarchy need about 7 steps. The lower hierarchy has less samples but a larger neighborhood radius (Section 2.4.3.5).

5.4 Texture Synthesis

Figure 5.1 shows the texture synthesis results by graph cut [152] (using the implementation in the GIMP Texturize Plugin [150]) and multi-resolution patch-match (using the implementation in [151] with guidance channels [51]). As shown, pixel-based texture synthesis might not preserve continuous structures as well as our vector-based method. These methods also need additional vector-pixel conversions and need to process all pixels instead of just samples around patterns. If standard texture synthesis methods are applied for vector patterns, the rasterization, synthesis, and vectorization process can introduce extra quality degradation and computation overhead, and thus might not be practical for interactive authoring as we present in the supplementary video.

6: Appendix: Clustered Vector Textures

6.1 Robustness analysis

6.1.1 Initial conditions and optimization

Figures 6.1 to 6.5 analyze the robustness of our method with different initial conditions and visualize the optimization process. Side-by-side comparisons of pattern optimization process using patch-based and random sample initialization are shown in the supplementary video.

Table 6.1: *Ratios between numbers of clustering operations and samples/clusters in each hierarchy.* For sample switch, the ratio is computed between the numbers of sample switches and the numbers of samples. For cluster merge/split, the ratios are computed between the numbers of cluster merges/splits and the numbers of clusters. Since a cluster usually contain many samples, there are many more sample switches than cluster splits/merges.

		Ratios		
Exemplar	Levels	Switch	Merge	Split
	1	0.1420	0.1047	0.0793
	2	0.1197	0.0526	0.0788
	3	0.0756	0.0323	0.0532
	1	0.1030	0.0782	0.0556
	2	0.0664	0.0384	0.0549
	3	0.0417	0.0319	0.0486
	1	0.1048	0.0994	0.0525
	2	0.0939	0.0358	0.0612
	3	0.0383	0.0095	0.0215

6.1.2 Hyperparameters

We also show the influence of hyperparameters $w_{\mathbf{f}}$ (in Equation (3.3)) and w_s (in Equation (3.9)) on the synthesis results. It is shown that our method is robust to moderate parameter variations.

6.1.3 Clustering operators

We study the effects of operators in the clustering step (Section 3.6.5.2). Figure 6.8 shows the effects of using different combinations of clustering operators. Table 6.1 shows the ratios between the numbers of sample switch operations and samples as well as the ratios between the numbers of cluster merge and split operations and clusters.

6.1.4 More Comparisons with prior art

We show more comparisons with prior methods in Figure 6.9.

6.1.5 A large synthesis result

We enlarge the exemplar (Figure 3.11j) 4×4 times, and the result is shown in Figure 6.10.

6.2 Implementation details

6.2.1 Representation

To compute the shape context feature vector $\mathbf{f}$, we use default hyperparameters (5 bins for logarithmic distances and 12 bins for angles) provided in [83].

6.2.2 Assignment step

Position assignment The sample position is computed using least squares [1, 2]

$$\underset{\{\mathbf{p}(s_o)\}}{\arg\min} \sum_{\substack{s_o \in \mathcal{O} \\ s_i = \mu_s(s_o)}} \sum_{\substack{m_s : \mathbf{n}(s_o) \to \mathbf{n}(s_i) \\ s'_o \in \mathbf{n}(s_o) \\ s'_i = m_s(s'_o)}} \left\| \hat{\mathbf{p}}(s_o, s'_o) - (\mathbf{p}(s_i) - \mathbf{p}(s'_i)) \right\|^2 \tag{6.1}$$

Sample existence assignment We follow [2] on sample existence assignment, which adaptively adjust the number of total samples by adding or removing samples based on confidence of existence ξ. Since the element filtering in Section 3.6.6 plays similar role to sample removal in getting rid of redundant elements (and thus redundant samples), which

is complemented with sample addition, we only add samples during existence assignment.

6.2.3 Performance

Our implementation has not yet been optimized for interactive applications. Our algorithm is run on a workstation with AMD Ryzen 9 3950X 16-Core 3.49 GHz CPU and 32 GB RAM. The algorithm is run on a single thread, except that the patch match algorithm [47] in the search step is parallelized using four threads on the CPU. We show the running time for each algorithm component in Table 6.2. The major computational bottleneck is in the search step. Our method can be accelerated by further parallelizing the nearest neighborhood search, assignment step with voting scheme and the queue updates of clustering operators with hardware acceleration; faster linear assignment problems (LAP) for the computation of neighborhood metric and shape energy; and sparser sampling of patterns while maintaining representative quality.

7: Appendix: Compositional Neural Textures

7.1 Dataset

We have partitioned the dataset into training and testing subsets through a random split, aiming to maintain a consistent data distribution across both training and testing scenarios. Figure 7.1 shows 100 images, randomly sampling from the training subset,

without processing with data augmentation (i.e. various shape and color transforms of the originals), to provide a glimpse into the dataset's diversity. This dataset spans an extensive range of textures, varying from stochastic to regular patterns.

7.2 Evaluation details

The metrics in Table 4.1 are calculated across 5400 images or image pairs. For texture diversification (TD), the Fréchet Inception Distance (FID) is computed between pairs of input image and one of its diversified version. To calculate the diversity (DVS) metric, we encode input textures as latent Gaussians. Then we create 4 unique variations by reshuffling appearance features. These variations are then processed through a VGG-19 network. The DVS is calculated as the average differences in the network's responses from its 2nd, 4th, 8th, and 12th layers. In texture transfer (TT), we use 5400 pairs of images. One image in each pair provides the appearance, and the other provides the structure. The metrics are derived by comparing the transferred results with the appearance- and structure-providing inputs. For texton editing, we encode images into latent Gaussians and introduce random geometric edits. These edits include random movements (with normal-distributed displacements having a mean of 0 and a standard deviation of 0.1 $\times$ the average distance to the nearest other Gaussians), scaling (by a factor ranging from 0.8 to 1.25), and rotating (within a range of -30 to 30 degrees). We use these edits and their reconstructions to compute the CC and FID metrics.

7.3 Texton segmentation

Our methodology centers on the ability to segment individual textons from a wide variety of textures, which facilitates a number of applications. Figures 7.2 and 7.3 show 120 uncurated segmentation results, demonstrating the effectiveness of our approach across diverse textures.

7.4 Texture transfer

By default, our texture transfer results are generated by "mean alignment" mechanism by "adjusting the mean of Gaussian appearance features of the structure-providing texture to match those of the appearance-providing texture" (Section 4.4.2). However, our framework also supports other mechanisms for texture transfer. For example, one can replace each appearance feature of the structure-providing texture with a randomly sampled feature from the appearance-providing texture. Figure 7.4 presents a comparison of the results generated using both these mechanisms. As expected, the replacement method tends to preserve less appearance information from the structure-providing texture compared to the mean alignment method. However, both methods are effective. In addition, Figure 7.5 shows more results about scale control in transferring texture to images.

7.5 Comparisons

In Figure 7.6, we present a broader range of uncurated comparisons utilizing the segment anything method. Further, we show more comparisons on texture transfer with

existing approaches. This involves randomly sampling a set of images and merging them in such a way that one provides the structure while the other provides the appearance. Some notable examples of these pairings are shown in Figures 7.7 and 7.8. Additionally, we include uncurated combinations of these images in Figures 7.9 to 7.14.

7.6 Ablation studies

Figures 7.15 and 7.16 shows more results for ablation studies of losses. Figure 7.17 shows more results for ablation studies of neural architectures.

7.7 Implementation details

7.7.1 Network architectures

Encoder $\mathcal{E}$. We utilize Mask2Former model [138] as the segmentation network $\mathcal{E}_s$. This network employs ResNet-50 [153] as the backbone and set up with 100 object quries and other default parameters as specified in [138] except that the number of transformer decoder layers is reduced to 4 and the number of classes is set as 1 given that texton is the sole semantic class under consideration. The fully convolutional encoder $\mathcal{E}_{FC}$ processes an input image to generate an appearance feature map $\mathbf{F}_a$ and a direction map $\mathbf{V}$, as illustrated in Table 7.1. To generate $\mathbf{F}_a$, $\mathcal{E}_{FC}$ comprises four downsampling residual blocks followed by two upsampling convolutional blocks. To generate the direction map, the network diverges after the residual blocks, incorporating an additional set of two upsampling convolutional blocks. Reflection padding is employed in $\mathcal{E}_{FC}$ to prevent the leakage of absolute positional information. For an input image of size 128×128, the segmentation masks generated

by Mask2Former are of dimensions $32 \times 32 \times n$, where $n = 100$ represents the maximum number of Gaussians or the number of queries utilized in Mask2Former. The appearance feature map $\mathbf{F}_a$ has dimensions of $32 \times 32 \times d_a$ with $d_a = n_f = 382$. The direction map $\mathbf{V}$ is sized $32 \times 32 \times 2$. The segmentation masks, appearance feature map and the direction map are upscaled to match the original image size (128×128 or 256×256 during the fine-tuning phase) before feeding into the Gaussian parameter estimation layer $\mathcal{E}_g$.

Generator. We adopt the SPADE generator as described in [141], configured with four SPADE ResBlocks. Instead of using the learnable constant feature map originally employed in SPADE, we replace it with a downscaled version of $\mathbf{F}_{input} = \max_i \alpha_i$, which has dimensions that are 1/4 the size of the input image. When the input image resolution is $H \times W$, the intermediate feature maps produced by the sequence of SPADE ResBlock are sized as follows: the first block generates $H/4 \times W/4 \times 384$ (n_f+length of ν (2)= 384), the second block $H/2 \times H/2 \times 192$, the third block $H \times W \times 96$, and the final block outputs $H \times W \times 3$.

Discriminators. We have two discriminators $\mathcal{D}$ and $\mathcal{D}_p$, each serving a different purpose. The discriminator $\mathcal{D}$ is identical to StyleGAN2 [154] without minibatch discrimination [143]. We employs the patch discriminator $\mathcal{D}_p$ consisting of 4 layers from [155].

7.7.2 Training details

In each iteration, we sample a minibatch of size N, generating N reconstructed images, N transformed images, and N/2 images reconstructed from reshuffled features for saving

memory. For the GAN loss specifics, we conform to the StyleGAN2 setting [154], including the non-saturating GAN loss and lazy R1 regularization executed every 16 steps. for both the $\mathcal{D}$ and the $\mathcal{D}_p$. The R1 regularization weight is 10.0 for the image discriminator $\mathcal{D}$ and 0.1 for the patch discriminator $\mathcal{D}_p$.

Initially, the weights are set as follows: $w_{Csis} = 0.008$, $w_E = 0.08$, and $w_C = 100$. These weights are dynamically adjusted at specific training milestones. After 5,000 steps, w_{Csis} is increased to 0.08. Subsequently, at the 100000 step, w_E is adjusted to 0.1. Meanwhile, the value of w_C undergoes two changes: it is reduced to 1 after 1000 steps, and then further lowered to 0.2 after 5,000 steps. Regarding the texture loss, we employ the default parameters specified in [121], with the exception of the patch sizes. Initially, the patch size is set to 2, which is then increased to 4 after 40,000 steps and further to 6 after 80,000 steps. During the fine-tuning phase, all parameters are maintained at their final adjusted values from the initial training phase. We use ADAM [156] with $5e^{-5}$ learning rate, $\beta 1 = 0.5$ and $\beta 2 = 0.999$. During the initial training phase on 128×128 images, we use batch size 64 on 8 A100 (40GB) GPUs. For fine-tuning on 256×256 images, we use batch size 56 on 8 A100 (80GB) GPUs.

7.7.3 Texture animations

We create texture animations using two types of flow fields: perturbed shear flow and vortex flow. Given a point of initial location (x_{init}, y_{init}) $(x_{init}, y_{init} \in [-1, 1])$, we compute the subsequent position of this point (x_t, y_t) at time t induced by the perturbed shear flow

as

$$x_t = \begin{cases} x_{init} - vt + \epsilon_t & y_{init} < -1/3 \\ x_{init} + vt + \epsilon_t & y_{init} > 1/3 \\ 0 & otherswise \end{cases} \tag{7.1}$$

and $y_t = y_{init}$, where v is the velocity of point; ϵ_t is the perturbation at time t. To compute ϵ_t, we initially uniformally sample 10 frames; for each of these frames, we sample a value for ϵ from a normal distribution that has mean of 0 and a standard deviation of 1/30. After obtaining these samples, we perform interpolation of ϵ_t values in the time domain to establish a continuous temporal profile of the perturbation. If a point moves beyond the image boundary, it will re-enter the image from the opposite side. To define our used vortex flow, for simplifying notations, we convert the coordinates into polar coordinate systems as $r = \sqrt{x^2 + y^2}$ and $\phi = \mathsf{atan2}(y/x)$, then at time t,

$$\phi_t = \phi_{init} + \omega t \tag{7.2}$$

and $r_t = r_{init}$, where ω is the angular velocity of point.

7.8 Options for High-quality Productions

We showcase the flexibility and effectiveness of our compositional texture representation across various texture manipulation tasks. To integrate this technology into products, it's crucial to achieve outputs of high quality and resolution, retaining all details from the source images. A direct approach would be to upscale our model to train on

large-scale, high-resolution images. However, we propose that training and implementing a high-resolution, sizable, dedicated end-to-end network for compositional texture editing isn't the most efficient for production. This approach demands excessive demands on a single network for both control and quality. Instead, we discuss some alternative, potentially more efficient methodologies that can be applied in the post-processing stage.

When working with a high-resolution (HR) input image, users first need to downsample it to match our network's capacity. This step allows for the creation of a low-resolution (LR) edited image by modifying encoded latent Gaussians. The process of converting LR edited images back to high-resolution, using the original HR input images, is a classic example of a reference-based super-resolution (SR) problem [157]. Generally, for effective results, the reference images used should possess a texture and/or content structure similar to the LR image [158]. Recent advancements in general reference-based super-resolution (SR) methods, which are potentially instrumental for deploying our techniques into production, have been detailed in [159, 158, 160, 161]. Moreover, given that both our inputs and edited outputs are textures, an alternative strategy could involve performing optimization at test-time. This would be aimed at maximizing the similarity in texture between the HR input and the LR edited output. Such an approach could effectively bridge the gap between differing resolutions, ensuring a closer match in texture quality and details between the input and output.

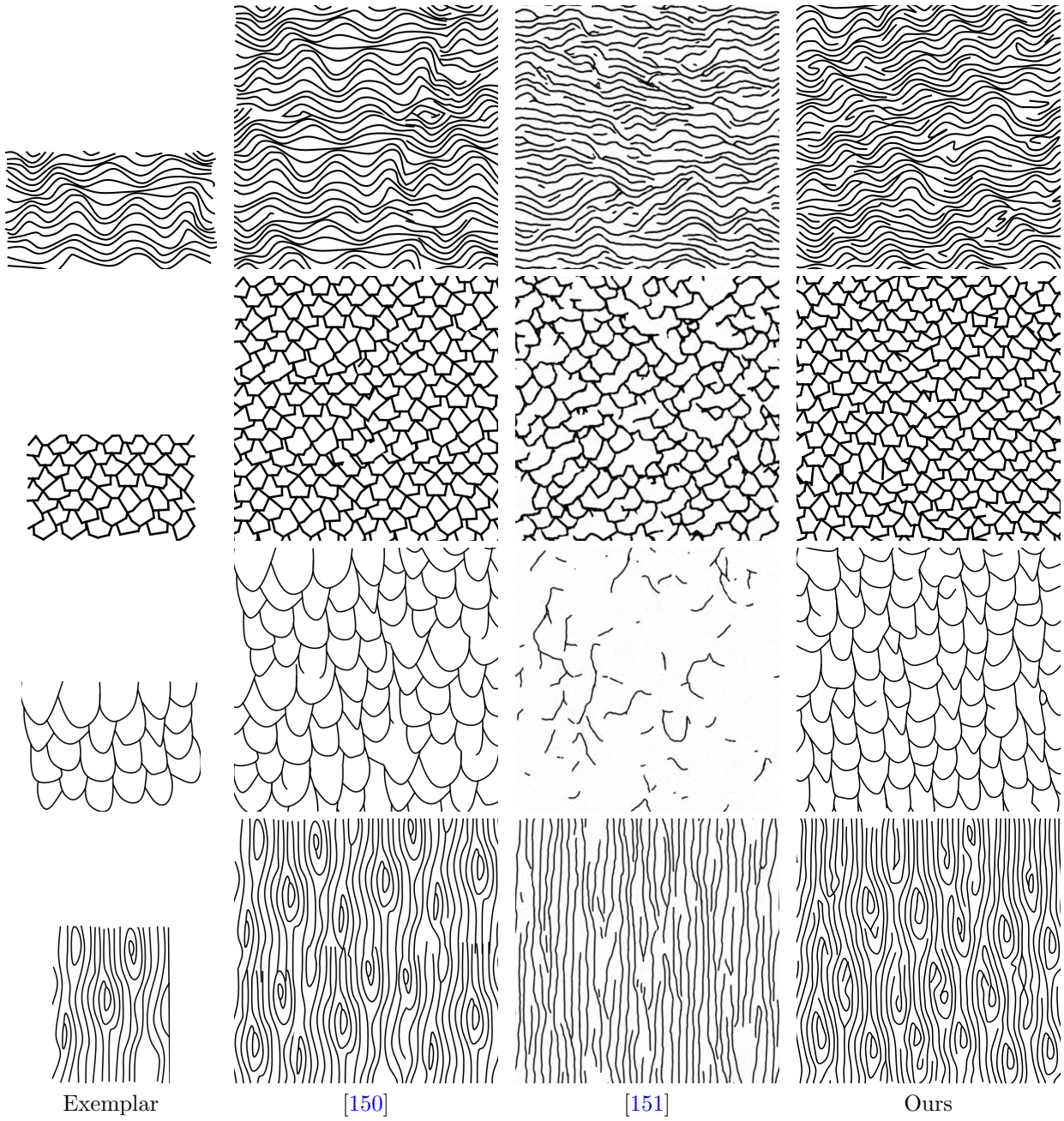


Figure 5.1: *Comparison of our algorithm to texture synthesis.* Texture synthesis methods need additional vector-pixel conversions and need to process all pixels instead of just samples around patterns.

