Continuous Curve Textures

PEIHAN TU, University of Maryland, College Park
LI-YI WEI, Adobe Research
KOJI YATANI and TAKEO IGARASHI, University of Tokyo
MATTHIAS ZWICKER, University of Maryland, College Park

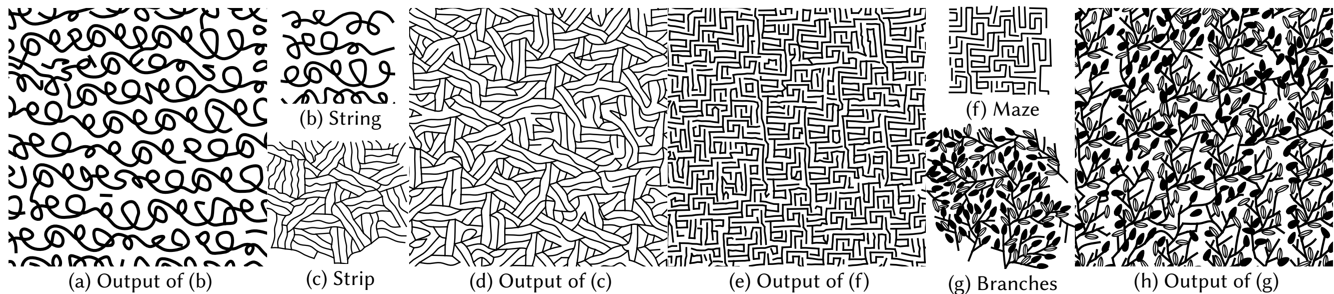


Fig. 1. *Example inputs and outputs of our method.* Given a small input exemplar, our algorithm can synthesize various types of continuous curve textures. Our results are shown in vector format. Please zoom in to see the details.

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. We propose an example-based method to synthesize continuous curve patterns from exemplars. 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.

CCS Concepts: • **Human-centered computing → Interactive systems and tools**; • **Computing methodologies → Texturing**.

Additional Key Words and Phrases: continuous, curve, texture, pattern, synthesis, interface

ACM Reference Format:
Peihan Tu, Li-Yi Wei, Koji Yatani, Takeo Igarashi, and Matthias Zwicker. 2020. Continuous Curve Textures. *ACM Trans. Graph.* 39, 6, Article 168 (December 2020), 16 pages. https://doi.org/10.1145/3414685.3417780

1 INTRODUCTION

Repetitive patterns are fundamental for a variety of tasks in design [Kazi et al. 2012; Lu et al. 2014] and engineering [Chen et al. 2016; Martínez et al. 2015; Schumacher et al. 2016; Zehnder et al. 2016; Zhou et al. 2014]. 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 [Barla et al. 2006; Hsu et al. 2018, 2020; Hurtut et al. 2009; Ijiri et al. 2008; Kazi et al. 2012; Lu et al. 2014; Ma et al. 2013, 2011; Suzuki et al. 2017]. 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.

We propose an example-based method that can automatically synthesize continuous curve patterns from user-supplied exemplars. Similar to prior pixel/sample-based methods [Landes et al. 2013; Lu et al. 2014, 2012; Ma et al. 2013, 2011; Roveri et al. 2015; Wei et al. 2009], 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 1).

Authors' addresses: Peihan Tu, University of Maryland, College Park; Li-Yi Wei, Adobe Research; Koji Yatani; Takeo Igarashi, University of Tokyo; Matthias Zwicker, University of Maryland, College Park.

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *ACM Transactions on Graphics*, https://doi.org/10.1145/3414685.3417780.

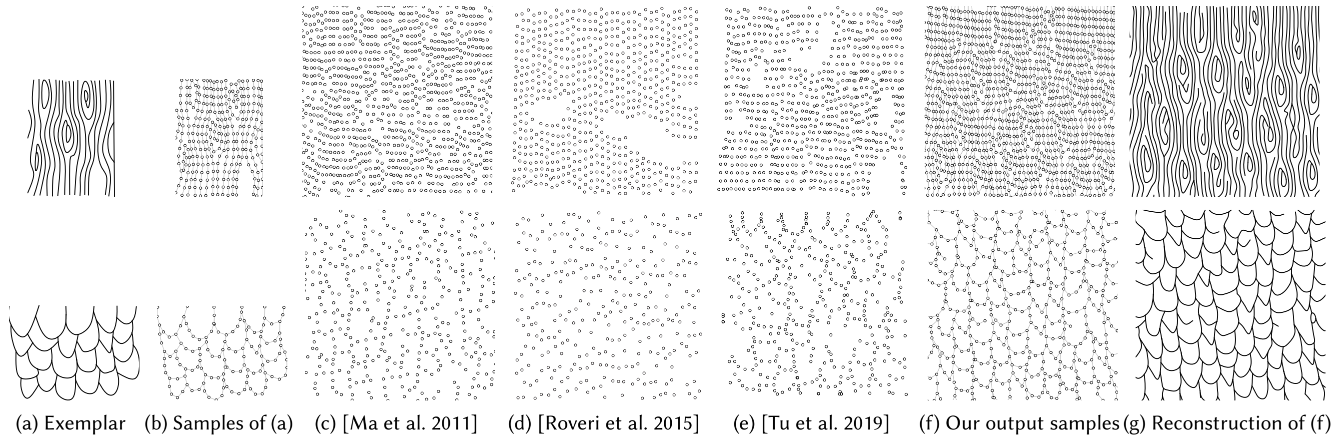


Fig. 2. *Comparison with prior point/sample synthesis algorithms.* The methods in [Ma et al. 2011; Roveri et al. 2015; Tu et al. 2019] generate samples without considering their connections in the exemplars (b). Thus, their results, as shown in (c), (d), and (e), preserve the sample distributions less 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).

Our main idea is to extend prior sample-based element synthesis methods [Hsu et al. 2018; Kazi et al. 2012; Ma et al. 2013, 2011] 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 [Hsu et al. 2018, 2020] that only apply to discrete elements. Since continuous patterns can exhibit higher complexity than discrete elements, we propose robust, hierarchical synthesis [Wei and Levoy 2000, 2001] 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 [Hsu et al. 2020; Xing et al. 2014] 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 [Kazi et al. 2012; Xing et al. 2014]. 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. We plan to share our code along with the publication of this paper to facilitate reproduction.

In sum, the contributions of this work 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 RELATED WORK

Our work is inspired by prior art in vector patterns, image textures, and interactive workflows. Procedural methods [Loi et al. 2017; Pedersen and Singh 2006; Santoni and Pellacini 2016] 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 [Gatys et al. 2016; Lu et al. 2014; Wei et al. 2009] rather than vector patterns. Below, we survey methods most related to our work.

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 [Barla et al. 2006; Bhat et al. 2004; Hsu et al. 2018; Hurtut et al. 2009; Ijiri et al. 2008; Landes et al. 2013; Ma et al. 2013, 2011; Roveri et al. 2015; Tu et al. 2019; Zhou et al. 2007, 2006]. However, these methods target discrete elements or samples [Hsu et al. 2018, 2020; Hurtut et al. 2009; Ijiri et al. 2008; Landes et al. 2013; Ma et al. 2011; Tu et al. 2019] and treat continuous structures as special cases via curve/surface reconstruction from point samples [Ma et al. 2011; Roveri et al. 2015]. Roveri et al. [2015] 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. Tu et al. [2019] 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 [Hertzmann et al. 2002] or growing L-system-like curves [Merrell and Manocha 2010]. Our system is inspired by these prior sample-based algorithms [Hsu et al. 2020; Ma et al. 2011; Roveri

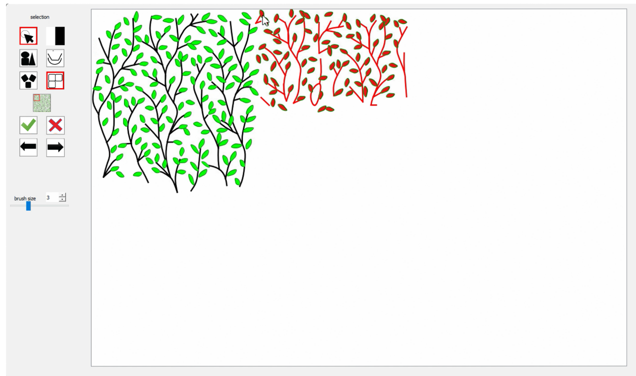
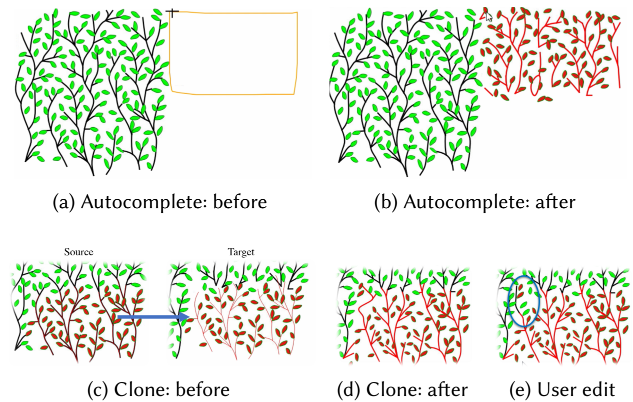


Fig. 3. *User Interface.* Our interface has a widget panel and a canvas. The widget panel provides basic tools, such as selection in a vector editor, as well as controls to the graphics parameters, including color and pen width, and modes unique to our autocomplete system.