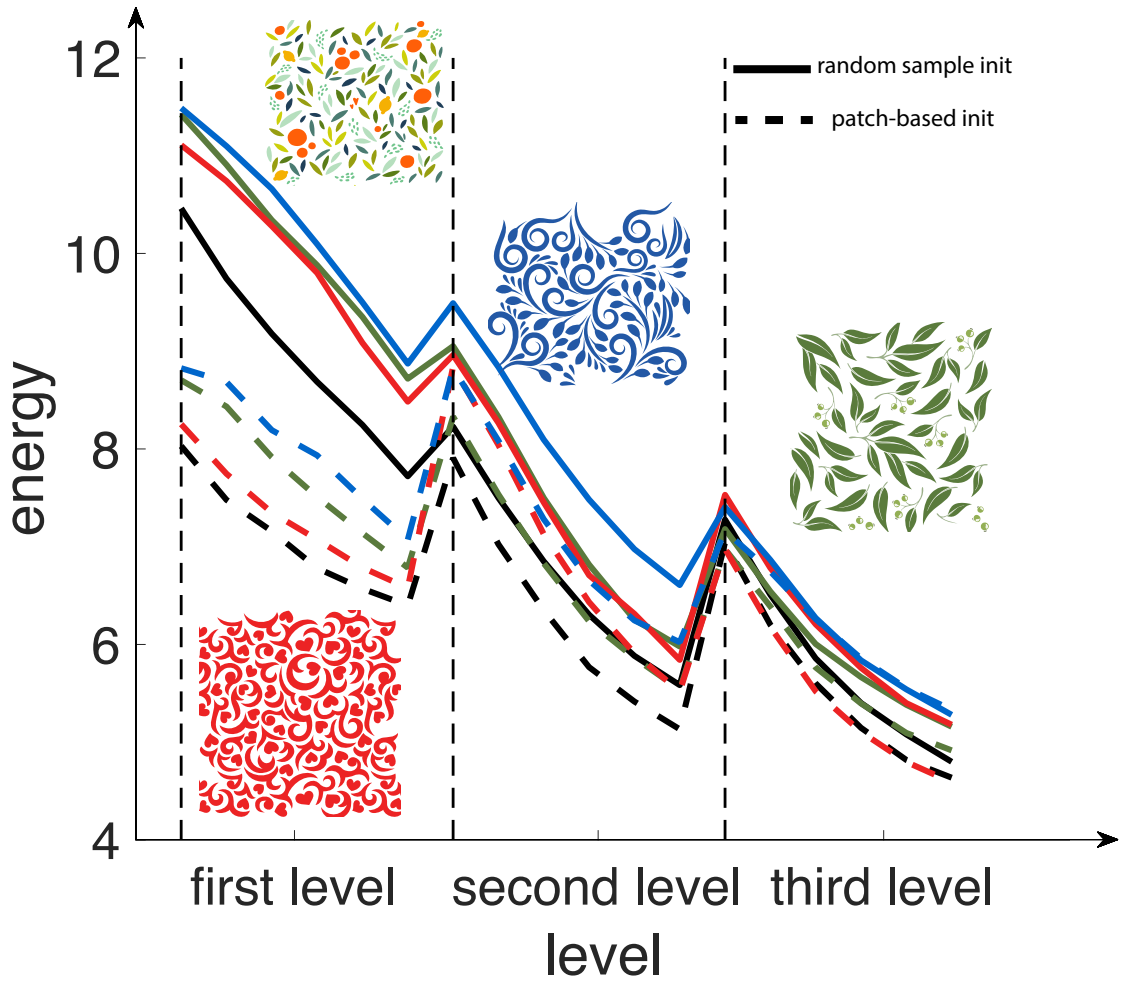


Figure 6.1: *Energy plots.* We visualize all energy plots in a single figure for direct visual comparison. Energies are normalized using the number of samples within an output neighborhood and the total number of matched neigborhood pairs. Solid and dashed curves represent plots produced with random sample and patch-based initializations, respectively. Corresponding exemplars and energy curves are shown in the same colors, except for the colorful pattern (shown on the top left) which corresponds to the black curves. Vertical dashed lines indicate the start of each optimization hierarchy.

Table 6.2: *Running time of each algorithm component at different hierarchy (unit: second).* Note there are seven search/assignment/clustering steps in a hierarchy, and what we show is the total time consumed by the seven steps. Since we adapt number of samples (# samples) during optimization, # samples is the average number of samples in a hierarchy of optimization.

					Assignment		Clustering				
Exemplar	Level	# Samples	Initialization	Search	Position	Attribute	Switch	Merge&Split	Recon	Total	
	1 2 3	1816 3032 4591	1.1 5.2 6.1	254.5 291.4 144.6	0.8 1.4 2.5	24.8 36.9 32.6	4.1 19.9 44.9	4.8 5.7 7.7	25.7 37.3 50.4	315.6 397.8 288.7	1002.1
	1 2 3	2845 4751 7370	2.4 10.3 12.3	1603.6 1829.4 913.6	2.1 4.4 9.5	89.6 134 189	30.2 98.5 237.6	17.9 25 44.3	53.8 77.7 128.5	1799.5 2179.2 1534.8	5513.5
	1 2 3	2388 4629 7626	3.1 10.5 13	814.4 1706.8 1482.7	1.4 3.8 9.7	57.2 145.2 280.5	12.0 101.1 227.2	6.1 16.9 30.3	22.3 37.2 55.1	916.5 2021.5 2098.5	5036.5
	1 2 3	1641 3057 4510	2.6 4.9 5.9	317.6 460.5 149.5	0.9 1.8 2.8	24.1 57.9 72	15.1 37.4 80.0	5.1 6.8 8.7	7.4 11.7 17.4	372.8 581 336.3	1290.1
	1 2 3	1200 2104 3357	0.6 2.6 3.0	61.7 84.3 87.1	0.4 0.7 1.6	7.2 11.1 17.1	2.7 6.1 12.5	1.3 2.3 3.6	3.4 4.5 6.6	77.2 111.7 131.6	320.5
	1 2 3	1309 2417 3888	0.7 2.7 3.3	75.3 134.5 174.7	0.5 0.9 1.9	9.4 18.6 33.6	3.1 9.9 18.7	2.3 3.5 4.6	2.4 3.6 4.8	93.6 173.8 241.5	508.8
	1 2 3	1900 3879 6965	1.3 4.6 6.1	350.2 849.3 1589	0.9 2.8 9.1	29.4 79.2 196.1	11.3 76.6 264.5	4.6 12.4 27.8	16.2 24.9 35.8	413.9 1049.8 2128.4	3592.1
	1 2 3	573 913 1505	4.5 19.7 22.1	5.8 6.5 22	0.1 0.3 0.7	1.1 3.2 12.2	4.5 26.2 45.3	0.4 0.3 0.4	1.6 2.5 2.5	18.1 58.7 105.2	182.0
	1 2 3	1707 2953 4688	4.7 6.8 7.4	269.1 361.9 80.5	1 1.7 2.6	22.9 47.6 62.1	11.4 29.1 72.6	3.7 7.7 11.8	12 16.5 27.7	324.9 471.4 264.6	1060.9
	1 2 3	2075 3311 5259	4.1 7.1 8.3	531.6 472.2 147.9	1.1 1.9 3.7	38 58.9 84.6	9.4 31.1 68.6	5.6 8.9 15.2	27.1 38.8 51.3	617 618.9 379.6	1615.5

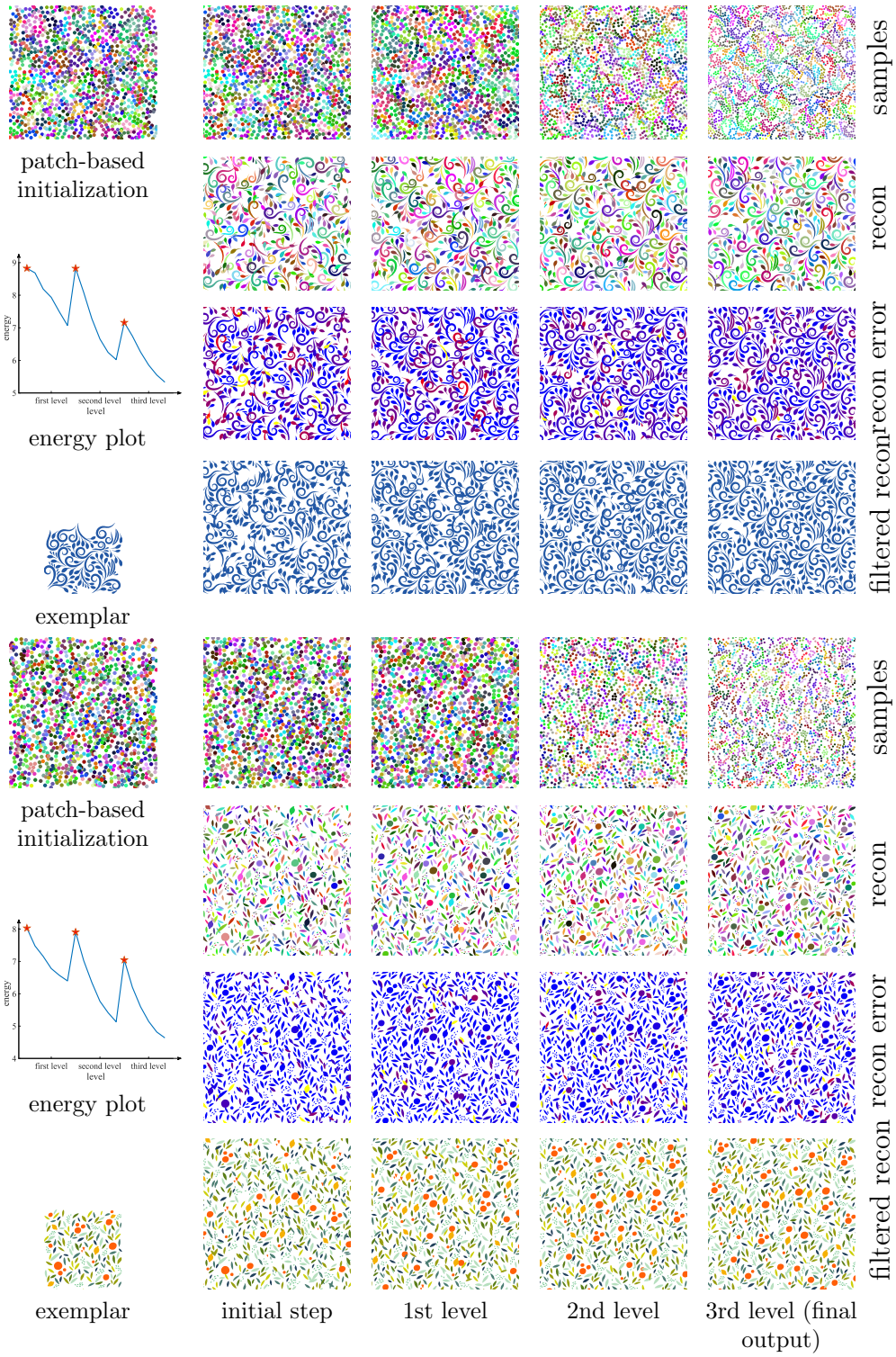


Figure 6.2: *Optimization process with patch-based initialization.* We show the intermediate samples (first/fifth rows, where the first column shows initialization), reconstructed patterns (second/sixth rows), reconstruction errors (third/seventh rows) and filtered reconstructions (forth/eighth rows) after the very first optimization step (initial step) and the last steps in each hierarchical levels. The energy plots are shown in the first column where red stars annotate the start of each optimization hierarchy. Corresponding clusters and elements are visualized in the same colors. In third/seventh rows, red and blue indicate high and low reconstruction errors; yellow indicates filtered-out elements with errors surpassing a threshold.

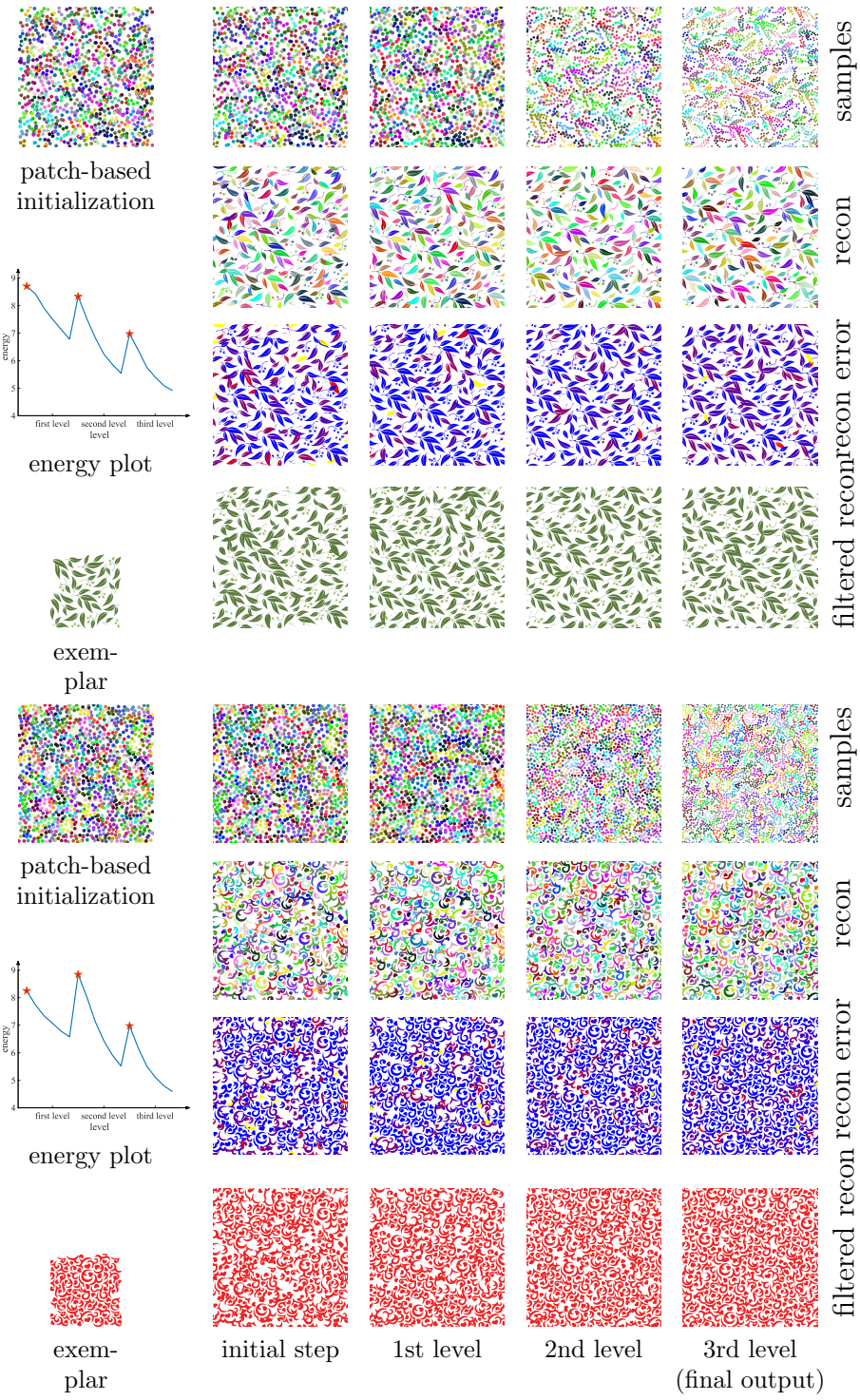


Figure 6.3: *Continuation from Figure 6.2.*

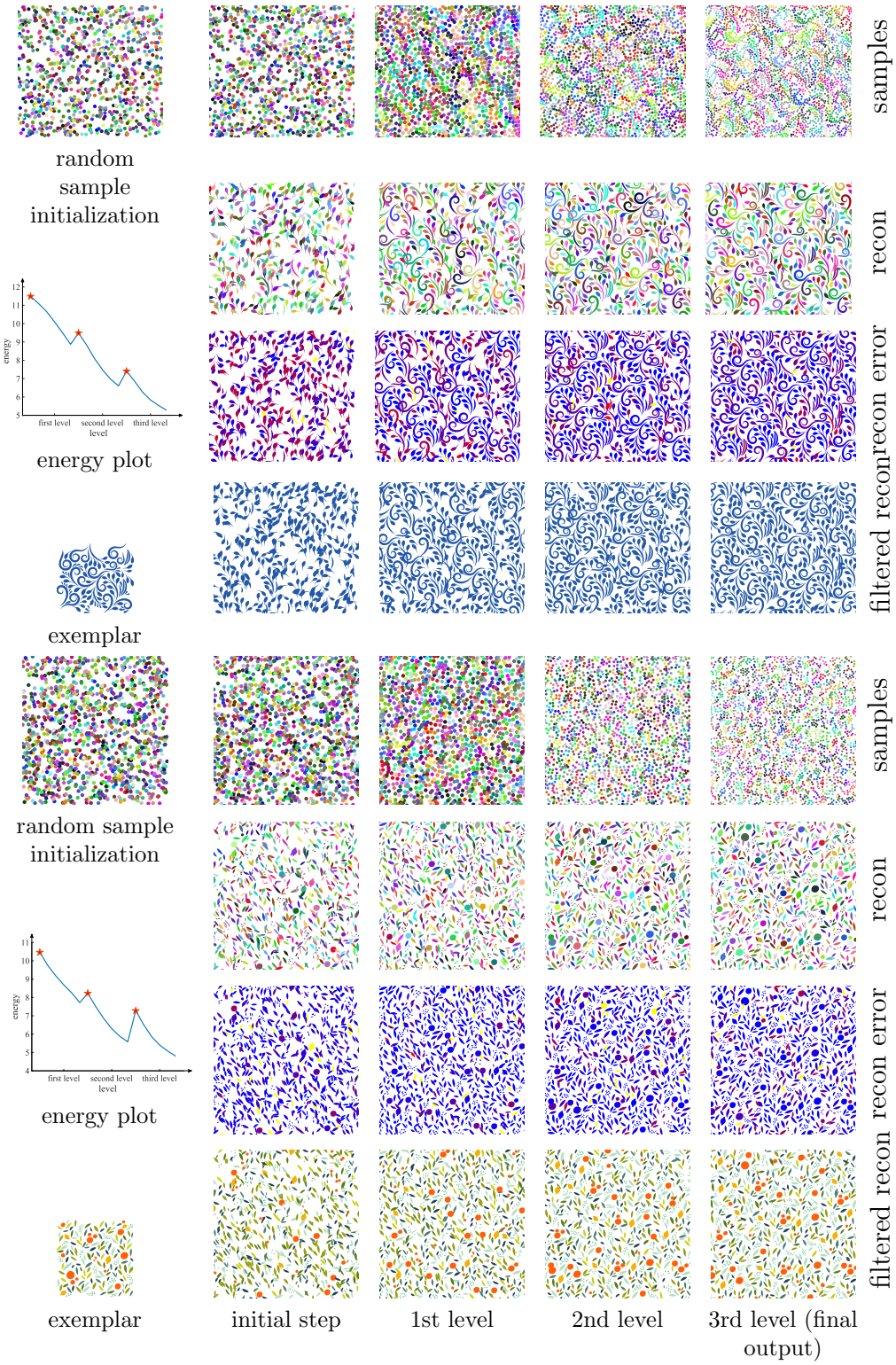


Figure 6.4: *Optimization process with random sample initialization.* We show the intermediate samples (first/fifth rows, where the first column shows initialization), reconstructed patterns (second/sixth rows), reconstruction errors (third/seventh rows) and filtered reconstructions (forth/eighth rows) after the very first optimization step (initial step) and the last steps in each hierarchical levels. The energy plots are shown in the first column where red stars annotate the start of each optimization hierarchy. Corresponding clusters and elements are visualized in the same colors. In third/seventh rows, red and blue indicate high and low reconstruction errors; yellow indicates filtered-out elements with errors surpassing a threshold.

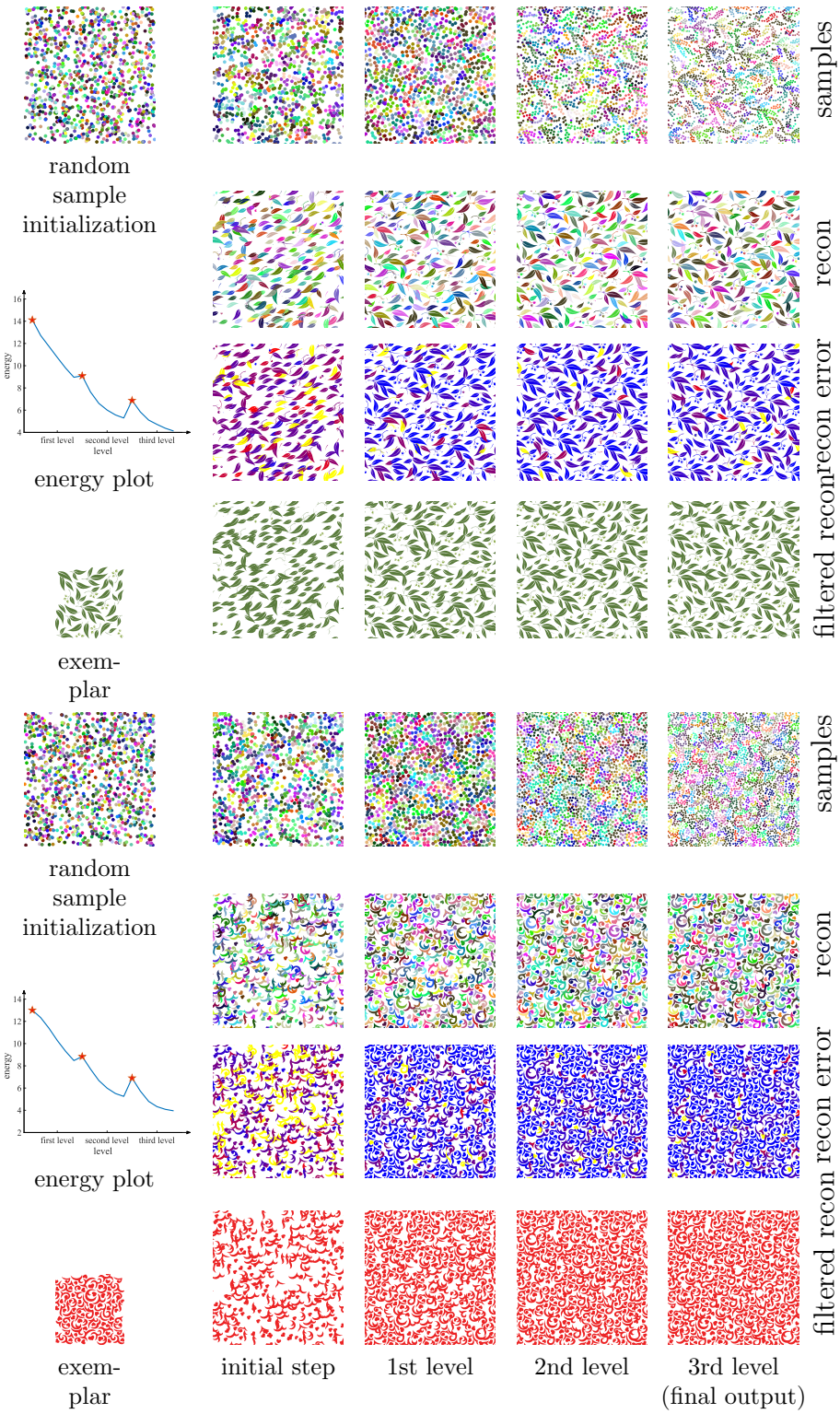


Figure 6.5: *Continuation from Figure 6.4.*

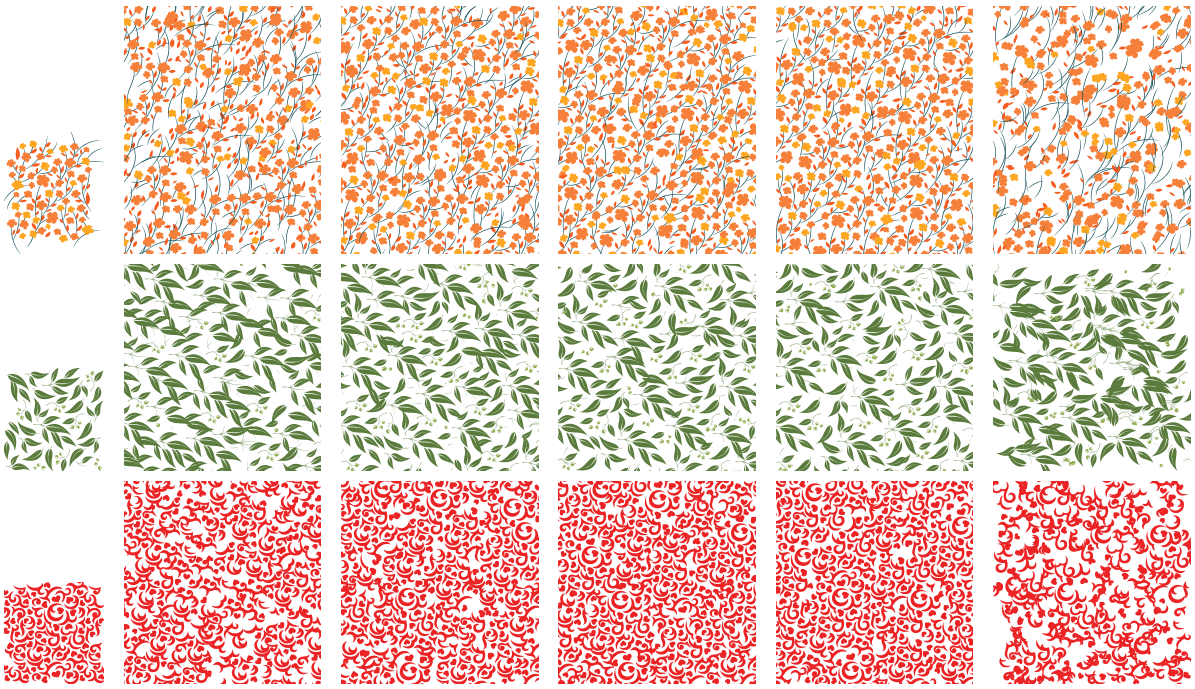


Figure 6.6: *Synthesis results using different hyperparameters* $w_{\mathbf{f}}$. $w_{\mathbf{f}} = 10$ is the chosen parameter.

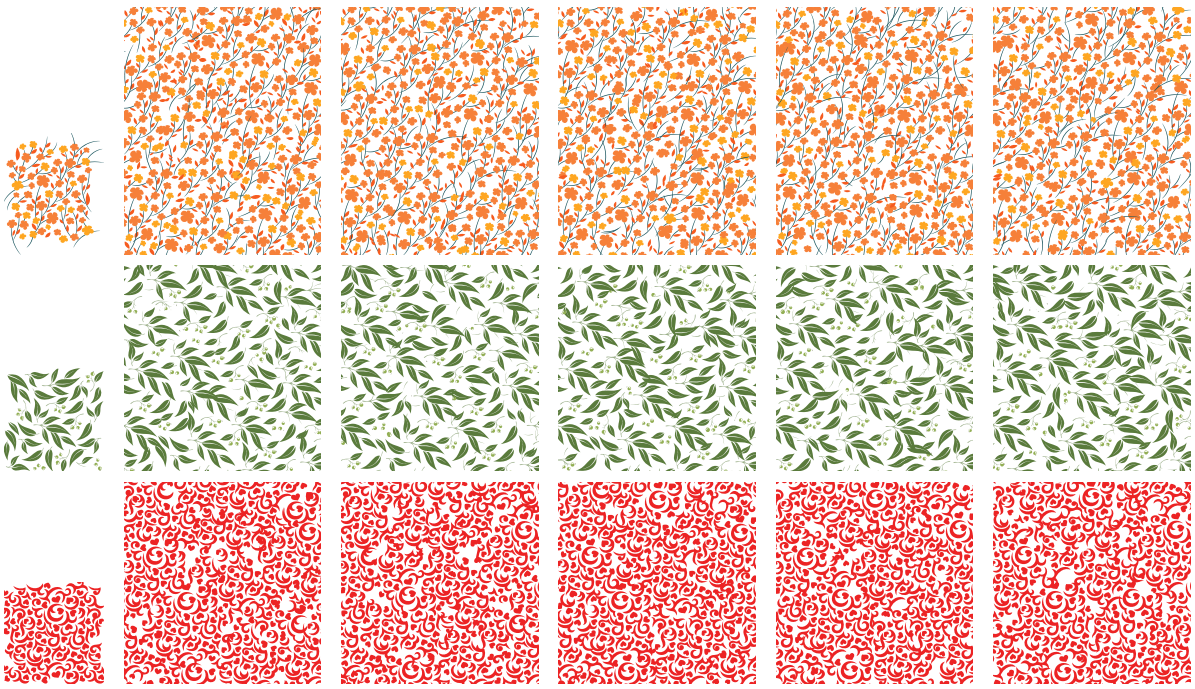


Figure 6.7: *Synthesis results using different hyperparameters* w_s. $w_s = 4$ is the chosen parameter.

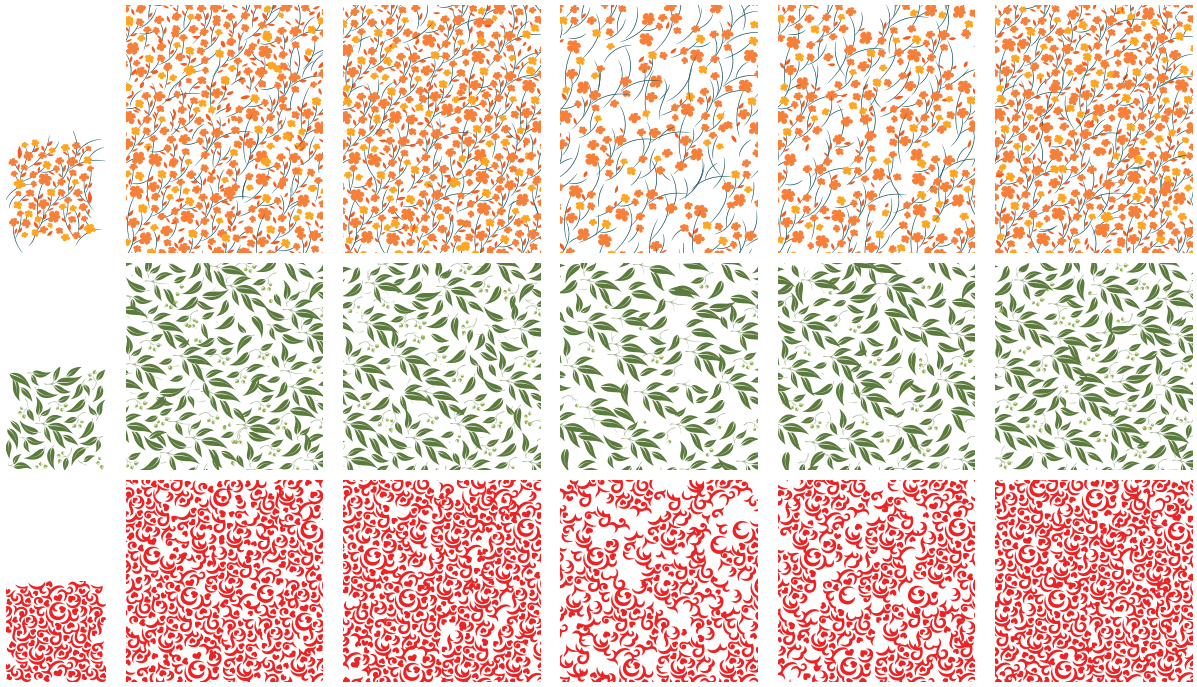


Figure 6.8: *Synthesis results using different combinations of clustering operators.* Our method tend to produce empty or broken regions without these clustering operators.

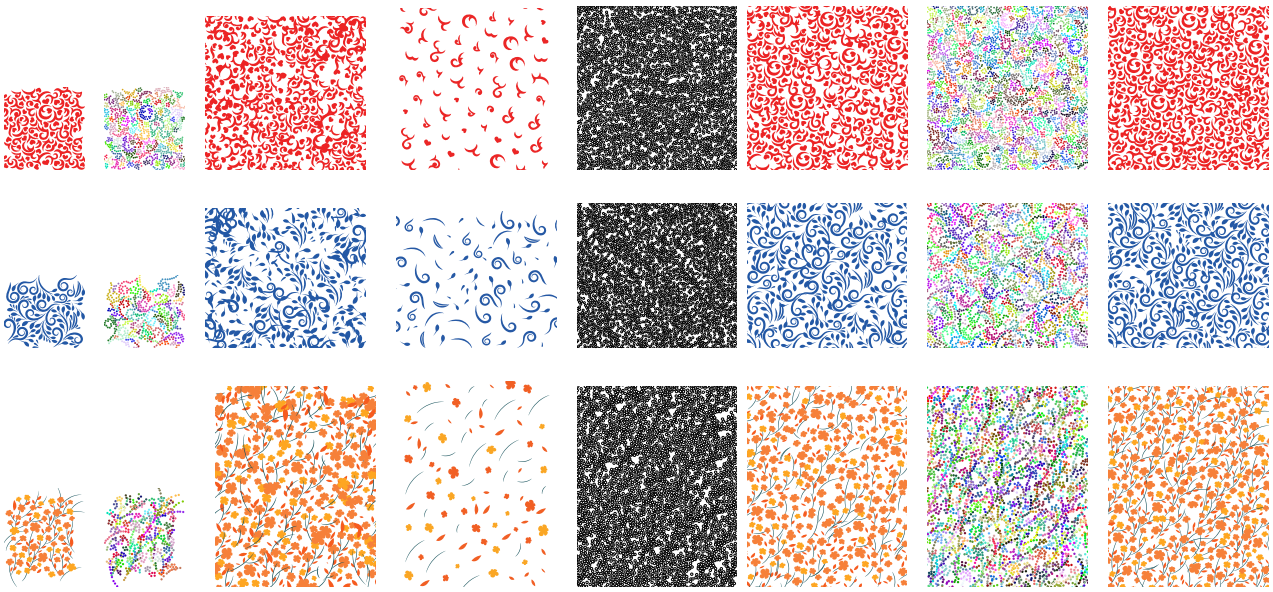


Figure 6.9: *More comparisons with previous methods.* Please see Figures 3.3 and 3.13 for detailed descriptions.