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

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

Fig. 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.

et al. 2015], 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 Interactive Authoring

Workflow analysis has been investigated to assist various content creation task [Nancel and Cockburn 2014]. Examples includes static and animated sketches [Xing et al. 2014, 2015], 3D sculpting [Peng et al. 2020, 2018], texture design [Suzuki et al. 2017], hand-writing beautification [Zitnick 2013], and image editing [Chen et al. 2011; Koyama et al. 2016]. Our work is inspired by prior autocomplete [Peng et al. 2018; Suzuki et al. 2017; Xing et al. 2014, 2015] and interactive systems [Bian et al. 2018; Hsu et al. 2020; Kazi et al. 2012]. 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.

3 USER INTERFACE

Our system can be used for both automatic synthesis and interactive editing. Similar to prior work like traditional texture synthesis [Wei et al. 2009], 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 4a) and let our system predict patterns that resemble what the users have already drawn (*autocomplete* mode, Figures 4a and 4b). The users can also explicitly control the prediction by copying-pasting from an input region to an output region (*clone* mode, Figures 4c and 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 for live actions. Since continuous patterns often contain complex structures beyond fine-grained autocomplete of individual strokes [Xing et al. 2014], we design our current interface to focus on autocomplete pattern regions instead of strokes.

4 METHOD

Our method extends the sample-based element texture synthesis method in [Ma et al. 2011] to consider not only individual point samples but also their curved connections via graphs [Hsu et al. 2018, 2020]. We describe our pattern representation in Section 4.1, similarity measures in Section 4.2, and the corresponding synthesis and reconstruction algorithms in Sections 4.3 and 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.

4.1 Representation

We represent patterns and elements via point samples and graphs (Figure 5) [Hsu et al. 2018; Ma et al. 2011; Roveri et al. 2015]. Each sample s records its position $\mathbf{p}(s)$, attributes $\mathbf{a}(s)$ (Table 1), and $i(s) \in [0, 1]$ to indicate the confidence of its existence to optimize the number of samples:

$$\mathbf{u}(s) = (\mathbf{p}(s), \mathbf{a}(s), i(s))\,. \tag{1}$$

Discrete elements. Following [Ma et al. 2013, 2011], for discrete elements, sample attributes include a sample id $q(s)$ that indicates the uniqueness of s to other samples within its containing element.

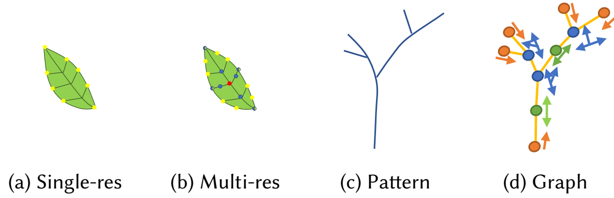


Fig. 5. *Pattern representation.* Existing algorithms [Ma et al. 2011] only use a single-resolution element representation (a). We propose a hierarchical element representation to improve the synthesis quality (b). Our synthesis proceeds from red, blue, to yellow samples in coarse to fine levels. A pattern (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 $|\mathcal{E}(s)|$ associated with it; orange, green, and blue indicate 1, 2, and 3.

Table 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. $\mathcal{E}(s) = \{e_{ss'}\}$ 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
$\mathbf{a}(s)$	$q(s)$	$\mathcal{E}(s) = \{e_{ss'}\}$	$\mathbf{o}(s)$

Continuous structures. In discrete element synthesis, it is sufficient to use only samples with id $q(s)$ (Figures 5a and 5b) to encode element shape because every element has the same topology [Ma et al. 2011]. On the other hand, continuous structures are composed of paths and have more flexibilities. In our paper, 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 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 5d). Note the method in [Hsu et al. 2018, 2020] 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 $\mathcal{E}(s)$ for s within $\mathbf{a}(s)$. $\mathcal{E}(s) = \{e_{ss'}\}$ 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 $\mathcal{E}(s)$ 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 4.4.2 and fig. 15). Note that, for any input samples s_i, we always have $|\mathcal{E}(s_i)| = N_\mathbf{o}(s_i)$ (Figure 5d), where $|\mathcal{E}|$ is the size of the edge set $\mathcal{E}$. However, this strict constraint is relaxed for the output during the synthesis to facilitate faster convergence.

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 [Wei and Levoy 2000, 2001]. 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 5a and 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 5c), we sample the intersections (blue samples in Figure 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 5.1.

4.2 Similarity Measure

A core part of pattern synthesis is a measure of similarity between local regions [Ma et al. 2011; Roveri et al. 2015; Wei et al. 2009]. Here, we describe our similarity measure for continuous patterns via their sample-graph representation (Section 4.1), which, in turn, will form the basis for our synthesis optimization (Section 4.3).

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:

$$\hat{\mathbf{p}}(s, s') = \mathbf{p}(s) - \mathbf{p}(s'), \tag{2}$$

$$\hat{\mathbf{a}}(s, s') = \left(\hat{q}(s, s'), \hat{\mathcal{E}}(s, s')\right) \tag{3}$$

The differences in sample id attribute q is

$$\hat{q}(s, s') = \mathbb{1}\left\{q(s) \neq q(s')\right\}, \tag{4}$$

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

$$\hat{\mathcal{E}}(s, s') = \left(\sum_{e_{s\hat{s}} \in \mathcal{E}(s)} dist\left(e_{s\hat{s}}, e_{s'\hat{s}'}\right)\right) + \beta \left||\mathcal{E}(s)| - |\mathcal{E}(s')|\right|, \tag{5}$$

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 $e_{s'\hat{s}'} = m_e(e_{s\hat{s}}) \in \mathcal{E}(s')$ is the matching edge for $e_{s\hat{s}}$ via the Hungarian algorithm (which solves the one-to-one matching relationship between edges) [Kuhn 1955] to minimize the first term in Equation (5) (m indicates matching relationship). β is a weighting parameter set to sampling distance δ (Section 4.1.1) in our experiments.

For newly added output samples that do not have any edges, either by initialization or existence assignment (Section 4.3), 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 4.3.3). We set

$\beta = 0$ for these samples and thus Equation (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 (3)) since $\mathcal{E}$ already contain similar information in $\mathbf{o}$. However, we still need to update $\mathbf{o}$ during synthesis (assignment step, Section 4.3.4) and reconstruction (Section 4.4). This requires us to match orientation entries o within $\mathbf{o}$ from s and s' respectively. The matching $o(s') = m_o(o(s)) \in \mathbf{o}(s')$ is computed via the Hungarian algorithm [Kuhn 1955] by minimizing the sum of smallest absolute differences between matched $o(s)$ and $o(s')$

$$\hat{\mathbf{o}}(s, s') = \sum_{o(s) \in \mathbf{o}(s)} \hat{o}(s, s'), \tag{6}$$

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

We also have not found it necessary to include i in the sample similarity measure (Equation (3)). Instead, i will be used to optimize the number of samples.

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

$$\|\mathbf{n}(s_o) - \mathbf{n}(s_i)\| = \sum_{s'_o \in m_n(\mathbf{n}(s_i))} \hat{\mathbf{u}}_{s_o s_i}(s'_o, s'_i) + \sum_{s'_o \in \mathbf{n}(s_o) \ominus m_n} c(s'_o), \tag{7}$$

where

$$\hat{\mathbf{u}}_{s_o s_i}(s'_o, s'_i) = \|\hat{\mathbf{p}}(s_o, s'_o) - \hat{\mathbf{p}}(s_i, s'_i)\| + \gamma \|\hat{\mathbf{a}}(s'_o, s'_i)\| \tag{8}$$

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 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 (7) partition $\mathbf{n}(s_o)$ into two sets. In the first term of Equation (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 (7), $c(s'_o)$ is the cost resulting from unmatched output samples s'_o. In our implementation, $\gamma = 0.5$. Equation (7) is designed for our robust neighborhood matching, described next.

4.2.3 Robust Neighborhood Matching. In [Ma et al. 2011], 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 4.2.1, since we have one-to-one sample correspondence. We call this *hard neighborhood matching* (Figure 6a). In [Roveri et al. 2015], 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 6b). The method [Roveri et al. 2015] "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.