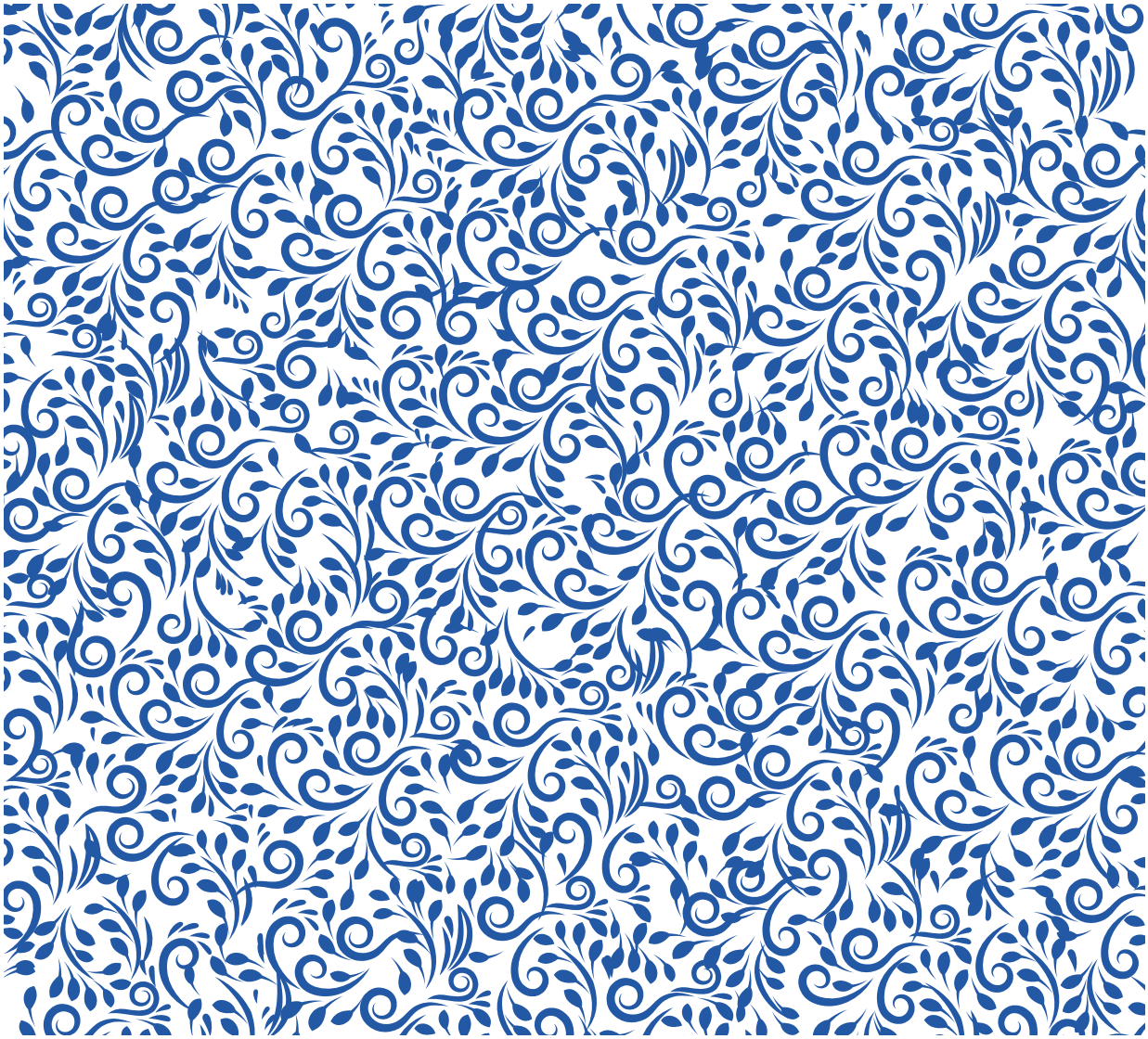


Figure 6.10: *A large synthesis result.* This result is generated by enlarging the exemplar (Figure 3.11j) 4×4 times.



Figure 7.1: *100 original dataset samples without augmentation.*

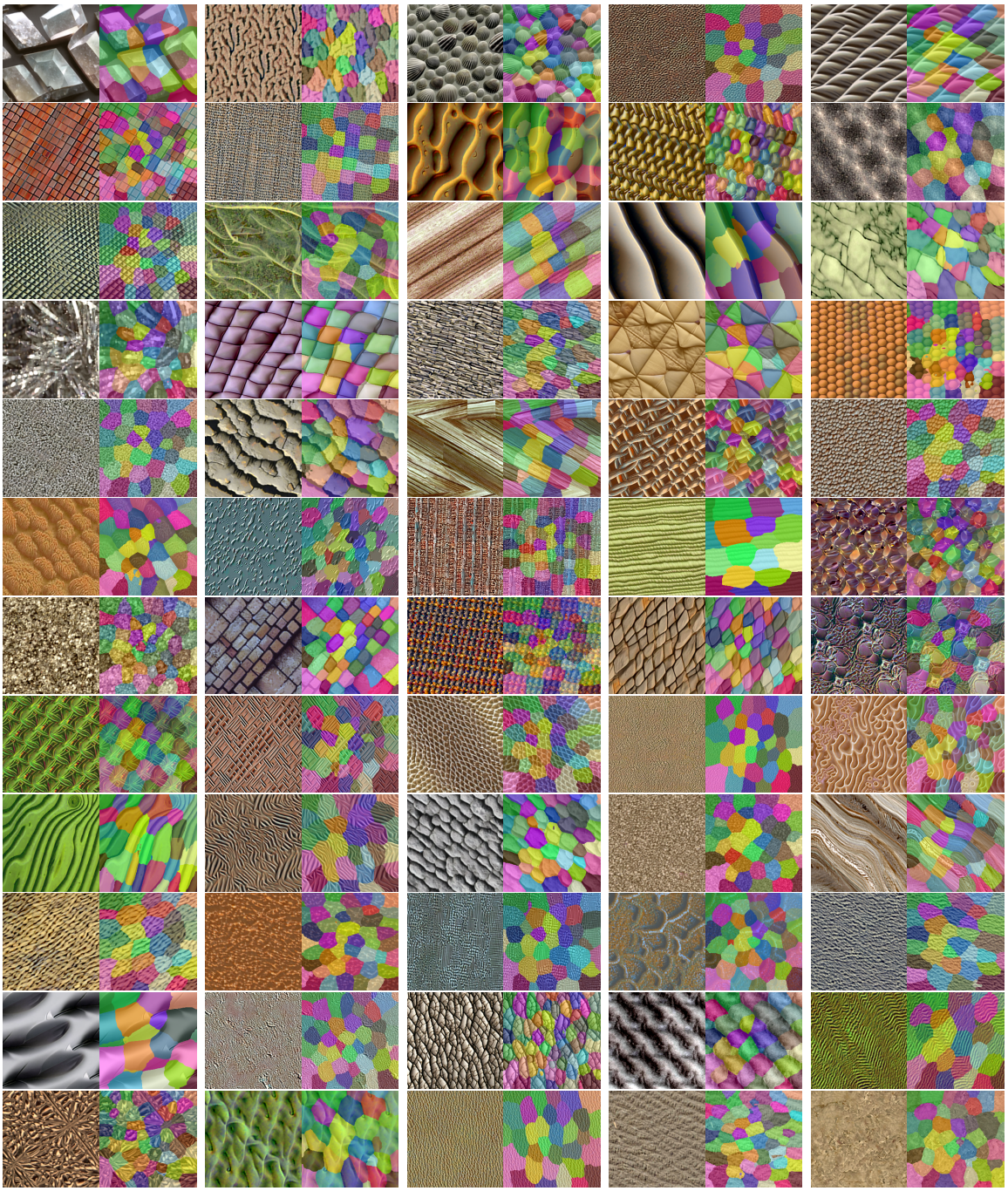


Figure 7.2: *Uncurated texton segmentation results.*

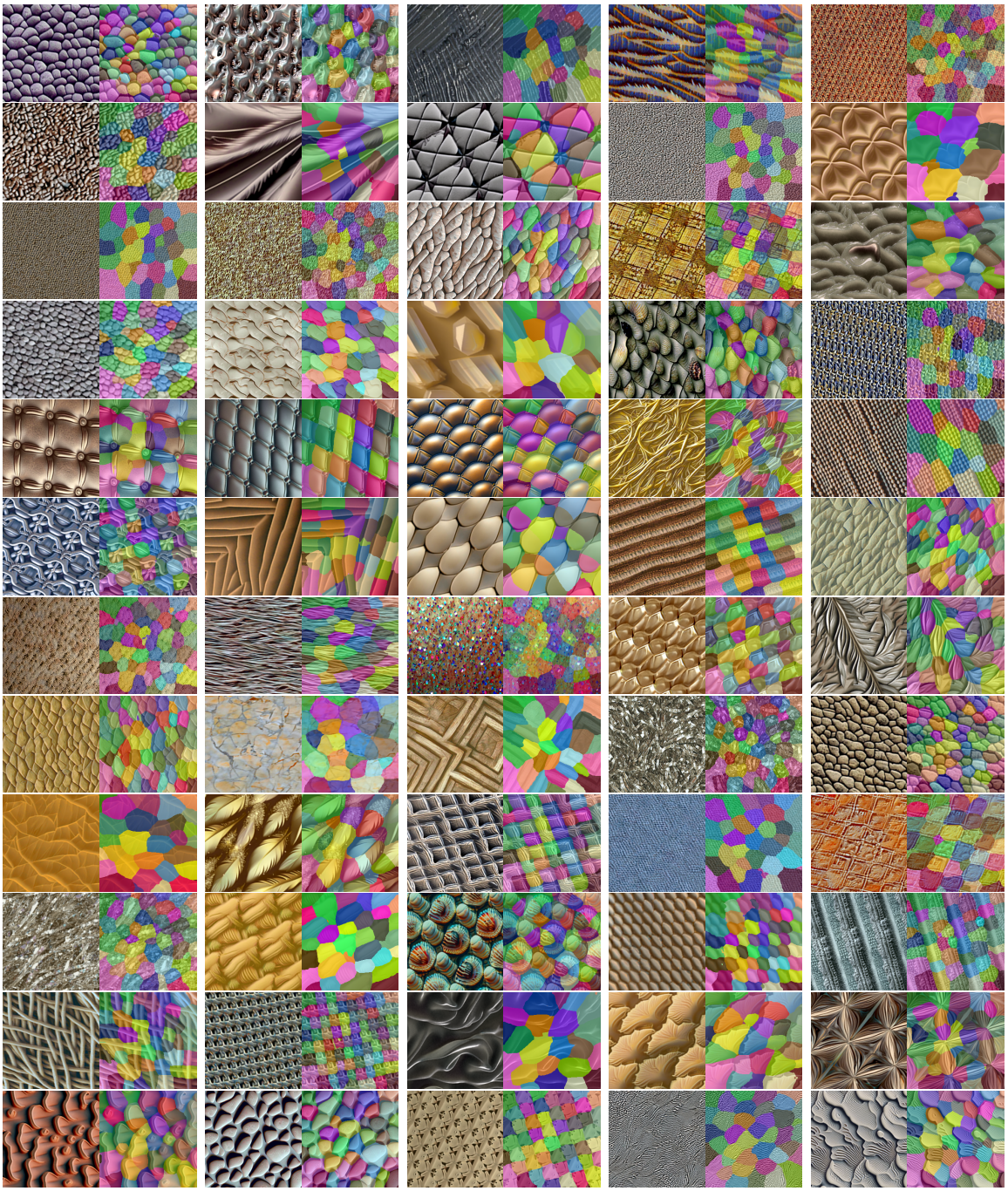


Figure 7.3: *Continuation from Figure 7.2.*



Figure 7.4: *Two texture transfer mechanisms.* Segmentations of the structure-providing textures are displayed on the right.

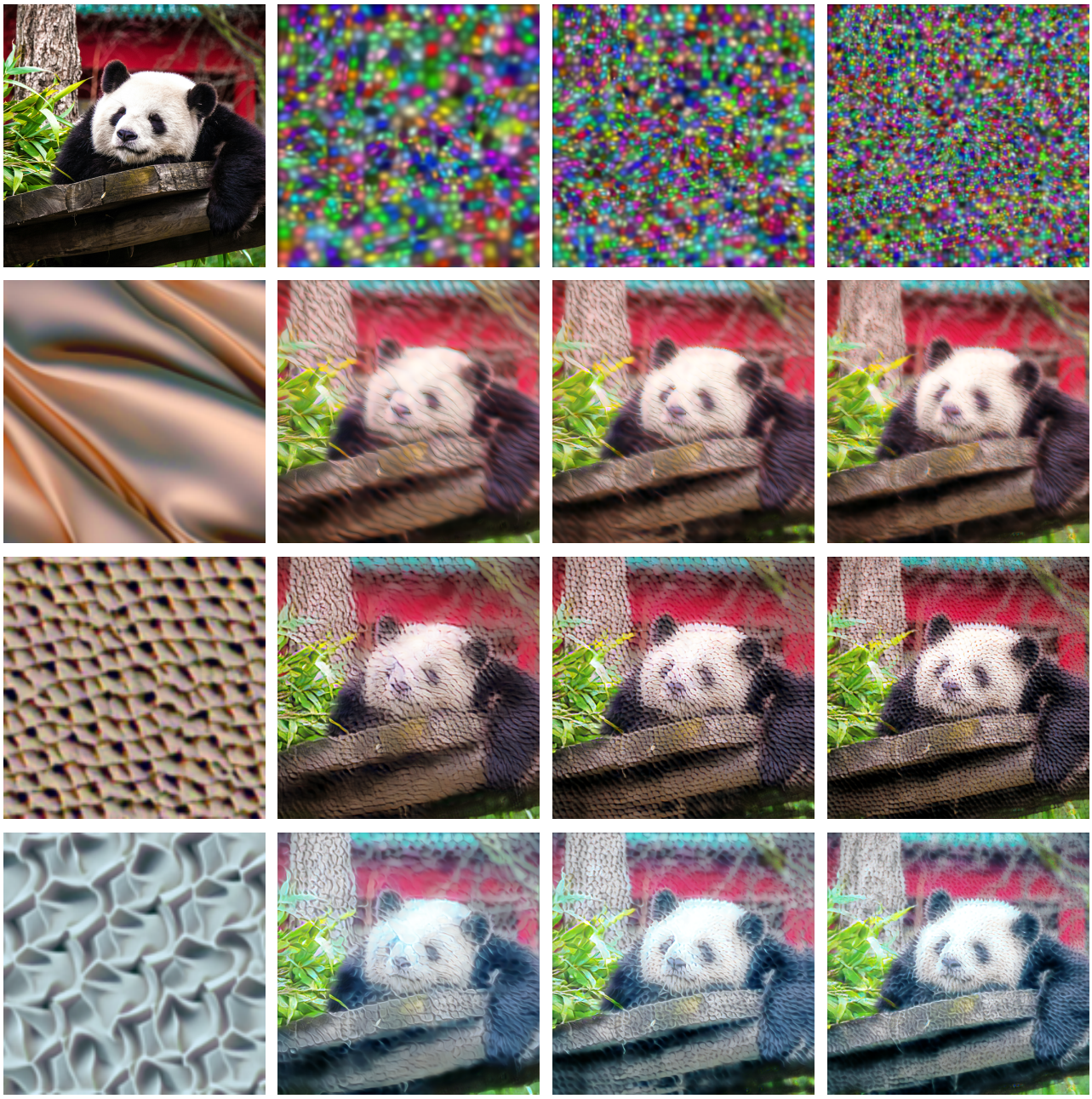


Figure 7.5: *Scale control in texture transfer.* The first row, from left to right: content image, large, median (2/3 of large scale) and small (2/3 of median scale) latent Gaussians. The remaining rows, from left to right: texture, transferred results with large, median and small textons.

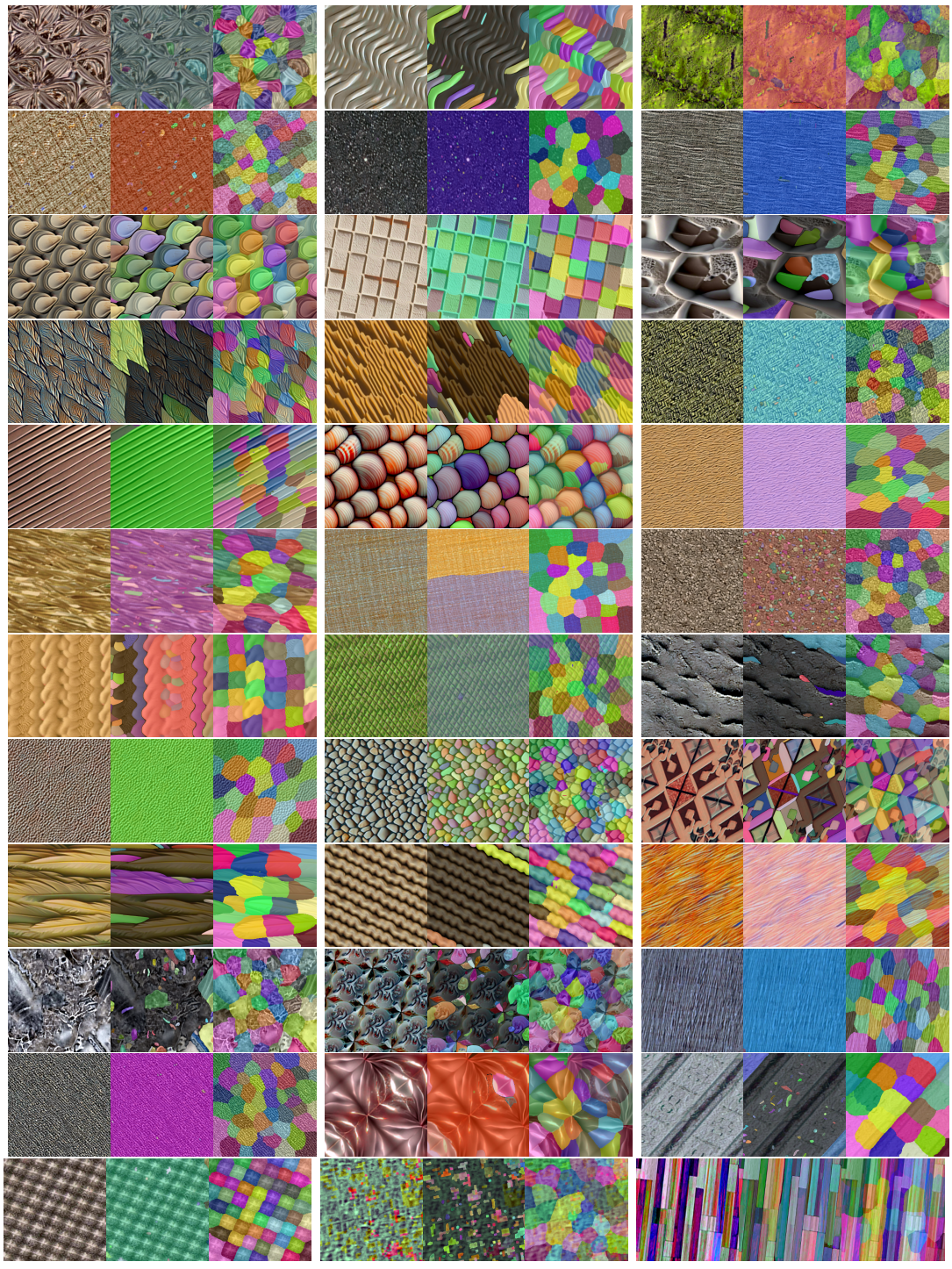


Figure 7.6: *Uncurated comparisons with SA.* Each set contains 3 images: the input (left), result from SA [145] trained with 1 billion masks (middle), and our model's output (right). Notably, despite SA's reliance on extensive supervision, our model successfully extracts textons without supervision.

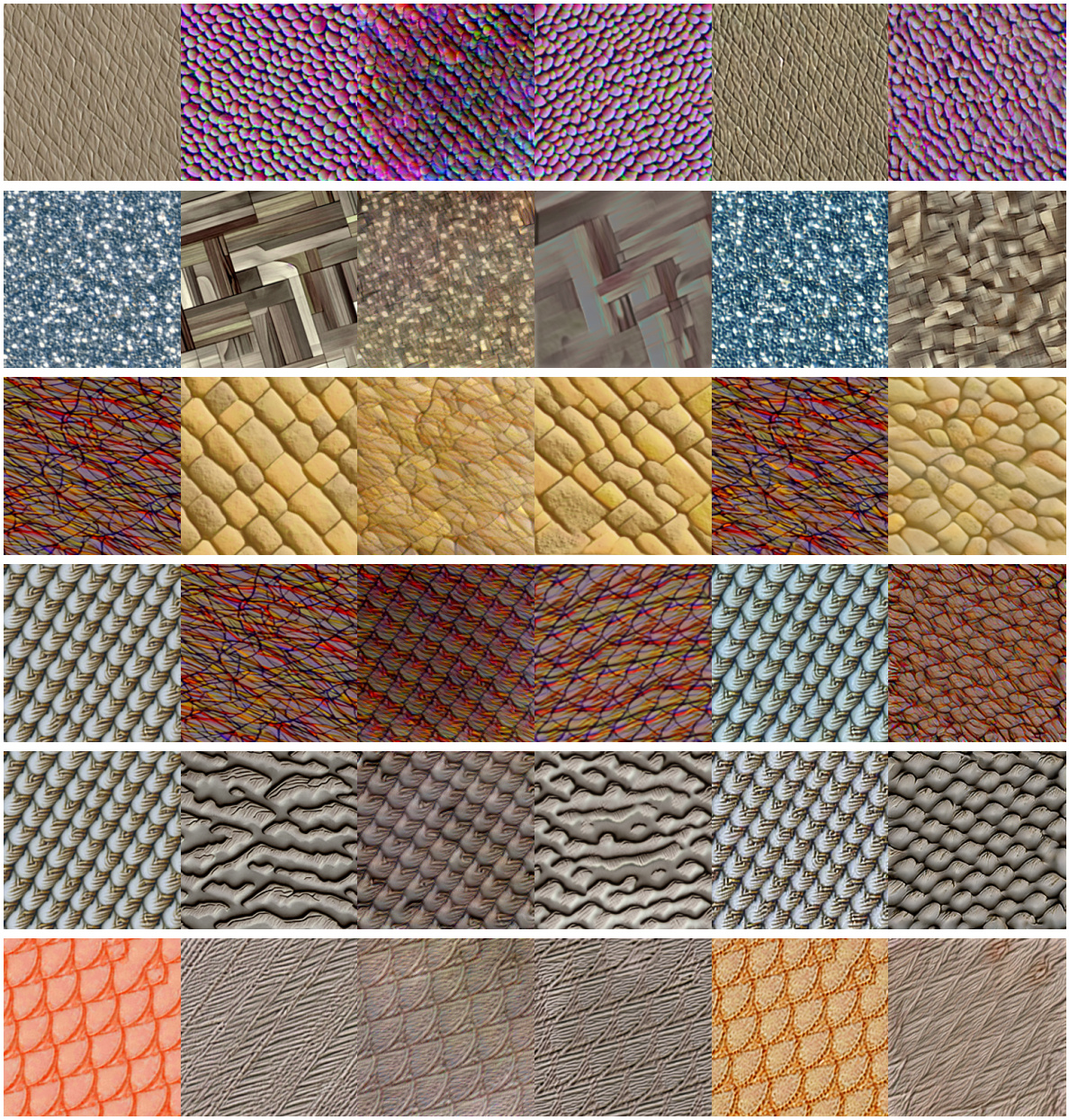


Figure 7.7: *Comparison with state-of-the-art approaches on texture transfer.*

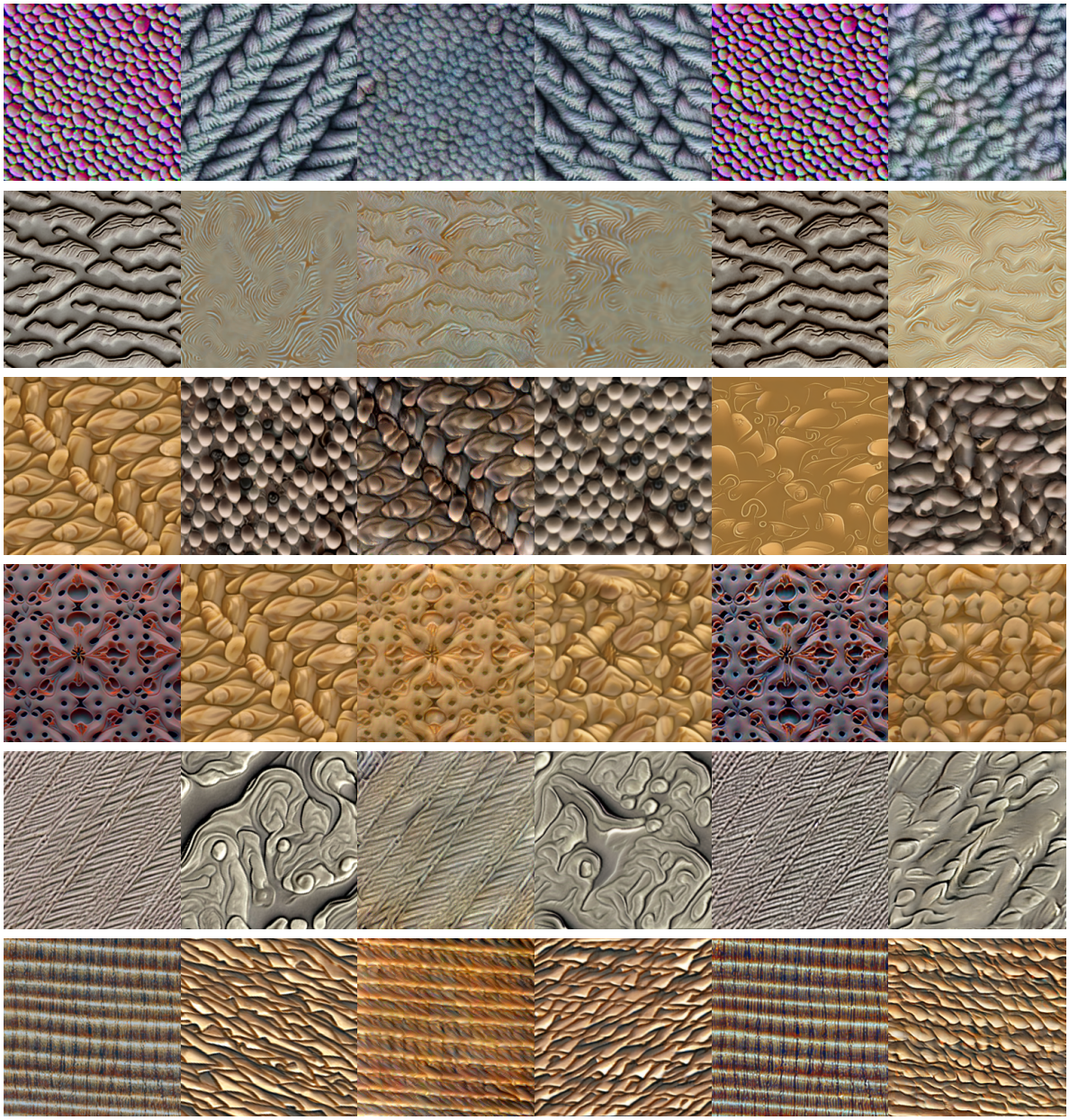


Figure 7.8: *Continuation from Figure 7.7.*

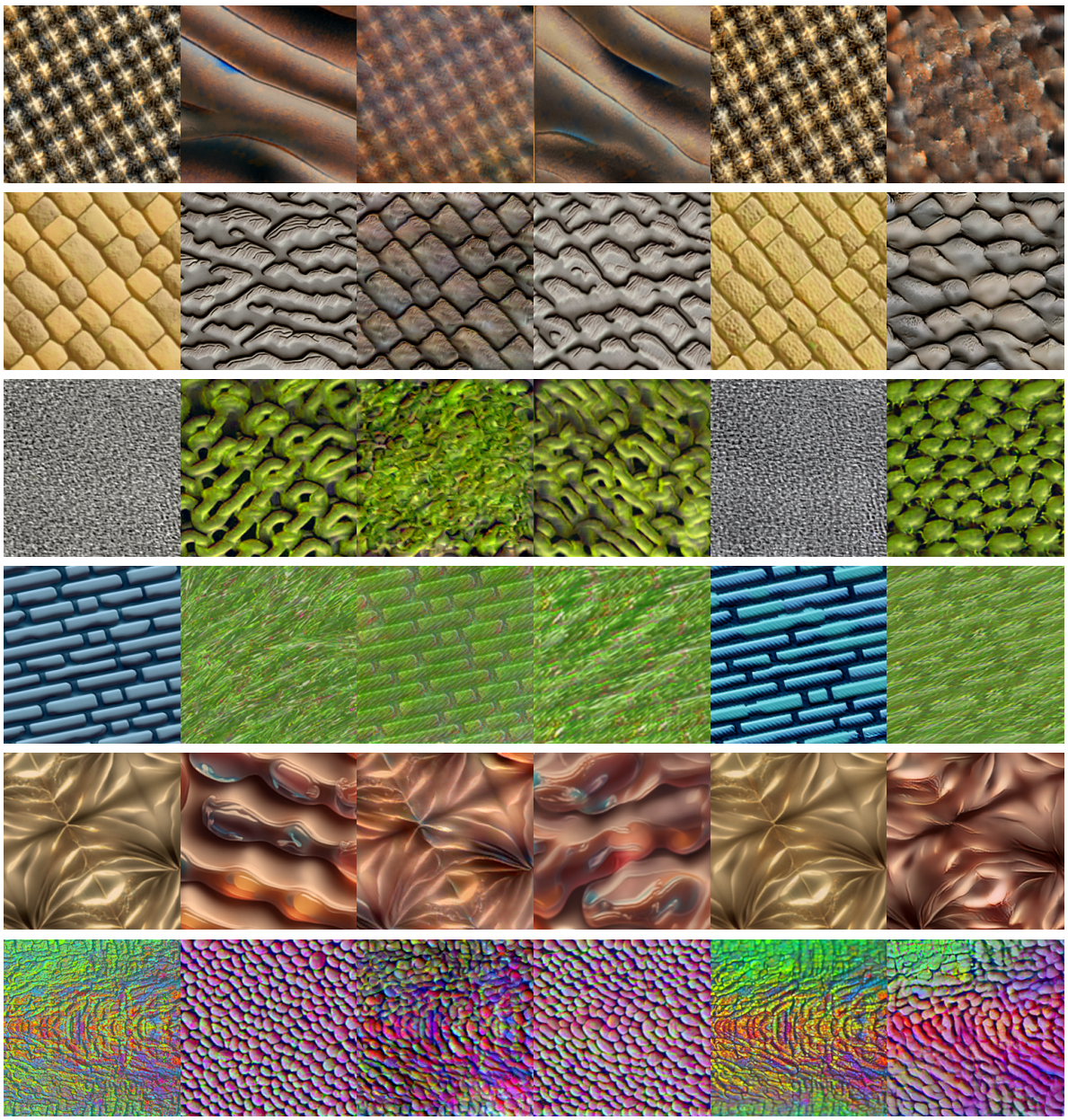


Figure 7.9: *Uncurated comparisons with state-of-the-art approaches on texture transfer.*