Instead, we propose to use a robust neighborhood matching that explicitly considers outliers in the output to address these issues (Figure 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 [Riesen and Bunke 2009], we define our cost matrix $C \in \mathbb{R}^{N_{\mathbf{n}o} \times (N_{\mathbf{n}i} + N_{\mathbf{n}o})}$ as:

$$C = \begin{bmatrix} C_m & | & C_u \end{bmatrix} \tag{9}$$

$$C_m = \begin{bmatrix} \hat{\mathbf{u}}_{s_o s_i}(s_o'^1, s_i'^1) & \hat{\mathbf{u}}_{s_o s_i}(s_o'^1, s_i'^2) & \cdots & \hat{\mathbf{u}}_{s_o s_i}(s_o'^1, s_i'^{N_{\mathbf{n}i}}) \\ \hat{\mathbf{u}}_{s_o s_i}(s_o'^2, s_i'^1) & \hat{\mathbf{u}}_{s_o s_i}(s_o'^2, s_i'^2) & \cdots & \hat{\mathbf{u}}_{s_o s_i}(s_o'^2, s_i'^{N_{\mathbf{n}i}}) \\ \vdots & \vdots & \ddots & \vdots \\ \hat{\mathbf{u}}_{s_o s_i}(s_o'^{N_{\mathbf{n}o}}, s_i'^1) & \hat{\mathbf{u}}_{s_o s_i}(s_o'^{N_{\mathbf{n}o}}, s_i'^2) & \cdots & \hat{\mathbf{u}}_{s_o s_i}(s_o'^{N_{\mathbf{n}o}}, s_i'^{N_{\mathbf{n}i}}) \end{bmatrix} \tag{10}$$

$$C_u = \begin{bmatrix} c_1 & c_1 & \cdots & c_1 \\ c_2 & c_2 & \cdots & c_2 \\ \vdots & \vdots & \vdots & \vdots \\ c_{N_{\mathbf{n}o}} & c_{N_{\mathbf{n}o}} & \cdots & c_{N_{\mathbf{n}o}} \end{bmatrix}, \tag{11}$$

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 [Ma et al. 2013], the sample matching is computed via only the C_m part of C, in which case every output sample should be matched. Our cost matrix is augmented with the 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 $\min(2, 1.2 + 0.4|\mathcal{E}(s_o^k)|)\delta$ if the output sample s_o^k is from a continuous pattern, and 1.5× 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.

4.3 Pattern Synthesis

Based on our pattern representation (Section 4.1) and similarity measures (Section 4.2), we now describe how to synthesize an output similar to a given input.

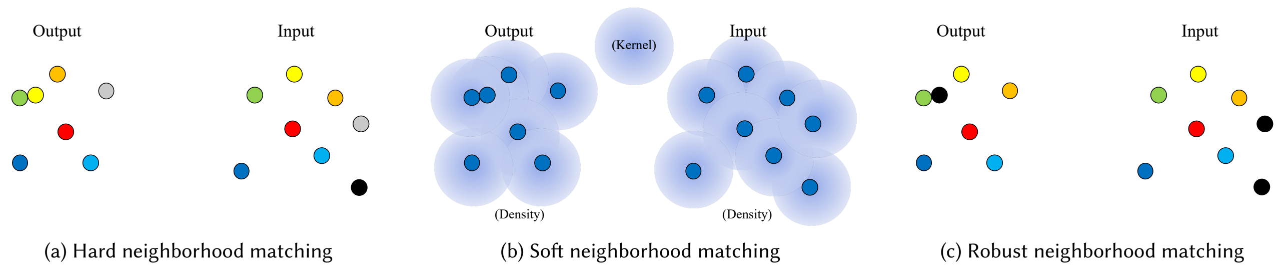


Fig. 6. *Different neighborhood matching methods.* In (a) each sample in the output (left) is forced to match with another one in the input (right), which could be problematic since some output samples are outliers and should not be matched to any input samples, and some output samples might be missing 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 [Roveri et al. 2015]. 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.

4.3.1 Optimization Objective. We synthesize output predictions O via optimizing the following objective:

$$E(O) = \sum_{s_i = m(s_o), s_o \in O} \|\mathbf{n}(s_o) - \mathbf{n}(s_i)\| + \Theta(O, \mathcal{D}). \qquad (12)$$

where s_o is matched with s_i. This energy sums up the similarity between every $\mathbf{n}(s_o)$ in O and its most similar $\mathbf{n}(s_i)$ via Equation (7). $\Theta(O, \mathcal{D})$ is the domain constraint term [Dumas et al. 2018] 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 (12), by iterating the search and assignment steps as detailed below.

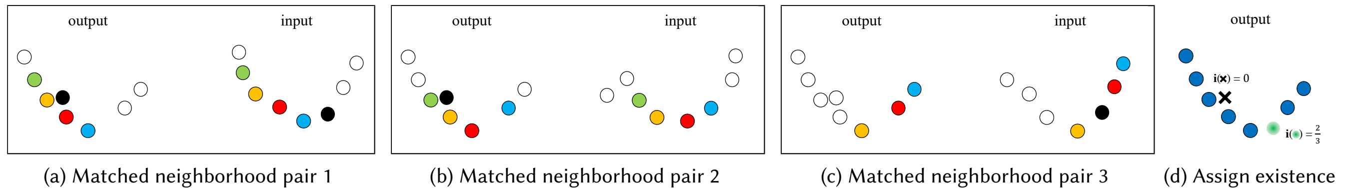
4.3.2 Initialization. Similar to prior patch-based texture synthesis methods [Efros and Freeman 2001; Liang et al. 2001], we copy new patches one-by-one with similar boundary patterns to existing patches for initialization. Each next patch is selected to ensure high similarity (as evaluated by Equation (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 [Ma et al. 2011]. In Section 5, we will show the robustness of our method to random sample initializations (Figure 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.

4.3.3 Search Step. We adopt PatchMatch [Barnes et al. 2009; Chen et al. 2012] 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 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 [Chen et al. 2012] 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 each level of hierarchical synthesis (Section 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 $O(n_O N_{\mathbf{n}}^3)$, where n_O is the total number of output samples and $N_{\mathbf{n}}$ is the average number of samples within neighborhoods. Please refer to Appendix C for more details.

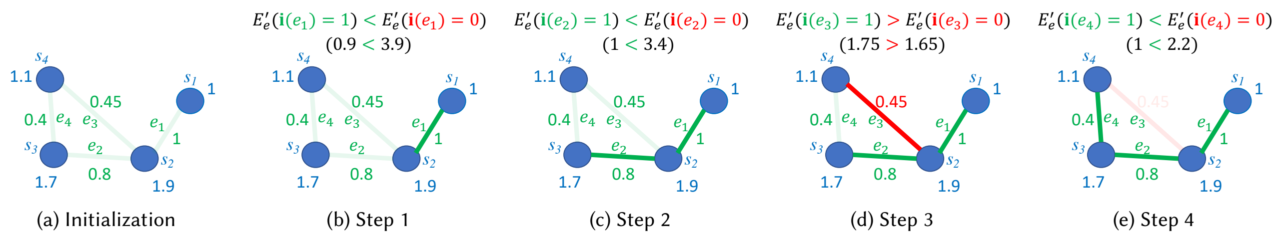
4.3.4 Assignment Step. Here, we describe how to determine the values of sample positions $\mathbf{p}$, attributes including edge $\boldsymbol{\mathcal{E}}$ and orientation $\mathbf{o}$, as well as sample existence i. The assignments of these different quantities are extended from the assignment step of pixel colors [Kwatra et al. 2005] and sample positions [Ma et al. 2011] 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 $\left(s_i, s'_i\right)$ provided by the previous search step. The estimated distance $\hat{\mathbf{p}}(s_o, s'_o)$ between



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

Fig. 7. *Existence assignment example.* We visualize how to compute confidences of existence of output samples $i(s_o)$ (Equation (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).



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

Fig. 8. *Edge assignment example.* This example illustrates how to solve Equation (16) for edge assignment by solving Equation (21) in a loop. Each step (b)-(e) solves Equation (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 $|\bar{\mathcal{E}}(s)|$ associated with samples (e.g. $\bar{i}(e_2) = 0.8$, $|\bar{\mathcal{E}}(s_2)| = 1.9$). 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. $E'_e\,(i(e_1) = 1) = |i(e_1) - \bar{i}(e_1)| + |\mathcal{E}(s_1) - \bar{\mathcal{E}}(s_1)| + |\mathcal{E}(s_2) - \bar{\mathcal{E}}(s_2)| = |1-1| + |1-1| + |1-1.9| = 0.9$). 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.

s_o and s'_o is

$$\hat{\mathbf{p}}(s_o, s'_o) \approx \mathbf{p}(s_i) - \mathbf{p}(s'_i). \tag{13}$$

We use least squares [Ma et al. 2011] to estimate $\mathbf{p}(s_o)$ by

$$\underset{\{\mathbf{p}(s_o)\}}{\arg\min} \sum_{s_o \in O} \sum_{s'_o \in \mathbf{n}(s_o)} \left\|\hat{\mathbf{p}}(s_o, s'_o) - \left(\mathbf{p}(s_i) - \mathbf{p}(s'_i)\right)\right\|^2 + \sum_{s_o \notin \mathcal{D}} \|\hat{\mathbf{p}}(s_o, \mathcal{D})\|^2. \tag{14}$$

The second term in Equation (14) 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:

$$\underset{i(s_o) \in [0\ 1]}{\arg\min} \sum_{s_i \in \{s_i\}} |i(s_o) - i(s_i)|^2\,, \tag{15}$$

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 (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 (15) is not guaranteed to decrease immediately after sample addition or removal, but it will generally decrease through iterations. Please refer to Figure 7 for an example and Appendix A for more algorithm details.

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

$$\underset{\{i(e_{s_o s'_o}) \in \{0,1\}\}}{\arg\min} \sum_{\{e_{s_o s'_o}\}} \left|i(e_{s_o s'_o}) - \bar{i}(e_{s_o s'_o})\right| + \sum_{\{s_o\}} \left|\left|\mathcal{E}(s_o)\right| - \left|\bar{\mathcal{E}}(s_o)\right|\right|, \tag{16}$$

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 (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 $|\mathcal{E}(s_o)|$ and the expected number of edges $|\overline{\mathcal{E}}(s_o)|$ connected to s_o. $|\overline{\mathcal{E}}(s_o)|$ is similarly computed by voting from overlapping output neighborhoods on the same sample s_o:

$$\underset{|\overline{\mathcal{E}}(s_o)|}{\arg\min} \sum_{s_i \in \{s_i\}} \left| |\overline{\mathcal{E}}(s_o)| - |\mathcal{E}(s_i)| \right|^2. \tag{17}$$

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

It is non-trivial to optimize Equation (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 (16). In other words, the multivariate optimization problem (Equation (16)) is optimized by solving univariate optimization problems in a loop. By decomposing the optimization variables in Equation (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 (16) can be written as:

$$\underset{i(e_{s^*_o s^{*\prime}_o}) \in \{0,1\}}{\arg\min} \quad E'_e + E''_e, \tag{18}$$

where

$$E'_e\left(i(e_{s^*_o s^{*\prime}_o})\right) = \left|i(e_{s^*_o s^{*\prime}_o}) - \bar{i}(e_{s^*_o s^{*\prime}_o})\right| + \left| |\mathcal{E}(s^*_o)| - |\overline{\mathcal{E}}(s^*_o)| \right| + \left| |\mathcal{E}(s^{*\prime}_o)| - |\overline{\mathcal{E}}(s^{*\prime}_o)| \right|, \tag{19}$$

$$E''_e\left(i(e_{s^*_o s^{*\prime}_o})\right) = \sum_{\{e_{s_o s'_o}\} \ominus e_{s^*_o s^{*\prime}_o}} \left|i(e_{s_o s'_o}) - \bar{i}(e_{s_o s'_o})\right| + \sum_{\{s_o\} \ominus \{s^*_o, s^{*\prime}_o\}} \left| |\mathcal{E}(s_o)| - |\overline{\mathcal{E}}(s_o)| \right|. \tag{20}$$

Since E''_e is a constant in Equation (18), it is equivalent to:

$$\underset{i(e_{s^*_o s^{*\prime}_o}) \in \{0,1\}}{\arg\min} \quad E'_e. \tag{21}$$

Equation (21) can be solved with brute-force search. The search space is 2 ($\{0, 1\}$). Figure 8 illustrates how to solve Equation (16) by solving Equation (21) in a loop.

Orientation assignment. In the search step (Section 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

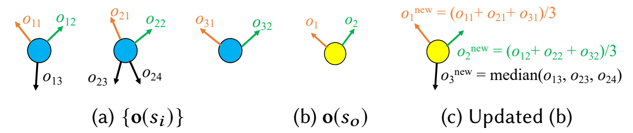


Fig. 9. *Orientation assignment example.* This example illustrates the orientation assignment step when $N_\mathbf{o}(s_o)$ is increased from 2 to 3. The output sample (b) is matched with the three input samples (a). Matched orientations o (arrows) are visualized in the same colors. Unmatched inputs 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).

$|\mathcal{E}(s_i)|$. Thus $N_\mathbf{o}$ can be computed like in Equation (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 (15) and (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 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.

4.3.5 Hierarchical Synthesis. Instead of using a single-resolution representation [Ma et al. 2011], we apply a hierarchical representation (Section 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 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 10 for an example.

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.

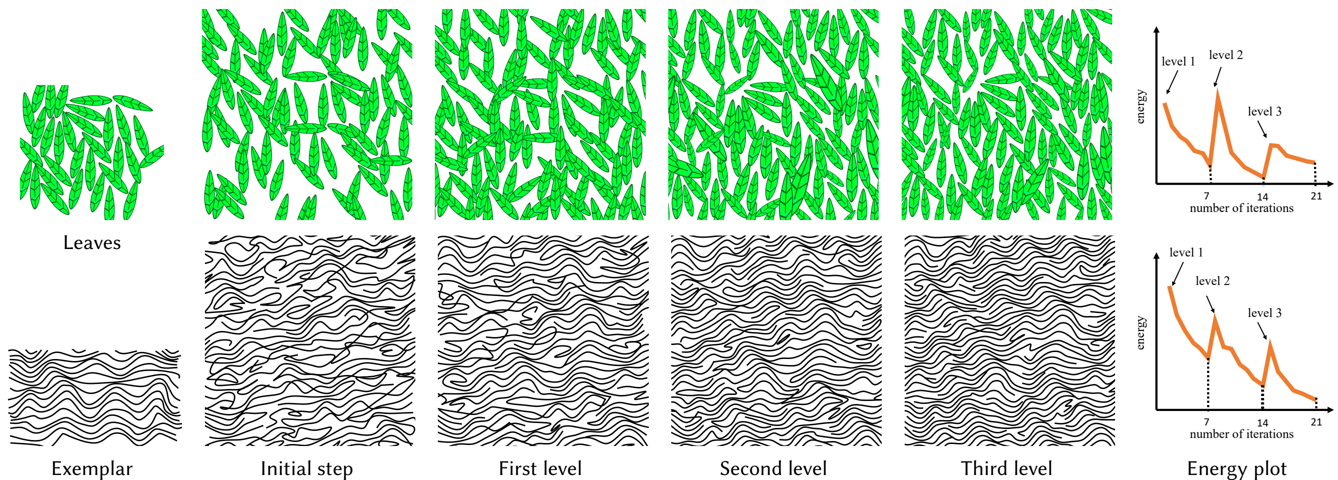


Fig. 10. *Hierarchical synthesis visualization.* The hierarchical synthesis proceeds from the coarsest (first level) to the finest levels (third level), with upsampling sample representations (Figure 5b). The hierarchical synthesis gradually refines the pattern from large to small scale structures. Note that the initialization has fewer elements than required (top row), hence our existence optimization adaptively controls the total number of elements.

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.

4.4.2 Continuous Structures. We have synthesized a graph whose sample positions and edge connections represent the topology of the output (Section 4.3). 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 11) as path samples, or two paths as junction samples (e.g. the yellow sample in Figure 11a). Samples with more than two neighbors are junction samples (e.g. the red sample in Figure 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 11a and 11d, or the orange and green ones associated with the red junction sample in Figure 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 11a and 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 15.

5 EVALUATION

We evaluate our method with sample results, ablation studies, and comparisons with existing art. We will make our code repository [Tu 2020] public to facilitate future research.

5.1 Results

Our method can automatically synthesize satisfactory results for a variety of patterns without user intervention, as exemplified in Figures 1, 12 and 13 and our (full) results in Figures 2, 10 and 14 to 16. 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 1a and Figure 2g or inconsistent curvatures in Figure 10k bottom) and global structures (such as the regular and warped grids in Figures 12h and 12p, the rectangular blocks in Figure 12l, and the straight lines in Figures 1e and 12t). For further quality improvement and customization, users can also interactively edit the system suggestions via our system interface, as demonstrated in Figure 13. Unless otherwise noted, all our results 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 Appendix B for more details about our parameter settings.

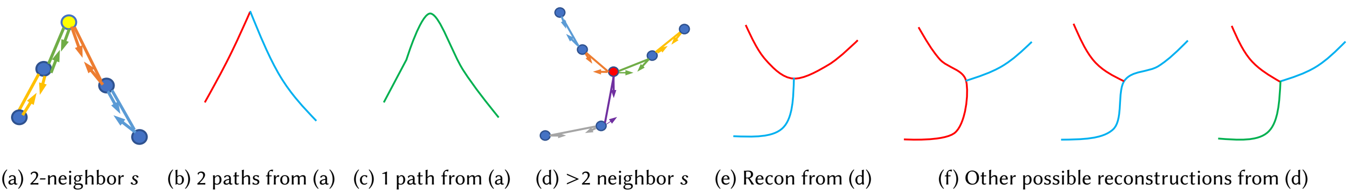


Fig. 11. *Curve reconstruction from a graph using orientation attributes.* If we only consider the different statuses of the yellow sample in (a), there are two possible reconstructions (b)(c), depending on whether it is a junction (b) or path (c) sample. If we only consider the different statuses of 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).