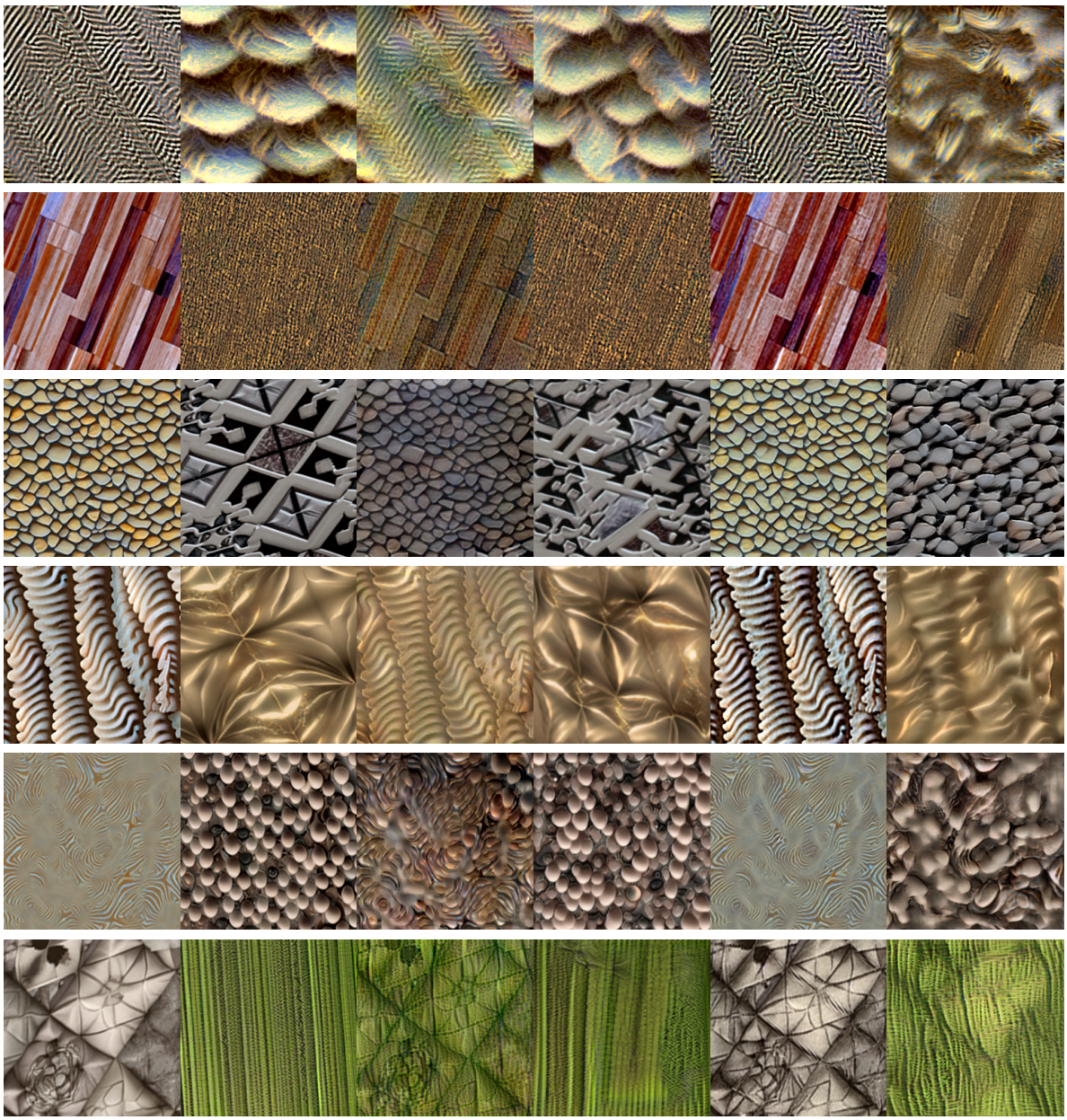


Figure 7.10: *Continuation from Figure 7.9.*

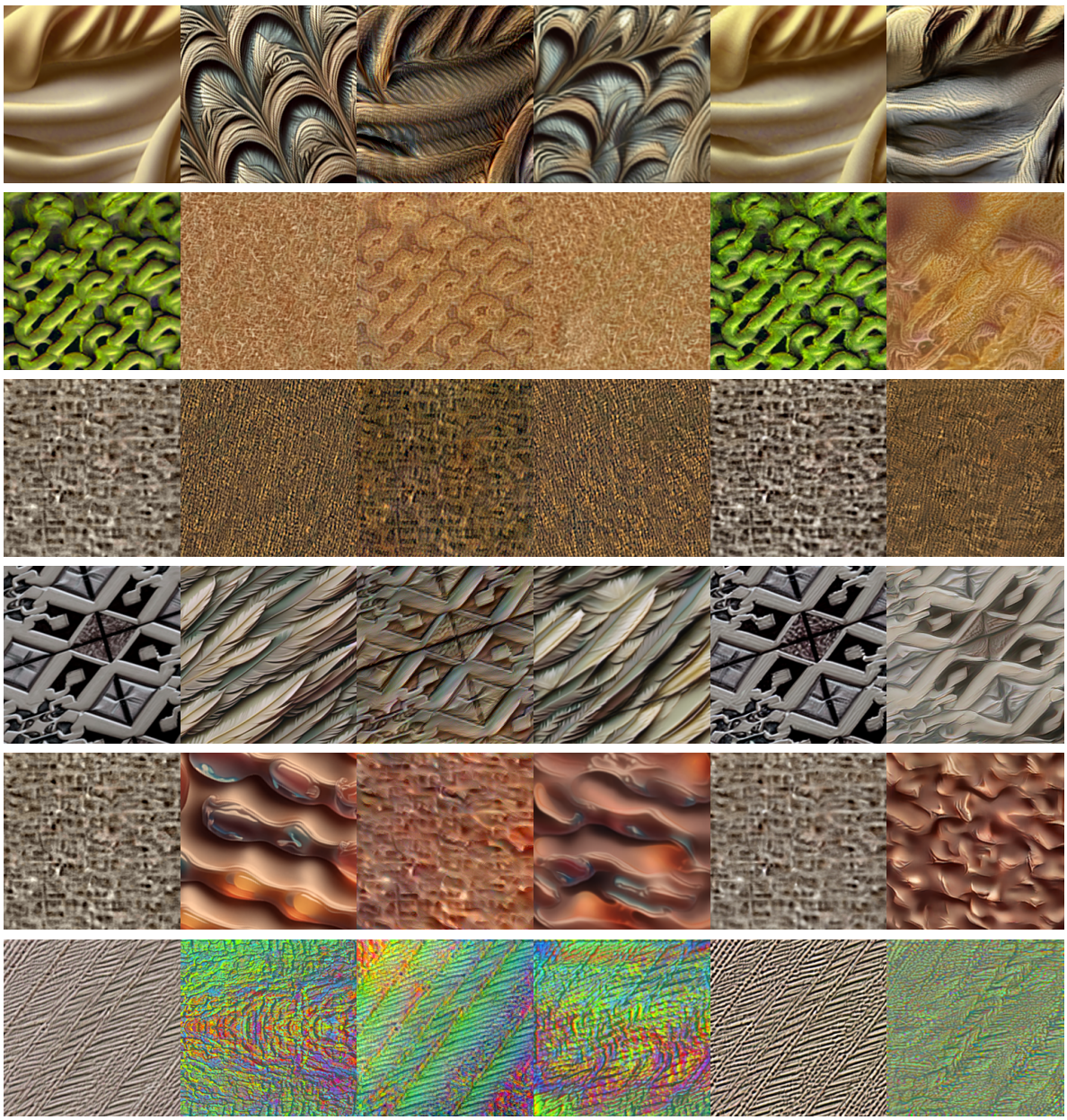


Figure 7.11: *Continuation from Figure 7.9.*



Figure 7.12: *Continuation from Figure 7.9.*

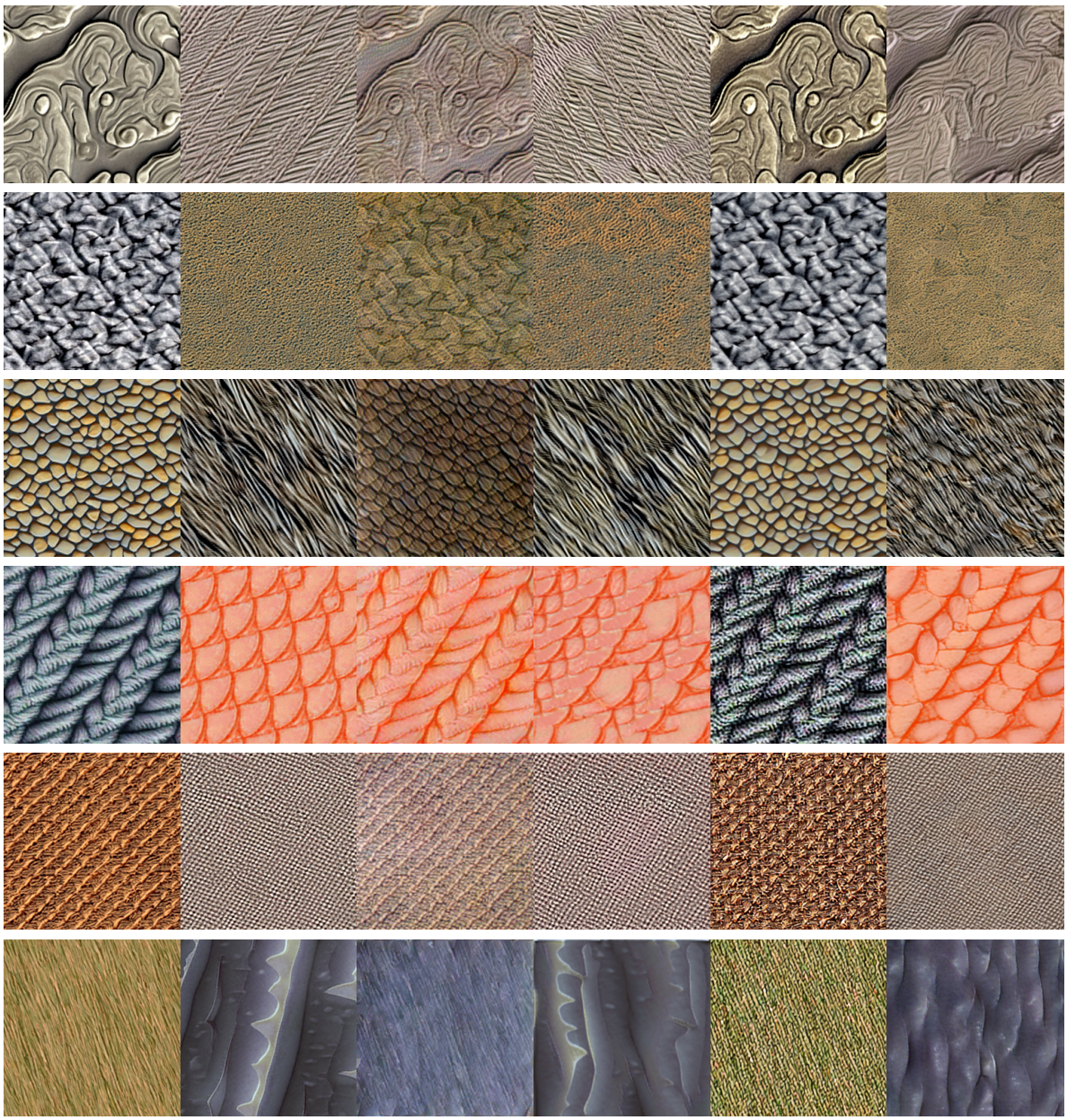


Figure 7.13: *Continuation from Figure 7.9.*

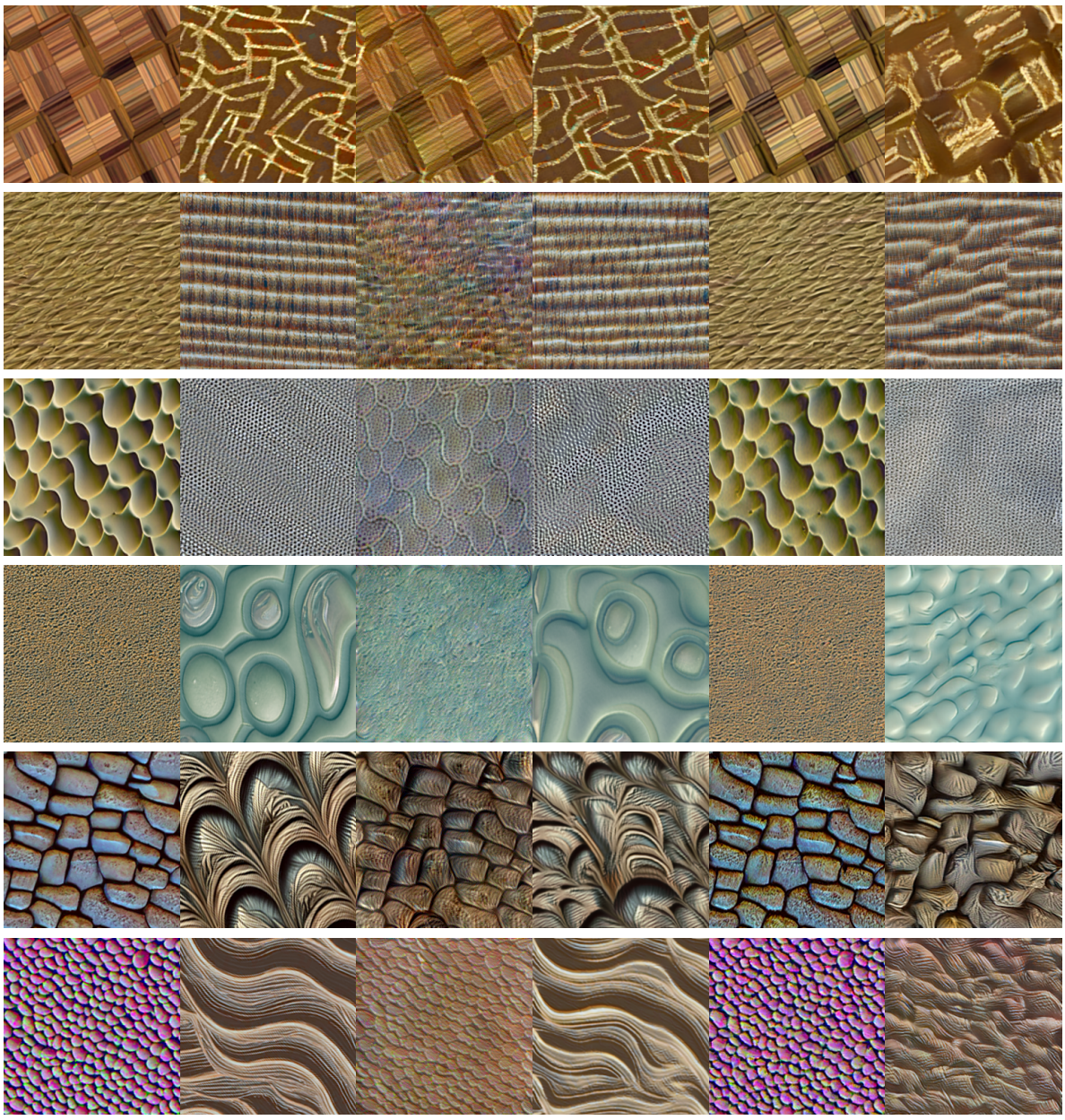


Figure 7.14: *Continuation from Figure 7.9.*

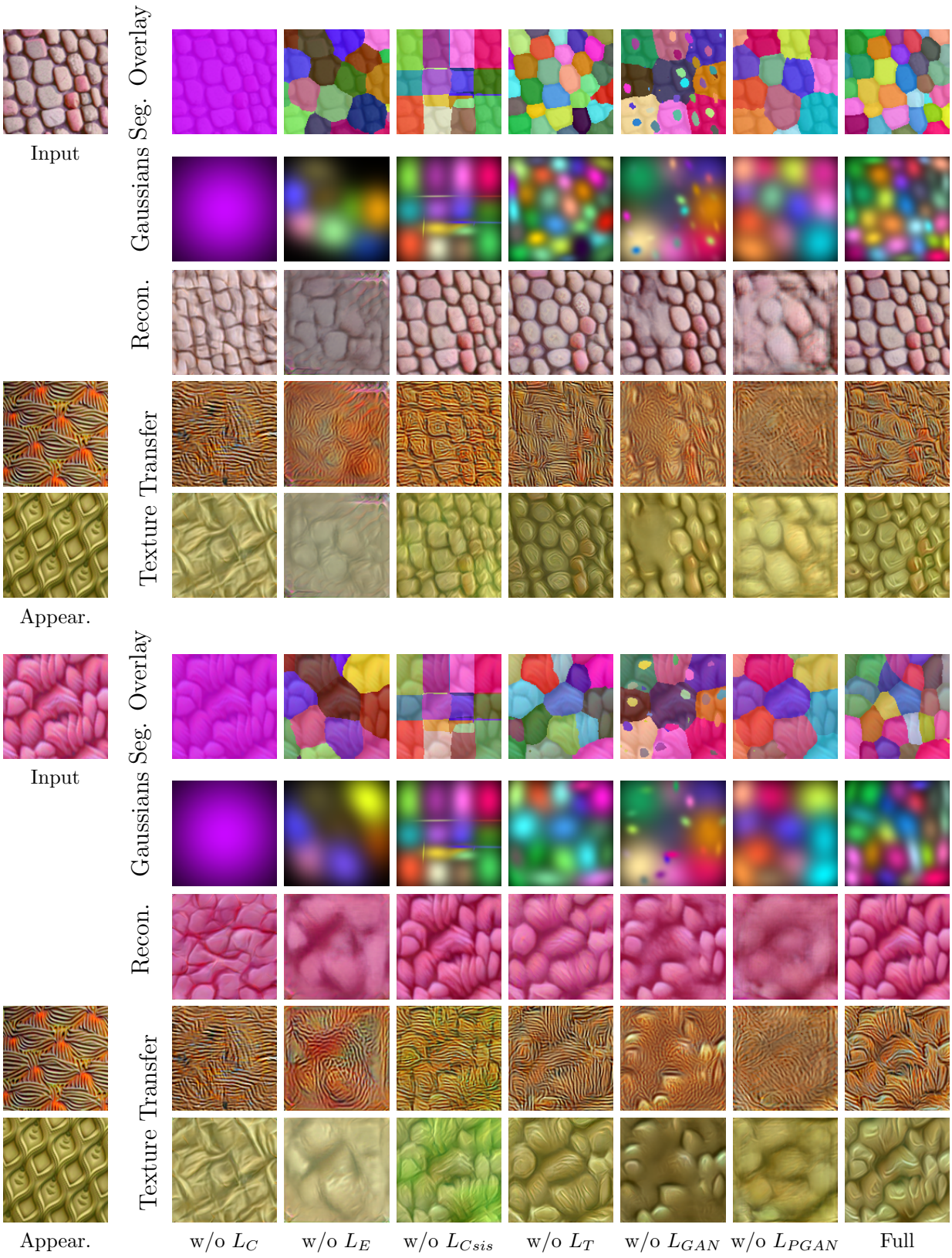


Figure 7.15: *Ablation studies on losses.* We ablate each loss from our implementation and visually compare their results with those using full losses. We visualize the segmentations $\mathbf{S}$, Gaussians $\mathbf{g}$, reconstructions I_r and texture transfer results given appearance-providing textures.