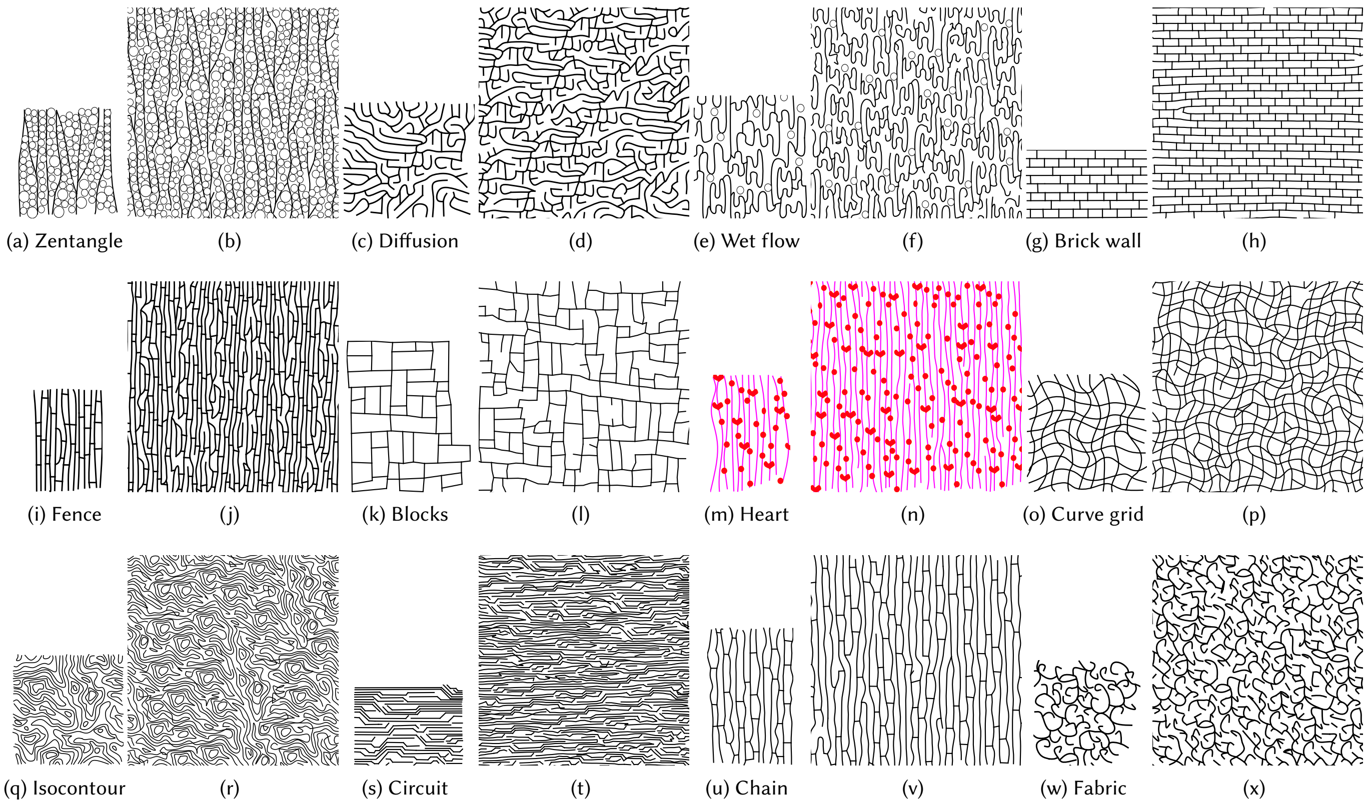


Fig. 12. *Automatic synthesis results by our method.* Within each pair of images, the input exemplar is smaller and shown on the left, the automatic synthesis result is bigger and shown on the right.

5.2 Ablation Study

Although we use patch-based methods for initialization in our implementation, our algorithm is robust to different initial conditions (Figure 14), even if the initial sample distribution is randomly distributed (white noise). Figure 15 is the ablation study for the orientation attribute **o**. Figure 16 shows other components of our algorithm. Without the edge term (Equation (5)) or robust matching (Section 4.2.3) in the search step, our algorithm produces lower quality results with obvious artifacts. Without existence assignment (the third paragraph in Section 4.3.4), the algorithm cannot automatically adjust the number of samples within local regions and can produce empty space or extra broken curves.

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

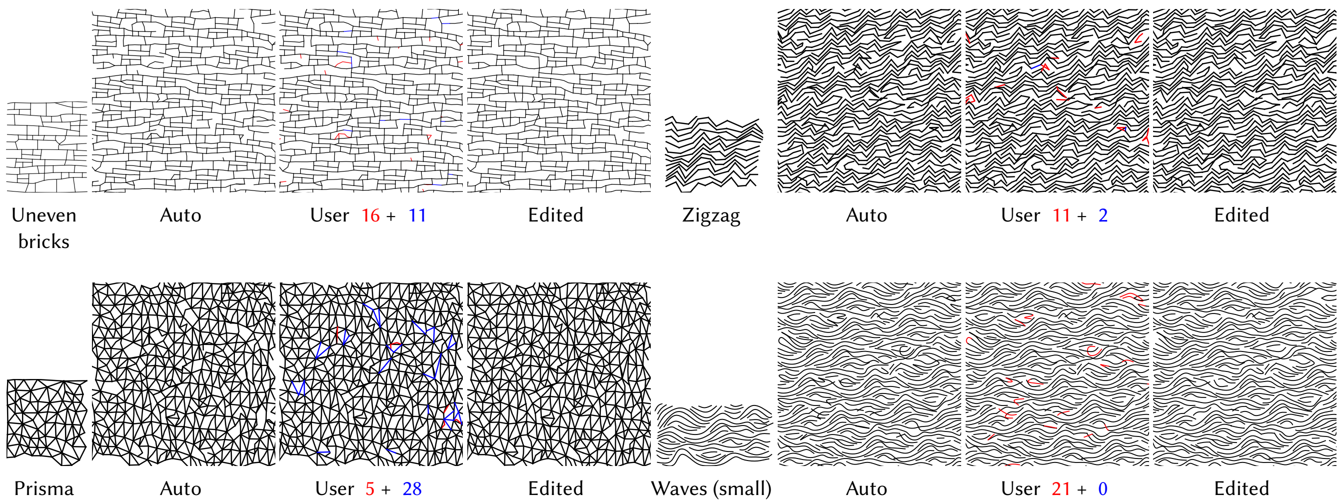


Fig. 13. *Automatic synthesis and user-assisted results.* We count the number of user operations needed for correcting artifacts. Red indicates the number of rejection and blue the number of manual path drawing.

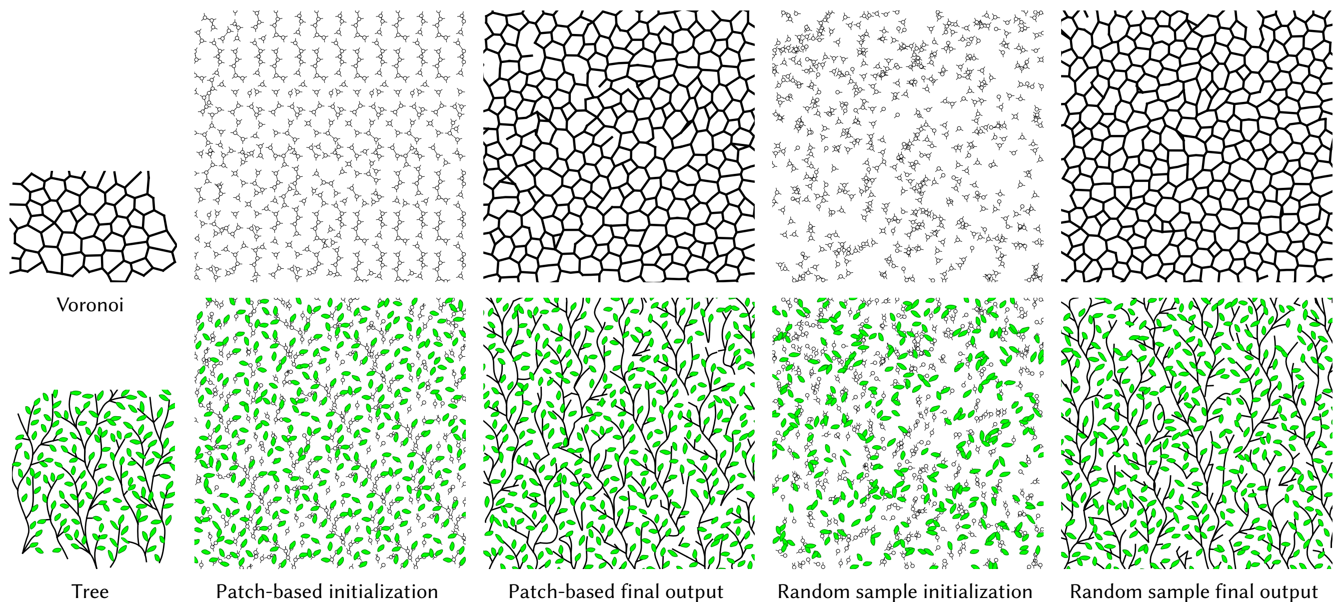


Fig. 14. *Algorithm robustness to different initialization.* Our algorithm can generate similar results with both patch-based or random initialization.

methods in [Ma et al. 2011; Roveri et al. 2015; Tu et al. 2019] are the most related. We compare against [Ma et al. 2011; Roveri et al. 2015] and a state-of-art point distribution synthesis method in [Tu et al. 2019] which applies convolutional neural networks to preserve both local and global structures. As shown in Figure 2, our method can produce better spatial sample distributions than [Ma et al. 2011; Roveri et al. 2015; Tu et al. 2019]. Note that we compare only sample distributions in Figure 2 since it is unclear how to reconstruct continuous curve patterns from synthesized samples without connectivity [Ma et al. 2011; Roveri et al. 2015; Tu et al. 2019].

We also *enhance* [Ma et al. 2011] for comparisons, by incorporating it with the sample connectivity (Figure 5d) and edge assignment step (Equation (16)), but without the edge set difference (Equation (5)) and robust matching (Section 4.2.3) in the search step, as well as without the existence assignment step (the third paragraph in Section 4.3.4). As shown in Figure 17, our method

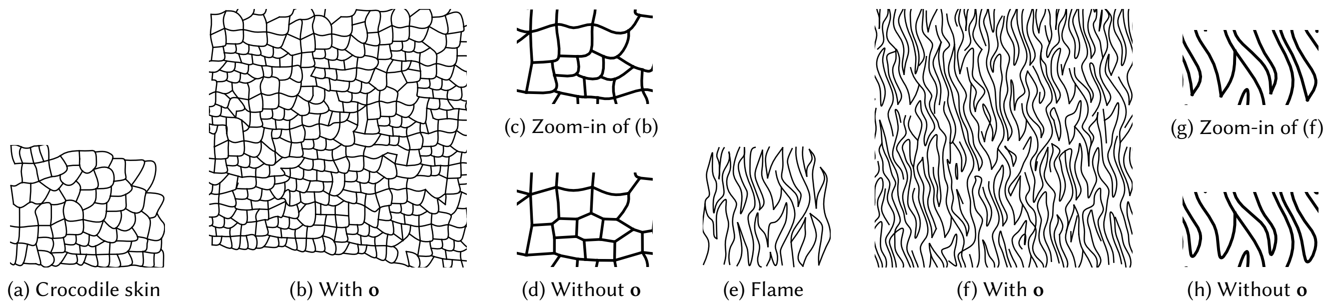


Fig. 15. *Ablation study for* **o**. The orientation attribute **o** is useful to faithfully recover curve appearance in the exemplars. In the "crocodile skin" example, the curves should be smooth at junction; in the "flame" example, the curves should be sharp at the flame tip.

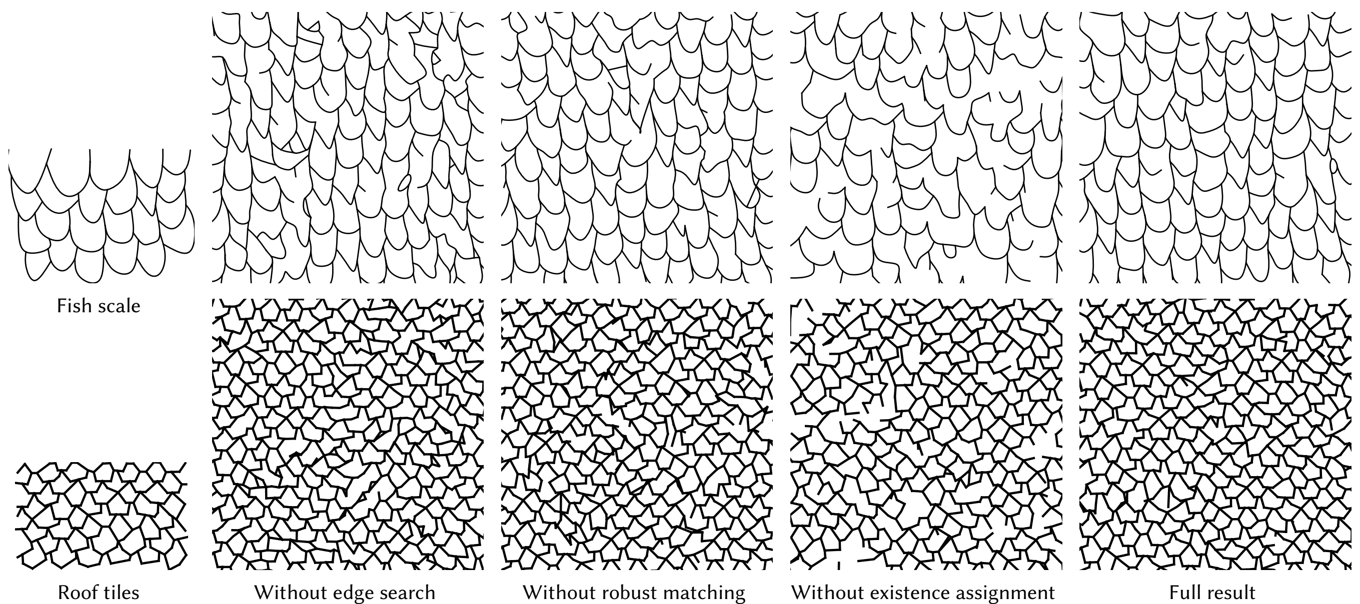


Fig. 16. *Ablation study.* Without using edges (Equation (5)) in the search step (the second column) or robust matching (Section 4.2.3) that considers outliers (the third column), the algorithm produces lower quality results. Without existence assignment (the third paragraph in Section 4.3.4), the algorithm produces broken curves and empty space due to outliers and missing samples (the forth column). Our results are shown in the last column.

can generate better results than the enhanced version of [Ma et al. 2011]. Unlike for [Ma et al. 2011], we are unable to enhance [Roveri et al. 2015; Tu et al. 2019] due to the lack of one-to-one sample correspondences which are needed for the edge assignment step.

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 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 5.1. These artifacts can be reduced by other improvements, such as bidirectional similarity, additional feature masks, and smart initialization [Kaspar et al. 2015].

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 [Hsu et al. 2020; Ma et al. 2011], while ensuring that curve segments sharing common samples are well connected.

Our current algorithm treats discrete elements and continuous structures separately and thus might not preserve identifiable elements within continuous structures, as exemplified in Figure 18.

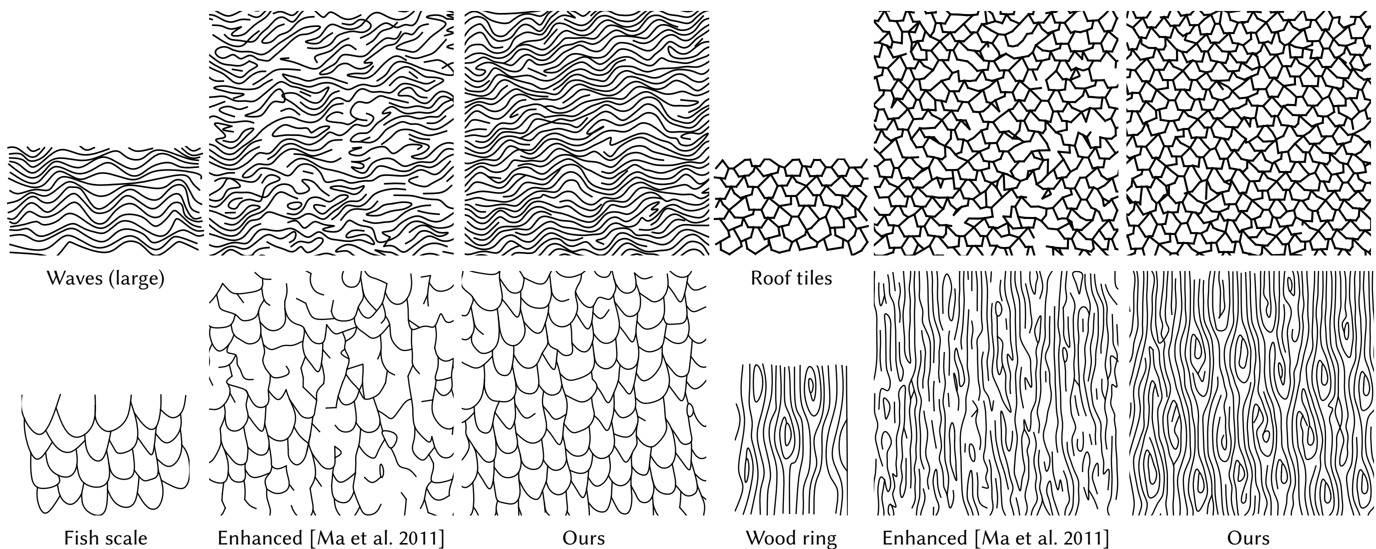


Fig. 17. *Comparison of our algorithm with enhanced [Ma et al. 2011].* We compare our methods to an enhanced version of [Ma et al. 2011] that incorporates our ideas, including the sample connectivity and edge assignment.

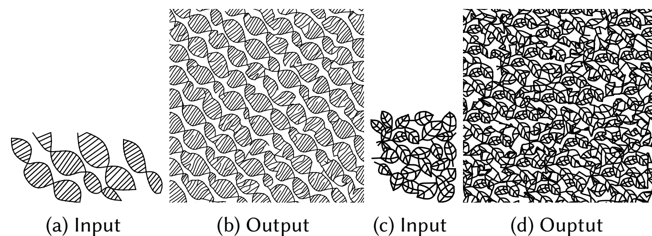


Fig. 18. *Failure case.* Our algorithm fails to preserve the identifiable DNA-segment and tree-leaf elements within the continuous structures.

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 [Huang et al. 2007].

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 [Takayama et al. 2010; Wang et al. 2011, 2010]. More controls can also be added to facilitate more diverse authoring effects such as local variations in scales and orientations [Hsu et al. 2020]. In addition to optimizing pattern appearance as in this work, adding mechanical structures constraints can facilitate the application of curve structures for rapid manufacturing [Bian et al. 2018; Chen et al. 2017, 2016; Li et al. 2019; Zehnder et al. 2016; Zhou et al. 2014].

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback. Peihan Tu conducted parts of this research as a visiting student with the University of Tokyo and an intern with Adobe Research. This work has been partially supported by an Adobe gift funding and JSPS KAKENHI Grant Number 17H00752.