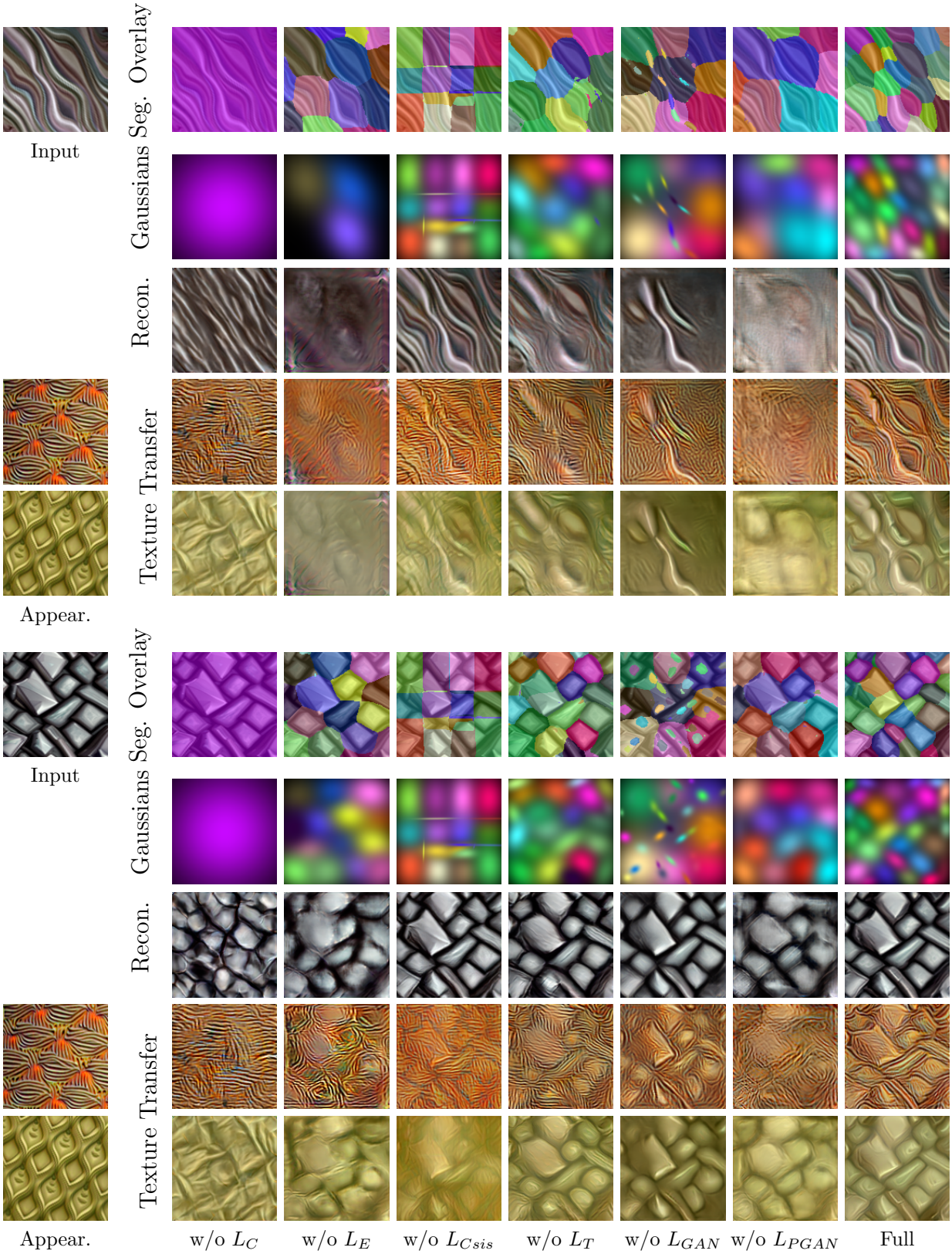


Figure 7.16: *Continuation of Figure 7.15.*

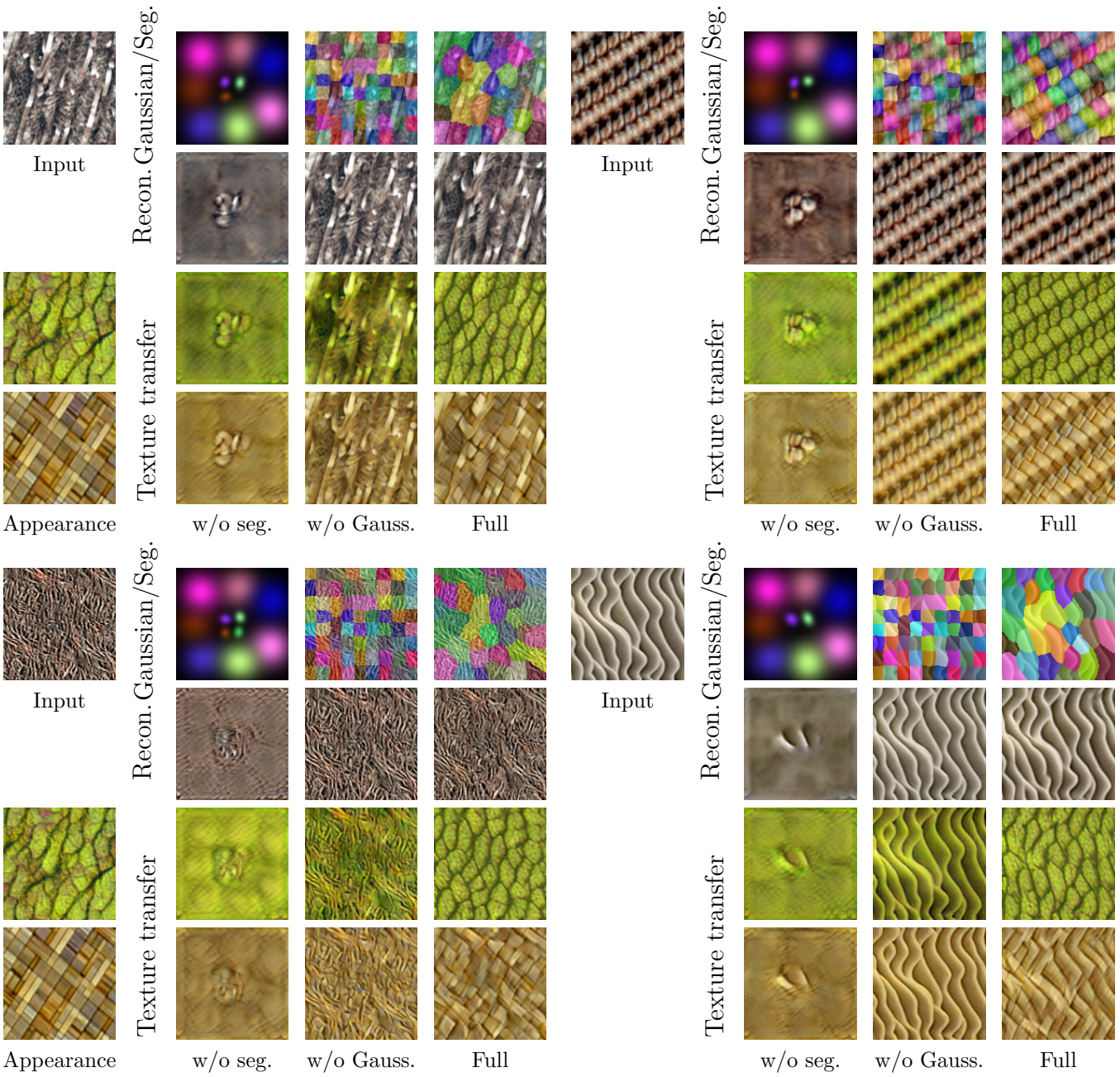


Figure 7.17: *Ablation studies on network components.* We visualize the Gaussians/segmentation masks, reconstruction and texture transfer results given appearance-providing textures.

FC image encoder $\mathcal{E}_{FC}$	Upsample ConvBlock $\mathsf{UpBlock}(n_i, n_o)$
$I \in \mathbb{R}^{H \times W \times 3}$	$x \in \mathbb{R}^{H_x \times W_x \times C_x}$
$\mathsf{ResBlock}(3, 64) \rightarrow \mathbb{R}^{H/2 \times W/2 \times 64}$	$\mathsf{Upsample}(2) \rightarrow \mathbb{R}^{2H_x \times 2W_x \times n_i}$
$\mathsf{ResBlock}(64, 128) \rightarrow \mathbb{R}^{H/4 \times W/4 \times 128}$	$\mathsf{Conv}(n_i, n_o) \rightarrow \mathbb{R}^{2H_x \times 2W_x \times n_o}$
$\mathsf{ResBlock}(128, 256) \rightarrow \mathbb{R}^{H/8 \times W/8 \times 256}$	$\mathsf{BatchNorm2D}(n_o) \xrightarrow{\mathsf{LeakyReLU}(0.2)} \mathbb{R}^{2H_x \times 2W_x \times n_o}$
$\mathsf{ResBlock}(256, 512) \rightarrow \mathbb{R}^{H/16 \times W/16 \times 512}$	
$\mathsf{UpBlock}(512, 256) \rightarrow \mathsf{UpBlock}(256, 382) \rightarrow \mathbf{F}_a \in \mathbb{R}^{H/4 \times W/4 \times 382}$ $\mathsf{UpBlock}(512, 256) \rightarrow \mathsf{UpBlock}(256, 2) \rightarrow \mathbf{V} \in \mathbb{R}^{H/4 \times W/4 \times 2}$	

Table 7.1: *The architecture of* $\mathcal{E}_{FC}$.

Bibliography

[1] Chongyang Ma, Li-Yi Wei, and Xin Tong. Discrete element textures. In *ACM SIGGRAPH 2011 Papers*, SIGGRAPH '11, pages 62:1–62:10, New York, NY, USA, 2011. ACM.

[2] Peihan Tu, Li-Yi Wei, Koji Yatani, Takeo Igarashi, and Matthias Zwicker. Continuous curve textures. *ACM Trans. Graph.*, 39(6), November 2020.

[3] Riccardo Roveri, A Cengiz Öztireli, Sebastian Martin, Barbara Solenthaler, and Markus Gross. Example based repetitive structure synthesis. *Computer Graphics Forum*, 34(5):39–52, 2015.

[4] Peihan Tu, Dani Lischinski, and Hui Huang. Point pattern synthesis via irregular convolution. *Computer Graphics Forum*, 38(5):109–122, 2019.

[5] Rubaiat Habib Kazi, Takeo Igarashi, Shengdong Zhao, and Richard Davis. Vignette: Interactive texture design and manipulation with freeform gestures for pen-and-ink illustration. In *CHI '12*, pages 1727–1736, 2012.

[6] Jingwan Lu, Connelly Barnes, Connie Wan, Paul Asente, Radomir Mech, and Adam Finkelstein. Decobrush: Drawing structured decorative patterns by example. *ACM Trans. Graph.*, 33(4):90:1–90:9, July 2014.

[7] Shizhe Zhou, Changyun Jiang, and Sylvain Lefebvre. Topology-constrained synthesis of vector patterns. *ACM Trans. Graph.*, 33(6):215:1–215:11, November 2014.

[8] Jonàs Martínez, Jérémie Dumas, Sylvain Lefebvre, and Li-Yi Wei. Structure and appearance optimization for controllable shape design. *ACM Trans. Graph.*, 34(6):229:1–229:11, October 2015.

[9] Jonas Zehnder, Stelian Coros, and Bernhard Thomaszewski. Designing structurally-sound ornamental curve networks. *ACM Trans. Graph.*, 35(4):99:1–99:10, July 2016.

[10] Weikai Chen, Xiaolong Zhang, Shiqing Xin, Yang Xia, Sylvain Lefebvre, and Wenping Wang. Synthesis of filigrees for digital fabrication. *ACM Trans. Graph.*, 35(4):98:1–98:13, July 2016.

[11] Christian Schumacher, Bernhard Thomaszewski, and Markus Gross. Stenciling: Designing structurally-sound surfaces with decorative patterns. *Computer Graphics Forum*, 35(5):101–110, 2016.

[12] Chongyang Ma, Li-Yi Wei, Sylvain Lefebvre, and Xin Tong. Dynamic element textures. *ACM Trans. Graph.*, 32(4):90:1–90:10, July 2013.

[13] Takashi Ijiri, Radomír Mech, Takeo Igarashi, and Gavin Miller. An example-based procedural system for element arrangement. In *Computer Graphics Forum*, volume 27, pages 429–436. Wiley Online Library, 2008.

[14] T. Hurtut, P.-E. Landes, J. Thollot, Y. Gousseau, R. Drouillhet, and J.-F. Coeurjolly. Appearance-guided synthesis of element arrangements by example. In *NPAR '09*, pages 51–60, 2009.

[15] Pascal Barla, Simon Breslav, Joëlle Thollot, François Sillion, and Lee Markosian. Stroke pattern analysis and synthesis. In *Computer Graphics Forum*, volume 25, pages 663–671. Wiley Online Library, 2006.

[16] Ryo Suzuki, Tom Yeh, Koji Yatani, and Mark D Gross. Autocomplete textures for 3d printing. *arXiv preprint arXiv:1703.05700*, 2017.

[17] Chen-Yuan Hsu, Li-Yi Wei, Lihua You, and Jian Jun Zhang. Brushing element fields. In *SIGGRAPH Asia 2018 Technical Briefs*, SA '18, pages 6:1–6:4, 2018.

[18] Chen-Yuan Hsu, Li-Yi Wei, Lihua You, and Jian Jun Zhang. Autocomplete element fields. In *CHI '20*, pages 1–13, 2020.

[19] Li-Yi Wei, Sylvain Lefebvre, Vivek Kwatra, and Greg Turk. State of the art in example-based texture synthesis. In *Eurographics 2009, State of the Art Report, EG-STAR*, pages 93–117. Eurographics Association, 2009.

[20] Jingwan Lu, Fisher Yu, Adam Finkelstein, and Stephen DiVerdi. Helpinghand: Example-based stroke stylization. *ACM Trans. Graph.*, 31(4):46:1–46:10, July 2012.

[21] Pierre-Edouard Landes, Bruno Galerne, and Thomas Hurtut. A shape-aware model for discrete texture synthesis. *Computer Graphics Forum*, 32(4):67–76, 2013.

[22] Li-Yi Wei and Marc Levoy. Fast texture synthesis using tree-structured vector quantization. In *SIGGRAPH '00*, pages 479–488, 2000.

[23] Li-Yi Wei and Marc Levoy. Texture synthesis over arbitrary manifold surfaces. In *SIGGRAPH '01*, pages 355–360, 2001.

[24] Jun Xing, Hsiang-Ting Chen, and Li-Yi Wei. Autocomplete painting repetitions. *ACM Trans. Graph.*, 33(6):172:1–172:11, November 2014.

[25] Hans Pedersen and Karan Singh. Organic labyrinths and mazes. In *NPAR '06*, pages 79–86, 2006.

[26] Hugo Loi, Thomas Hurtut, Romain Vergne, and Joelle Thollot. Programmable 2d arrangements for element texture design. *ACM Trans. Graph.*, 36(4), May 2017.

[27] Christian Santoni and Fabio Pellacini. gtangle: A grammar for the procedural generation of tangle patterns. *ACM Trans. Graph.*, 35(6):182:1–182:11, November 2016.

[28] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. Image style transfer using convolutional neural networks. In *CVPR '16*, pages 2414–2423, 2016.

[29] Pravin Bhat, Stephen Ingram, and Greg Turk. Geometric texture synthesis by example. In *SGP '04*, pages 41–44, 2004.

[30] Kun Zhou, Xin Huang, Xi Wang, Yiying Tong, Mathieu Desbrun, Baining Guo, and Heung-Yeung Shum. Mesh quilting for geometric texture synthesis. *ACM Trans. Graph.*, 25(3):690–697, July 2006.

[31] Howard Zhou, Jie Sun, Greg Turk, and James M. Rehg. Terrain synthesis from digital elevation models. *IEEE Transactions on Visualization and Computer Graphics*, 13(4):834–848, July 2007.

[32] Aaron Hertzmann, Nuria Oliver, Brian Curless, and Steven M. Seitz. Curve analogies. In *EGRW '02*, pages 233–246, 2002.

[33] Paul Merrell and Dinesh Manocha. Example-based curve synthesis. *Computers & Graphics*, 34(4):304–311, 2010.

[34] Mathieu Nancel and Andy Cockburn. Causality: A conceptual model of interaction history. In *CHI '14*, pages 1777–1786, 2014.

[35] Jun Xing, Li-Yi Wei, Takaaki Shiratori, and Koji Yatani. Autocomplete hand-drawn animations. *ACM Trans. Graph.*, 34(6):169:1–169:11, October 2015.

[36] Mengqi Peng, Jun Xing, and Li-Yi Wei. Autocomplete 3D sculpting. *ACM Trans. Graph.*, 37(4):132:1–132:15, July 2018.

[37] Mengqi Peng, Li-Yi Wei, Rubaiat Habib Kazi, and Vladimir G. Kim. Autocomplete animated sculpting. In *UIST '20*, 2020.

[38] C. Lawrence Zitnick. Handwriting beautification using token means. *ACM Trans. Graph.*, 32(4):53:1–53:8, July 2013.

[39] Hsiang-Ting Chen, Li-Yi Wei, and Chun-Fa Chang. Nonlinear revision control for images. *ACM Trans. Graph.*, 30(4):105:1–105:10, July 2011.

[40] Yuki Koyama, Daisuke Sakamoto, and Takeo Igarashi. Selph: Progressive learning and support of manual photo color enhancement. In *CHI '16*, pages 2520–2532, 2016.

[41] Xiaojun Bian, Li-Yi Wei, and Sylvain Lefebvre. Tile-based pattern design with topology control. *Proc. ACM Comput. Graph. Interact. Tech.*, 1(1):23:1–23:15, July 2018.

[42] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[43] Kaspar Riesen and Horst Bunke. Approximate graph edit distance computation by means of bipartite graph matching. *Image and Vision computing*, 27(7):950–959, 2009.

[44] Jérémie Dumas, Jonàs Martínez, Sylvain Lefebvre, and Li-Yi Wei. Printable aggregate elements. *arXiv preprint arXiv:1811.02626*, 2018.

[45] Alexei A. Efros and William T. Freeman. Image quilting for texture synthesis and transfer. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, page 341–346, New York, NY, USA, 2001. Association for Computing Machinery.

[46] Lin Liang, Ce Liu, Ying-Qing Xu, Baining Guo, and Heung-Yeung Shum. Real-time texture synthesis by patch-based sampling. *ACM Trans. Graph.*, 20(3):127–150, July 2001.

[47] Connelly Barnes, Eli Shechtman, Adam Finkelstein, and Dan B Goldman. Patchmatch: A randomized correspondence algorithm for structural image editing. *ACM Trans. Graph.*, 28(3):24:1–24:11, July 2009.

[48] Xiaobai Chen, Tom Funkhouser, Dan B Goldman, and Eli Shechtman. Non-parametric texture transfer using meshmatch. *Adobe Technical Report*, (2), 2012.

[49] Vivek Kwatra, Irfan Essa, Aaron Bobick, and Nipun Kwatra. Texture optimization for example-based synthesis. *ACM Trans. Graph.*, 24(3):795–802, jul 2005.

[50] Peihan Tu. Continuous curve textures source code, 2020. `https://github.com/tph9608/continuous-curve-texture/`.

[51] Alexandre Kaspar, Boris Neubert, Dani Lischinski, Mark Pauly, and Johannes Kopf. Self tuning texture optimization. *Comput. Graph. Forum*, 34(2):349–359, May 2015.

[52] Hao-Da Huang, Xin Tong, and Wen-Cheng Wang. Accelerated parallel texture optimization. *Journal of Computer Science and Technology*, 22(5):761–769, 2007.

[53] Kenshi Takayama, Olga Sorkine, Andrew Nealen, and Takeo Igarashi. Volumetric modeling with diffusion surfaces. In *SIGGRAPH ASIA '10*, 2010.

[54] Lvdi Wang, Kun Zhou, Yizhou Yu, and Baining Guo. Vector solid textures. *ACM Transactions on Graphics (TOG)*, 29(4):1–8, 2010.

[55] Lvdi Wang, Yizhou Yu, Kun Zhou, and Baining Guo. Multiscale vector volumes. *ACM Transactions on Graphics (TOG)*, 30(6):1–8, 2011.

[56] Weikai Chen, Yuexin Ma, Sylvain Lefebvre, Shiqing Xin, Jonàs Martínez, and wenping wang. Fabricable tile decors. *ACM Trans. Graph.*, 36(6):175:1–175:15, November 2017.

[57] Yifei Li, David E. Breen, James McCann, and Jessica Hodgins. Algorithmic quilting pattern generation for pieced quilts. In *Proceedings of the 45th Graphics Interface Conference on Proceedings of Graphics Interface 2019*, GI'19, pages 13:1–13:9, 2019.

[58] Paul Guerrero, Gilbert Bernstein, Wilmot Li, and Niloy J. Mitra. Patex: Exploring pattern variations. *ACM Trans. Graph.*, 35(4):48:1–48:13, July 2016.

[59] Kin Chung Kwan, Lok Tsun Sinn, Chu Han, Tien-Tsin Wong, and Chi-Wing Fu. Pyramid of arclength descriptor for generating collage of shapes. *ACM Trans. Graph.*, 35(6), November 2016.

[60] Jennifer Jacobs, Joel Brandt, Radomír Mech, and Mitchel Resnick. Extending manual drawing practices with artist-centric programming tools. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, pages 590:1–590:13, New York, NY, USA, 2018. ACM.

[61] Jianwei Guo, Haiyong Jiang, Bedrich Benes, Oliver Deussen, Xiaopeng Zhang, Dani Lischinski, and Hui Huang. Inverse procedural modeling of branching structures by inferring l-systems. *ACM Trans. Graph.*, 39(5), June 2020.

[62] Yaroslav Ganin, Sergey Bartunov, Yujia Li, Ethan Keller, and Stefano Saliceti. Computer-aided design as language, 2021.

[63] Connelly Barnes and Fang-Lue Zhang. A survey of the state-of-the-art in patch-based synthesis. *Computational Visual Media*, 3(1):3–20, 2017.

[64] Brian Price and William Barrett. Object-based vectorization for interactive image editing. *The Visual Computer*, 22(9):661–670, 2006.

[65] Jean-Dominique Favreau, Florent Lafarge, and Adrien Bousseau. Photo2clipart: Image abstraction and vectorization using layered linear gradients. *ACM Trans. Graph.*, 36(6), November 2017.

[66] Li-Yi Wei. Multi-class blue noise sampling. *ACM Trans. Graph.*, 29(4):79:1–79:8, July 2010.