REFERENCES

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

Connelly Barnes, Eli Shechtman, Adam Finkelstein, and Dan B Goldman. 2009. PatchMatch: A Randomized Correspondence Algorithm for Structural Image Editing. *ACM Trans. Graph.* 28, 3, Article 24 (July 2009), 11 pages. https://doi.org/10.1145/1531326.1531330

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

Xiaojun Bian, Li-Yi Wei, and Sylvain Lefebvre. 2018. Tile-based Pattern Design with Topology Control. *Proc. ACM Comput. Graph. Interact. Tech.* 1, 1, Article 23 (July 2018), 15 pages. https://doi.org/10.1145/3203204

Hsiang-Ting Chen, Li-Yi Wei, and Chun-Fa Chang. 2011. Nonlinear Revision Control for Images. *ACM Trans. Graph.* 30, 4, Article 105 (July 2011), 10 pages. https://doi.org/10.1145/2010324.1965000

Weikai Chen, Yuexin Ma, Sylvain Lefebvre, Shiqing Xin, Jonàs Martínez, and wenping wang. 2017. Fabricable Tile Decors. *ACM Trans. Graph.* 36, 6, Article 175 (Nov. 2017), 15 pages. https://doi.org/10.1145/3130800.3130817

Weikai Chen, Xiaolong Zhang, Shiqing Xin, Yang Xia, Sylvain Lefebvre, and Wenping Wang. 2016. Synthesis of Filigrees for Digital Fabrication. *ACM Trans. Graph.* 35, 4, Article 98 (July 2016), 13 pages. https://doi.org/10.1145/2897824.2925911

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

Emmanuel Cornet and Jean-Baptiste Rouquier. 2004. GIMP Texturize plugin. https://lmanul.github.io/gimp-texturize/.

Jérémie Dumas, Jonàs Martínez, Sylvain Lefebvre, and Li-Yi Wei. 2018. Printable Aggregate Elements. *arXiv preprint arXiv:1811.02626* (2018).

Alexei A. Efros and William T. Freeman. 2001. Image Quilting for Texture Synthesis and Transfer. In *SIGGRAPH '01*. 341–346. https://doi.org/10.1145/383259.383296

Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. 2016. Image Style Transfer Using Convolutional Neural Networks. In *CVPR '16*. 2414–2423.

Aaron Hertzmann, Nuria Oliver, Brian Curless, and Steven M. Seitz. 2002. Curve Analogies. In *EGRW '02*. 233–246.

Chen-Yuan Hsu, Li-Yi Wei, Lihua You, and Jian Jun Zhang. 2018. Brushing Element Fields. In *SIGGRAPH Asia 2018 Technical Briefs (SA '18)*. Article 6, 4 pages. https://doi.org/10.1145/3283254.3283274

Chen-Yuan Hsu, Li-Yi Wei, Lihua You, and Jian Jun Zhang. 2020. Autocomplete Element Fields. In *CHI '20*. 1–13. https://doi.org/10.1145/3313831.3376248

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

T. Hurtut, P.-E. Landes, J. Thollot, Y. Gousseau, R. Drouillhet, and J.-F. Coeurjolly. 2009. Appearance-guided Synthesis of Element Arrangements by Example. In *NPAR '09*. 51–60. https://doi.org/10.1145/1572614.1572623

Takashi Ijiri, Radomír Mech, Takeo Igarashi, and Gavin Miller. 2008. An Example-based Procedural System for Element Arrangement. In *Computer Graphics Forum*, Vol. 27. Wiley Online Library, 429–436.

Alexandre Kaspar, Boris Neubert, Dani Lischinski, Mark Pauly, and Johannes Kopf. 2015. Self Tuning Texture Optimization. *Comput. Graph. Forum* 34, 2 (May 2015), 349–359. https://doi.org/10.1111/cgf.12565

Rubaiat Habib Kazi, Takeo Igarashi, Shengdong Zhao, and Richard Davis. 2012. Vignette: Interactive Texture Design and Manipulation with Freeform Gestures for Pen-and-ink Illustration. In *CHI '12*. 1727–1736. https://doi.org/10.1145/2207676.2208302

Yuki Koyama, Daisuke Sakamoto, and Takeo Igarashi. 2016. SelPh: Progressive Learning and Support of Manual Photo Color Enhancement. In *CHI '16*. 2520–2532. https://doi.org/10.1145/2858036.2858111

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

Vivek Kwatra, Irfan Essa, Aaron Bobick, and Nipun Kwatra. 2005. Texture Optimization for Example-based Synthesis. *ACM Trans. Graph.* 24, 3 (July 2005), 795–802. https://doi.org/10.1145/1073204.1073263

Vivek Kwatra, Arno Schödl, Irfan Essa, Greg Turk, and Aaron Bobick. 2003. Graphcut Textures: Image and Video Synthesis Using Graph Cuts. In *SIGGRAPH '03*. 277–286. https://doi.org/10.1145/1201775.882264

Pierre-Edouard Landes, Bruno Galerne, and Thomas Hurtut. 2013. A Shape-Aware Model for Discrete Texture Synthesis. *Computer Graphics Forum* 32, 4 (2013), 67–76.

Yifei Li, David E. Breen, James McCann, and Jessica Hodgins. 2019. Algorithmic Quilting Pattern Generation for Pieced Quilts. In *Proceedings of the 45th Graphics Interface Conference on Proceedings of Graphics Interface 2019 (GI'19)*. Article 13, 9 pages. https://doi.org/10.20380/GI2019.13

Lin Liang, Ce Liu, Ying-Qing Xu, Baining Guo, and Heung-Yeung Shum. 2001. Real-time Texture Synthesis by Patch-based Sampling. *ACM Trans. Graph.* 20, 3 (July 2001), 127–150. https://doi.org/10.1145/501786.501787

Hugo Loi, Thomas Hurtut, Romain Vergne, and Joelle Thollot. 2017. Programmable 2D Arrangements for Element Texture Design. *ACM Trans. Graph.* 36, 4, Article 105a (May 2017). https://doi.org/10.1145/3072959.2983617

Jingwan Lu, Connelly Barnes, Connie Wan, Paul Asente, Radomir Mech, and Adam Finkelstein. 2014. DecoBrush: Drawing Structured Decorative Patterns by Example. *ACM Trans. Graph.* 33, 4, Article 90 (July 2014), 9 pages. https://doi.org/10.1145/2601097.2601190

Jingwan Lu, Fisher Yu, Adam Finkelstein, and Stephen DiVerdi. 2012. HelpingHand: Example-based Stroke Stylization. *ACM Trans. Graph.* 31, 4, Article 46 (July 2012), 10 pages. https://doi.org/10.1145/2185520.2185542

Chongyang Ma, Li-Yi Wei, Sylvain Lefebvre, and Xin Tong. 2013. Dynamic Element Textures. *ACM Trans. Graph.* 32, 4, Article 90 (July 2013), 10 pages. https://doi.org/10.1145/2461912.2461921

Chongyang Ma, Li-Yi Wei, and Xin Tong. 2011. Discrete Element Textures. *ACM Trans. Graph.* 30, 4, Article 62 (July 2011), 10 pages. https://doi.org/10.1145/2010324.1964957

Jonàs Martínez, Jérémie Dumas, Sylvain Lefebvre, and Li-Yi Wei. 2015. Structure and Appearance Optimization for Controllable Shape Design. *ACM Trans. Graph.* 34, 6, Article 229 (Oct. 2015), 11 pages. https://doi.org/10.1145/2816795.2818101

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

Mathieu Nancel and Andy Cockburn. 2014. Causality: A Conceptual Model of Interaction History. In *CHI '14*. 1777–1786. https://doi.org/10.1145/2556288.2556990

Hans Pedersen and Karan Singh. 2006. Organic Labyrinths and Mazes. In *NPAR '06*. 79–86. https://doi.org/10.1145/1124728.1124742

Mengqi Peng, Li-Yi Wei, Rubaiat Habib Kazi, and Vladimir G. Kim. 2020. Autocomplete Animated Sculpting. In *UIST '20*. https://doi.org/10.1145/3379337.3415884

Mengqi Peng, Jun Xing, and Li-Yi Wei. 2018. Autocomplete 3D Sculpting. *ACM Trans. Graph.* 37, 4, Article 132 (July 2018), 15 pages. https://doi.org/10.1145/3197517.3201297

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

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

Christian Santoni and Fabio Pellacini. 2016. gTangle: A Grammar for the Procedural Generation of Tangle Patterns. *ACM Trans. Graph.* 35, 6, Article 182 (Nov. 2016), 11 pages. https://doi.org/10.1145/2980179.2982417

Christian Schumacher, Bernhard Thomaszewski, and Markus Gross. 2016. Stenciling: Designing Structurally-Sound Surfaces with Decorative Patterns. *Computer Graphics Forum* 35, 5 (2016), 101–110.

Ryo Suzuki, Tom Yeh, Koji Yatani, and Mark D Gross. 2017. Autocomplete Textures for 3D Printing. *arXiv preprint arXiv:1703.05700* (2017).

Kenshi Takayama, Olga Sorkine, Andrew Nealen, and Takeo Igarashi. 2010. Volumetric Modeling with Diffusion Surfaces. In *SIGGRAPH ASIA '10*. Article Article 180, 8 pages. https://doi.org/10.1145/1866158.1866202

Peihan Tu. 2020. Continuous Curve Textures Source Code. https://github.com/tph9608/continuous-curve-texture/.

Peihan Tu, Dani Lischinski, and Hui Huang. 2019. Point Pattern Synthesis via Irregular Convolution. *Computer Graphics Forum* 38, 5 (2019), 109–122. https://doi.org/10.1111/cgf.13793

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

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

Li-Yi Wei. 2016. Texture Synthesis. https://github.com/1iyiwei/texture

Li-Yi Wei, Sylvain Lefebvre, Vivek Kwatra, and Greg Turk. 2009. State of the Art in Example-based Texture Synthesis. In *Eurographics 2009, State of the Art Report, EG-STAR*. Eurographics Association. http://www-sop.inria.fr/reves/Basilic/2009/WLKT09

Li-Yi Wei and Marc Levoy. 2000. Fast Texture Synthesis Using Tree-structured Vector Quantization. In *SIGGRAPH '00*. 479–488. https://doi.org/10.1145/344779.345009

Li-Yi Wei and Marc Levoy. 2001. Texture Synthesis over Arbitrary Manifold Surfaces. In *SIGGRAPH '01*. 355–360. https://doi.org/10.1145/383259.383298

Jun Xing, Hsiang-Ting Chen, and Li-Yi Wei. 2014. Autocomplete Painting Repetitions. *ACM Trans. Graph.* 33, 6, Article 172 (Nov. 2014), 11 pages. https://doi.org/10.1145/2661229.2661247

Jun Xing, Li-Yi Wei, Takaaki Shiratori, and Koji Yatani. 2015. Autocomplete Hand-drawn Animations. *ACM Trans. Graph.* 34, 6, Article 169 (Oct. 2015), 11 pages. https://doi.org/10.1145/2816795.2818079

Jonas Zehnder, Stelian Coros, and Bernhard Thomaszewski. 2016. Designing Structurally-sound Ornamental Curve Networks. *ACM Trans. Graph.* 35, 4, Article 99 (July 2016), 10 pages. https://doi.org/10.1145/2897824.2925888

Howard Zhou, Jie Sun, Greg Turk, and James M. Rehg. 2007. Terrain Synthesis from Digital Elevation Models. *IEEE Transactions on Visualization and Computer Graphics* 13, 4 (July 2007), 834–848. https://doi.org/10.1109/TVCG.2007.1027

Kun Zhou, Xin Huang, Xi Wang, Yiying Tong, Mathieu Desbrun, Baining Guo, and Heung-Yeung Shum. 2006. Mesh Quilting for Geometric Texture Synthesis. *ACM Trans. Graph.* 25, 3 (July 2006), 690–697. https://doi.org/10.1145/1141911.1141942

Shizhe Zhou, Changyun Jiang, and Sylvain Lefebvre. 2014. Topology-constrained Synthesis of Vector Patterns. *ACM Trans. Graph.* 33, 6, Article 215 (Nov. 2014), 11 pages. https://doi.org/10.1145/2661229.2661238

C. Lawrence Zitnick. 2013. Handwriting Beautification Using Token Means. *ACM Trans. Graph.* 32, 4, Article 53 (July 2013), 8 pages. https://doi.org/10.1145/2461912.2461985

A EXISTENCE ASSIGNMENT

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

1: **function** GenerateNewOutputSamples($\{\mathbf{n}(s_i), \mathbf{n}(s_o)\}$)
2: $\{s_o^c\} \leftarrow \emptyset$ {candidate sample set}
3: $\{\mathcal{S}\} \leftarrow \emptyset$ {$\mathcal{S}$ is a cluster that contains some candidate samples}
4: $\{s_o\} \leftarrow \emptyset$ {new output sample set}
5: **for** $\mathbf{n}(s_i), \mathbf{n}(s_o) \in \{\mathbf{n}(s_i), \mathbf{n}(s_o)\}$ **do**
6:     **for** $s_i' \in \mathbf{n}(s_i)$ **do**
7:         **if** s_i' is unmatched **then**
8:             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'
9:             $\{s_o^c\} \leftarrow \{s_o^c\} \cup s_o^c$
10:         **end if**
11:     **end for**
12: **end for**
13: **for** $s_o^c \in \{s_o^c\}$ **do**
14:     {Generate clusters from the candidate sample set by greedily looping over all candidates; more advanced clustering technique can be applied to replace this step}
15:     Find the $\mathcal{S}$ within $\{\mathcal{S}\}$ with nearest center to s_o^c
16:     **if** Distance($\mathcal{S}, s_o^c$) $< 0.5\delta$ **then**
17:         {Distance($\mathcal{S}, s_o^c$) computes the spatial distance between the center of $\mathcal{S}$ and s_o^c}
18:         $\mathcal{S} \leftarrow \mathcal{S} \cup s_o^c$
19:     **else**
20:         $\mathcal{S}' \leftarrow \emptyset$
21:         $\mathcal{S}' \leftarrow \mathcal{S}' \cup s_o^c$
22:         $\{\mathcal{S}\} \leftarrow \{\mathcal{S}\} \cup \mathcal{S}'$
23:     **end if**
24: **end for**
25: **for** $\mathcal{S} \in \{\mathcal{S}\}$ **do**
26:     create a new s_o with averaged sample positions and attributes by merging all $\{s_o^c\}$ within $\mathcal{S}$
27:     $i(s_o) \leftarrow \frac{\#\mathcal{S}}{\#\text{overlapping } \mathbf{n} \text{ over } s_o}$ {existence assignment}
28:     **if** $i(s_o) > 0.5$ **then**
29:         $\{s_o\} \leftarrow s_o$
30:     **end if**
31: **end for**
32: **return** $\{s_o\}$

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

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 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 sample distribution every iteration.

B PARAMETERS

Table 2. *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 12h	$\{50, 40, 30\}$	$\{20, 15, 10\}$	350×200
Figure 12p	$\{60, 50\}$	$\{40, 30\}$	300×300
Figure 1e	$\{40, 30, 20\}$	$\{20, 15, 10\}$	250×250
Figure 13j	$\{60, 50, 40\}$	$\{40, 30, 30\}$	250×250
Figure 12f	$\{50, 40, 30\}$	$\{30, 20, 10\}$	300×350
Figure 1h			400×400
Figure 16e	⋮	⋮	400×300
Figure 12t			500×400
Figure 15b			350×300
Figure 12d			350×200
Figure 12x	⋮	⋮	300×300
Figure 12j			200×300
Figure 15f			300×300
Figure 12n	$\{60, 50, 40\}$	$\{40, 30, 25\}$	200×300
Figure 12r			400×400
Figure 14h			350×400
Figure 16j			400×250
Figure 1a	⋮	⋮	300×250
Figure 1d			350×300
Figure 14c			350×250
Figure 17c			500×300
Figure 13n			400×200
Figure 17l	⋮	⋮	250×350
Figure 12v			250×400
Figure 12b			300×300
Figure 13f			300×250

Table 2 lists the parameters for the results shown in the paper.

C 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 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 [Kuhn 1955] 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 [Barnes et al. 2009] is composed of two alternating steps: propagation and random search. In an optimization step, there are the $an_{O}I_{max}$ 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_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 $O(n_O N_{\mathbf{n}}^3)$. 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 4.3.5).

D TEXTURE SYNTHESIS

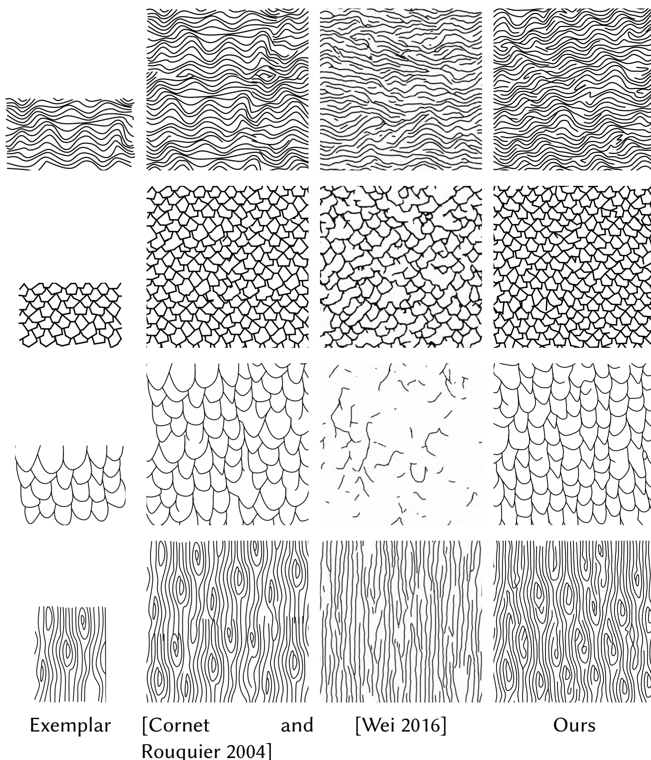


Fig. 19. *Comparison of our algorithm to texture synthesis.* Texture synthesis methods need additional vector-pixel conversions and need to process all pixels instead of just samples around patterns.

Figure 19 shows the texture synthesis results by graph cut [Kwatra et al. 2003] (using the implementation in the GIMP Texturize Plugin [Cornet and Rouquier 2004]) and multi-resolution patchmatch (using the implementation in [Wei 2016] with guidance channels [Kaspar et al. 2015]). 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.