[67] Pascal Guehl, Rémi Allegre, J-M Dischler, Bedrich Benes, and Eric Galin. Semi-procedural textures using point process texture basis functions. In *Computer Graphics Forum*, volume 39, pages 159–171. Wiley Online Library, 2020.

[68] Reza Adhitya Saputra, Craig S. Kaplan, and Paul Asente. Animationpak: Packing elements with scripted animations. In *Graphics Interface '20*, 2020. `https://openreview.net/forum?id=sr89orrDo-o`.

[69] Alejo Hausner. Simulating decorative mosaics. In *SIGGRAPH '01*, pages 573–580, 2001.

[70] Changqing Zou, Junjie Cao, Warunika Ranaweera, Ibraheem Alhashim, Ping Tan, Alla Sheffer, and Hao Zhang. Legible compact calligrams. *ACM Trans. Graph.*, 35(4), July 2016.

[71] Michael T Wong, Douglas E Zongker, and David H Salesin. Computer-generated floral ornament. In *SIGGRAPH '98*, pages 423–434, 1998.

[72] Giacomo Nazzaro, Enrico Puppo, and Fabio Pellacini. Decosurf: Recursive geodesic patterns on triangle meshes, 2020.

[73] Giacomo Nazzaro, Enrico Puppo, and Fabio Pellacini. Geotangle: Interactive design of geodesic tangle patterns on surfaces. *ACM Trans. Graph.*, 41(2), nov 2021.

[74] Ondrej Šťava, Bedrich Beneš, Radomir Měch, Daniel G Aliaga, and Peter Krištof. Inverse procedural modeling by automatic generation of l-systems. In *Computer Graphics Forum*, volume 29, pages 665–674. Wiley Online Library, 2010.

[75] Yi-Ting Yeh and Radomír Měch. Detecting symmetries and curvilinear arrangements in vector art. In *Computer Graphics Forum*, volume 28, pages 707–716. Wiley Online Library, 2009.

[76] Lena Gieseke, Paul Asente, Radomir Mech, Bedrich Benes, and Martin Fuchs. A survey of control mechanisms for creative pattern generation. *Computer Graphics Forum*, 2021.

[77] Jingwan Lu, Connelly Barnes, Stephen DiVerdi, and Adam Finkelstein. Realbrush: Painting with examples of physical media. *ACM Trans. Graph.*, 32(4), July 2013.

[78] Noa Fish, Lilach Perry, Amit Bermano, and Daniel Cohen-Or. Sketchpatch: Sketch stylization via seamless patch-level synthesis. *ACM Trans. Graph.*, 39(6), 12 2020.

[79] Aaron Hertzmann, Charles E. Jacobs, Nuria Oliver, Brian Curless, and David H. Salesin. Image analogies. In *SIGGRAPH '01*, pages 327–340, New York, NY, USA, 2001. Association for Computing Machinery.

[80] Amir Rosenberger, Daniel Cohen-Or, and Dani Lischinski. Layered shape synthesis: Automatic generation of control maps for non-stationary textures. *ACM Trans. Graph.*, 28(5):1–9, December 2009.

[81] Yitzchak David Lockerman, Basile Sauvage, Rémi Allègre, Jean-Michel Dischler, Julie Dorsey, and Holly Rushmeier. Multi-scale label-map extraction for texture synthesis. *ACM Trans. Graph.*, 35(4):140:1–140:12, July 2016.

[82] Jakub Fišer, Ondřej Jamriška, Michal Lukáč, Eli Shechtman, Paul Asente, Jingwan Lu, and Daniel Sýkora. Stylit: Illumination-guided example-based stylization of 3d renderings. *ACM Trans. Graph.*, 35(4), July 2016.

[83] Serge Belongie, Jitendra Malik, and Jan Puzicha. Shape context: A new descriptor for shape matching and object recognition. In *Advances in neural information processing systems*, pages 831–837, 2001.

[84] W3. Svg rendering model, 2020. https://www.w3.org/TR/SVG/render.html.

[85] Robert L Cook. Stochastic sampling in computer graphics. *ACM Transactions on Graphics (TOG)*, 5(1):51–72, 1986.

[86] Sylvain Lefebvre and Hugues Hoppe. Appearance-space texture synthesis. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pages 541–548, New York, NY, USA, 2006. ACM.

[87] WWCRE Schapire and Yoram Singer. Learning to order things. *Advances in Neural Information Processing Systems*, 10:451, 1998.

[88] Muxingzi Li, Florent Lafarge, and Renaud Marlet. Approximating shapes in images with low-complexity polygons. In *The IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[89] Lifeng Liu and Stan Sclaroff. Region segmentation via deformable model-guided split and merge. In *Proceedings Eighth IEEE International Conference on Computer Vision. ICCV 2001*, volume 1, pages 98–104. IEEE, 2001.

[90] Ulrike Von Luxburg. A tutorial on spectral clustering. *Statistics and computing*, 17(4):395–416, 2007.

[91] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Transactions on pattern analysis and machine intelligence*, 22(8):888–905, 2000.

[92] Yang Zhou, Zhen Zhu, Xiang Bai, Dani Lischinski, Daniel Cohen-Or, and Hui Huang. Non-stationary texture synthesis by adversarial expansion. *ACM Transactions on Graphics (TOG)*, 37(4):1–13, 2018.

[93] Guilin Liu, Rohan Taori, Ting-Chun Wang, Zhiding Yu, Shiqiu Liu, Fitsum A. Reda, Karan Sapra, Andrew Tao, and Bryan Catanzaro. Transposer: Universal texture synthesis using feature maps as transposed convolution filter, 2020.

[94] Morteza Mardani, Guilin Liu, Aysegul Dundar, Shiqiu Liu, Andrew Tao, and Bryan Catanzaro. Neural ffts for universal texture image synthesis. In *NeurIPS '20*, 2020.

[95] Pradyumna Reddy, Michael Gharbi, Michal Lukac, and Niloy J. Mitra. Im2vec: Synthesizing vector graphics without vector supervision, 2021.

[96] Li-Yi Wei, Jianwei Han, Kun Zhou, Hujun Bao, Baining Guo, and Heung-Yeung Shum. Inverse texture synthesis. In *ACM SIGGRAPH 2008 Papers*, SIGGRAPH '08, New York, NY, USA, 2008. Association for Computing Machinery.

[97] Denis Simakov, Yaron Caspi, Eli Shechtman, and Michal Irani. Summarizing visual data using bidirectional similarity. In *CVPR 2008*, pages 1–8, 2008.

[98] Siu Chi Hsu and Irene H. H. Lee. Drawing and animation using skeletal strokes. In *SIGGRAPH '94*, pages 109–118, 1994.

[99] Guojun Gan, Chaoqun Ma, and Jianhong Wu. *Data clustering: theory, algorithms, and applications*. SIAM, 2020.

[100] Li-Yi Wei, Arjun V Anand, Shally Kumar, and Tarun Beri. Simple methods to represent shapes with sample spheres, 2020.

[101] Wojciech Matusik, Matthias Zwicker, and Frédo Durand. Texture design using a simplicial complex of morphable textures. *ACM Trans. Graph.*, 24(3):787–794, July 2005.

[102] Anna Frühstück, Ibraheem Alhashim, and Peter Wonka. Tilegan: synthesis of large-scale non-homogeneous textures. *ACM Transactions on Graphics (ToG)*, 38(4):1–11, 2019.

[103] Ning Yu, Connelly Barnes, Eli Shechtman, Sohrab Amirghodsi, and Michal Lukáč. Texture mixer: A network for controllable synthesis and interpolation of texture. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[104] Rachele Bellini, Yanir Kleiman, and Daniel Cohen-Or. Time-varying weathering in texture space. *ACM Transactions on Graphics (TOG)*, 35(4):1–11, 2016.

[105] Tali Dekel, Tomer Michaeli, Michal Irani, and William T. Freeman. Revealing and modifying non-local variations in a single image. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia)*, 2015.

[106] Stephen Brooks and Neil Dodgson. Self-similarity based texture editing. *ACM Transactions on Graphics (TOG)*, 21(3):653–656, 2002.

[107] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[108] Shervin Minaee, Yuri Boykov, Fatih Porikli, Antonio Plaza, Nasser Kehtarnavaz, and Demetri Terzopoulos. Image segmentation using deep learning: A survey. *IEEE transactions on pattern analysis and machine intelligence*, 44(7):3523–3542, 2021.

[109] Amir Hertz, Or Perel, Raja Giryes, Olga Sorkine-Hornung, and Daniel Cohen-Or. Spaghetti: Editing implicit shapes through part aware generation. *ACM Transactions on Graphics (TOG)*, 41(4):1–20, 2022.

[110] Dave Epstein, Taesung Park, Richard Zhang, Eli Shechtman, and Alexei A Efros. Blobgan: Spatially disentangled scene representations. *ECCV*, 2022.

[111] Wei Zhai, Yang Cao, Zheng-Jun Zha, HaiYong Xie, and Feng Wu. Deep structure-revealed network for texture recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11010–11019, 2020.

[112] Jia Xue, Hang Zhang, and Kristin Dana. Deep texture manifold for ground terrain recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 558–567, 2018.

[113] Hang Zhang, Jia Xue, and Kristin Dana. Deep ten: Texture encoding network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 708–717, 2017.

[114] Xueting Li, Xiaolong Wang, Ming-Hsuan Yang, Alexei A Efros, and Sifei Liu. Scraping textures from natural images for synthesis and editing. In *Computer Vision–ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part XV*, pages 391–408. Springer, 2022.

[115] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. Texture synthesis using convolutional neural networks. In *International Conference on Neural Information Processing Systems*, pages 262–270, 2015.

[116] Urs Bergmann, Nikolay Jetchev, and Roland Vollgraf. Learning texture manifolds with the periodic spatial gan. *arXiv preprint arXiv:1705.06566*, 2017.

[117] Philipp Henzler, Niloy J Mitra, and Tobias Ritschel. Learning a neural 3d texture space from 2d exemplars. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8356–8364, 2020.

[118] Guoqing Hao, Satoshi Iizuka, Kensho Hara, Edgar Simo-Serra, Hirokatsu Kataoka, and Kazuhiro Fukui. Diffusion-based holistic texture rectification and synthesis. *SIGGRAPH Asia Conference Paper*, 2023.

[119] Vivek Kwatra, Arno Schödl, Irfan Essa, Greg Turk, and Aaron Bobick. Graphcut textures: Image and video synthesis using graph cuts. *Acm transactions on graphics (tog)*, 22(3):277–286, 2003.

[120] Sylvain Lefebvre and Hugues Hoppe. Parallel controllable texture synthesis. In *ACM SIGGRAPH 2005 Papers*, pages 777–786. 2005.

[121] Yang Zhou, Kaijian Chen, Rongjun Xiao, and Hui Huang. Neural texture synthesis with guided correspondence. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 18095–18104, 2023.

[122] Yanxi Liu, Wen-Chieh Lin, and James Hays. Near-regular texture analysis and manipulation. *ACM Transactions on Graphics (TOG)*, 23(3):368–376, 2004.

[123] Beichen Li, Liang Shi, and Wojciech Matusik. End-to-end procedural material capture with proxy-free mixed-integer optimization. *ACM Transactions on Graphics (TOG)*, 42(4):1–15, 2023.

[124] Liang Shi, Beichen Li, Miloš Hašan, Kalyan Sunkavalli, Tamy Boubekeur, Radomir Mech, and Wojciech Matusik. Match: Differentiable material graphs for procedural material capture. *ACM Transactions on Graphics (TOG)*, 39(6):1–15, 2020.

[125] Paul Guerrero, Miloš Hašan, Kalyan Sunkavalli, Radomír Měch, Tamy Boubekeur, and Niloy J Mitra. Matformer: A generative model for procedural materials. *arXiv preprint arXiv:2207.01044*, 2022.

[126] Yiwei Hu, Paul Guerrero, Milos Hasan, Holly Rushmeier, and Valentin Deschaintre. Generating procedural materials from text or image prompts. In *ACM SIGGRAPH 2023 Conference Proceedings*, pages 1–11, 2023.

[127] Francesco Locatello, Dirk Weissenborn, Thomas Unterthiner, Aravindh Mahendran, Georg Heigold, Jakob Uszkoreit, Alexey Dosovitskiy, and Thomas Kipf. Object-centric learning with slot attention. *Advances in Neural Information Processing Systems*, 33:11525–11538, 2020.

[128] Jindong Jiang, Fei Deng, Gautam Singh, and Sungjin Ahn. Object-centric slot diffusion. *NeurIPS*, 2023.

[129] Yuting Zhang, Yijie Guo, Yixin Jin, Yijun Luo, Zhiyuan He, and Honglak Lee. Unsupervised discovery of object landmarks as structural representations. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2694–2703, 2018.

[130] Gamaleldin F Elsayed, Aravindh Mahendran, Sjoerd van Steenkiste, Klaus Greff, Michael C Mozer, and Thomas Kipf. Savi++: Towards end-to-end object-centric learning from real-world videos. *arXiv preprint arXiv:2206.07764*, 2022.

[131] Charlie Nash and Christopher KI Williams. The shape variational autoencoder: A deep generative model of part-segmented 3d objects. In *Computer Graphics Forum*, volume 36, pages 1–12. Wiley Online Library, 2017.

[132] Michael Niemeyer and Andreas Geiger. Giraffe: Representing scenes as compositional generative neural feature fields. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11453–11464, 2021.

[133] Mehdi SM Sajjadi, Daniel Duckworth, Aravindh Mahendran, Sjoerd van Steenkiste, Filip Pavetic, Mario Lucic, Leonidas J Guibas, Klaus Greff, and Thomas Kipf. Object scene representation transformer. *Advances in Neural Information Processing Systems*, 35:9512–9524, 2022.

[134] Xingzhe He, Bastian Wandt, and Helge Rhodin. Autolink: Self-supervised learning of human skeletons and object outlines by linking keypoints. *arXiv preprint arXiv:2205.10636*, 2022.

[135] Julian Ost, Fahim Mannan, Nils Thuerey, Julian Knodt, and Felix Heide. Neural scene graphs for dynamic scenes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2856–2865, June 2021.

[136] Xingzhe He, Bastian Wandt, and Helge Rhodin. Ganseg: Learning to segment by unsupervised hierarchical image generation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1225–1235, 2022.

[137] Ce Zheng, Wenhan Wu, Chen Chen, Taojiannan Yang, Sijie Zhu, Ju Shen, Nasser Kehtarnavaz, and Mubarak Shah. Deep learning-based human pose estimation: A survey. *ACM Computing Surveys*, 56(1):1–37, 2023.

[138] Bowen Cheng, Ishan Misra, Alexander G Schwing, Alexander Kirillov, and Rohit Girdhar. Masked-attention mask transformer for universal image segmentation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 1290–1299, 2022.

[139] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. In *International Conference on Learning Representations*, 2016.

[140] Fengting Yang, Qian Sun, Hailin Jin, and Zihan Zhou. Superpixel segmentation with fully convolutional networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 13964–13973, 2020.

[141] Taesung Park, Ming-Yu Liu, Ting-Chun Wang, and Jun-Yan Zhu. Semantic image synthesis with spatially-adaptive normalization. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 2337–2346, 2019.

[142] Richard Zhang, Phillip Isola, Alexei A Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *CVPR*, 2018.

[143] Taesung Park, Jun-Yan Zhu, Oliver Wang, Jingwan Lu, Eli Shechtman, Alexei Efros, and Richard Zhang. Swapping autoencoder for deep image manipulation. *Advances in Neural Information Processing Systems*, 33:7198–7211, 2020.

[144] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers. In *European Conference on Computer Vision*, pages 213–229, 2020.

[145] Meta. Segment anything, 2023.

[146] Guoning Chen, Vivek Kwatra, Li-Yi Wei, Charles D Hansen, and Eugene Zhang. Design of 2d time-varying vector fields. *IEEE Transactions on Visualization and Computer Graphics*, 18(10):1717–1730, 2011.

[147] Nicholas Kolkin, Jason Salavon, and Gregory Shakhnarovich. Style transfer by relaxed optimal transport and self-similarity. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10051–10060, 2019.

[148] Narek Tumanyan, Omer Bar-Tal, Shai Bagon, and Tali Dekel. Splicing vit features for semantic appearance transfer. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10748–10757, 2022.

[149] Bin Cheng, Zuhao Liu, Yunbo Peng, and Yue Lin. General image-to-image translation with one-shot image guidance. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 22736–22746, 2023.

[150] Emmanuel Cornet and Jean-Baptiste Rouquier. Gimp texturize plugin, 2004. `https://lmanul.github.io/gimp-texturize/`.

[151] Li-Yi Wei. Texture synthesis, 2016.

[152] Vivek Kwatra, Arno Schödl, Irfan Essa, Greg Turk, and Aaron Bobick. Graphcut textures: Image and video synthesis using graph cuts. In *SIGGRAPH '03*, pages 277–286, 2003.

[153] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[154] Tero Karras, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. Analyzing and improving the image quality of stylegan. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 8110–8119, 2020.

[155] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1125–1134, 2017.

[156] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[157] William T Freeman, Thouis R Jones, and Egon C Pasztor. Example-based super-resolution. *IEEE Computer graphics and Applications*, 22(2):56–65, 2002.

[158] Zhifei Zhang, Zhaowen Wang, Zhe Lin, and Hairong Qi. Image super-resolution by neural texture transfer. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 7982–7991, 2019.

[159] Haitian Zheng, Mengqi Ji, Haoqian Wang, Yebin Liu, and Lu Fang. Crossnet: An end-to-end reference-based super resolution network using cross-scale warping. In *Proceedings of the European conference on computer vision (ECCV)*, pages 88–104, 2018.

[160] Fuzhi Yang, Huan Yang, Jianlong Fu, Hongtao Lu, and Baining Guo. Learning texture transformer network for image super-resolution. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 5791–5800, 2020.

[161] Jiezhang Cao, Jingyun Liang, Kai Zhang, Yawei Li, Yulun Zhang, Wenguan Wang, and Luc Van Gool. Reference-based image super-resolution with deformable attention transformer. In *European conference on computer vision*, pages 325–342. Springer, 2022.