Interactive example-palettes for discrete element texture synthesis

Timothy Davison, Faramarz Samavati, Christian Jacob
{davison,samavati,jacob}@ucalgary.ca

April 2, 2018

Abstract

Textures composed of individual discrete elements are found in everything from human-made glass-tilings to forests and tropical coral. Previous work in discrete element texture synthesis has been limited to synthesizing scenes composed of one discrete element texture. We propose an interactive sketch-based system for synthesizing scenes composed of many discrete element textures. Our main idea is an example-palette, where a user can use our sketch-based tools to create and combine discrete element textures before painting them into a scene. Our interactive sketch-based tools use a new and fast region-growing algorithm that iteratively synthesizes new elements derived from the example-palette. We demonstrate the application of our system for building virtual worlds (such as for video games) and sketch-based modeling. We achieve results that are not possible with state-of-the-art techniques.

1 Introduction

Our world is filled with repeating and semi-repeating arrangements of discrete elements. Some are simple, like cobblestone pathways, while others are elaborate and intricate, like glass tilings.

Consider the problem of building a hypothetical bunny-planet for a children's fantasy computer game (Figure 1). Gravity pulls one to the surface of the bunny-planet. Our bunny-planet is covered in forests, crops, meadows, villages and even mushroom gardens. Cobblestone pathways weave across its surface. We introduce an interactive, sketch-based system for synthesizing virtual worlds and objects composed of multiple discrete element textures.

In example-based discrete element texture synthesis, the user provides a small example of how the cobblestone in a pathway is arranged. An algorithm uses the example to synthesize locally similar non-repeating output. Previous techniques in example-based discrete element texture synthesis are limited to a single example, so something like our bunny-planet is impossible with them. Furthermore, for synthesizing virtual worlds, it is critical to keep the human in the loop to guide synthesis but to remove the tedious task of manual element placement. These are the main problems we set out to solve.

Our key idea is an interactive, sketch-based, example-palette for discrete element texture synthesis (Figure 2). In our bunny-planet example, we created an example-palette, using our sketch-based tools, with discrete element textures for the forest, the cobblestone pathways, the village houses, and mushroom gardens. One selects a texture, such as the mushrooms (or even a combination of textures), and with our sketch-based tools, sketches the mushroom texture into the scene. As the user sketches, our fast region-growing algorithm is synthesizing elements along the brush path in real-time. Just like mixing paint in a palette to create new colors, one can quickly and interactively sketch new textures into the example-palette derived from other textures (Figure 9). With our interactive system, one can create complicated and intricate virtual worlds and objects with little effort.

Our contributions are the following:

- An interactive sketch-based **example-palette** for synthesizing virtual worlds and objects composed of multiple discrete element textures (Section 5.2).
- A fast, novel, **region-growing algorithm** that iteratively synthesizes new elements based on previously synthesized elements (Section 5). We introduce a method to keep the active-problem size small (Section 5.1).
- Our **sketch-based tools** (Section 5.5) guide our region-growing algorithm to synthesize new elements. We demonstrate applications for sketch-based modeling.
- We synthesize elements directly on **3D surfaces** (Section 5.7). Our method allows the user to sketch directly on a surface without having to worry about surface parameterization or texture mapping.

Our method enables us to synthesize virtual worlds and objects not possible with previous methods—for example, our glass-tiled bottle (Figure 11) or our bunny-planet (Figure 1). We analyze our method in comparison to previous methods in Section 6.1.

2 Related work

Our work belongs to the area of example-based discrete element texture synthesis. Discrete element textures are an extension of the image texture synthesis idea to discrete elements. Our work is also related to geometry synthesis, model synthesis, statistical synthesis and procedural modeling.

In **example-based texture synthesis** a 2D example image is used to synthesize a large scale texture [1]. Pixel-based approaches commonly choose pixels to add based upon neighborhood comparisons between the previously synthesized pixels and example pixels (Efros and Leung [2] and Wei and Levoy [3]). Wei and Levoy [4] synthesize textures over arbitrary manifold surfaces. Later works reduced the number

Figure 1: This hypothetical bunny-planet, that one might find in a children's computer game, was interactively synthesized with our system. We designed the discrete element textures in the example-palette (left) and applied them to the Standford bunny mesh using our sketch-based generative tools. (top right) A whole view of the bunny-planet. The supplementary material contains a video of the bunny-planet design (filename: bunny_planet.mp4).

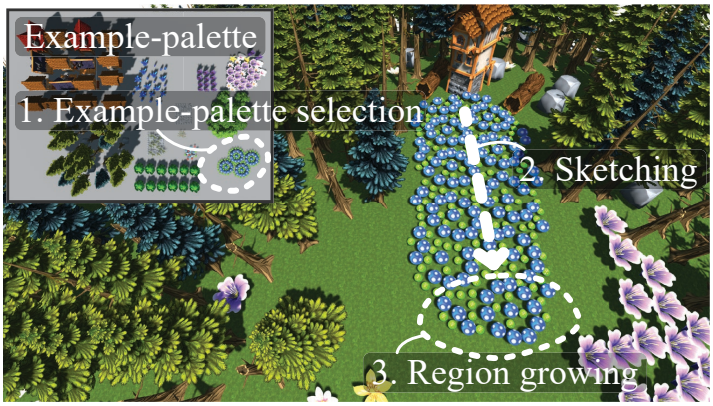


Figure 2: Interactive synthesis of a mushroom garden. 1) The user selects a mushroom garden discrete element texture from the example-palette. 2) The user sketches the garden onto the surface. 3) Our fast region-growing algorithm iteratively synthesizes new elements in the highlighed area as the user sketches.

of neighborhood comparisons making synthesis much faster (Ashikhmin [5] and Tong et al. [6]). Kwatra et al. [7] reframe the problem of texture synthesis as one of energy minimization, while Han et al. [8] build on this with a fast and discrete solver. Cohen et al. [9] use Wang tiles as a fast way to synthesize large non-repeating textures composed of related tiles. Recently Li and Wand [10] achieve real-time texture synthesis with generative neural networks. Texture synthesis has been applied for multi-scale synthesis [11, 12].

Discrete element texture synthesis is similar to example-based texture synthesis, with the additional complication of where to place elements (we are no longer synthesizing pixels on a convenient regular grid). Many methods for discrete element texture synthesis also use the neighborhood comparison ideas developed in image texture synthesis. Ijiri et al. [13] synthesize 2D arrangements of elements by incrementally growing a network of interconnected elements and is closely related to Barla et al. [14]'s method for synthesizing stroke patterns. Ma et al. [15] extend Kwatra et al.'s [7] expectation maximization framework to discrete element textures. Ma et al. [16] synthesize dynamic swimming schools of fish. Xing et al. [17] apply discrete element texture synthesis for drawing autocompletion [17]. Roveri et al. [18] reframe discrete element texture synthesis as a continuous problem for repetitive 3D structure synthesis.

Our region-growing algorithm is related to the fast and interactive region-growing algorithm in Ijiri et al.'s [13] work. However, their system only supports a single example texture. Furthermore, we achieve new results (such as the coral in Figure 3 or the glass tilings in Figure 11) that are impossible with their method (Section 4.1).

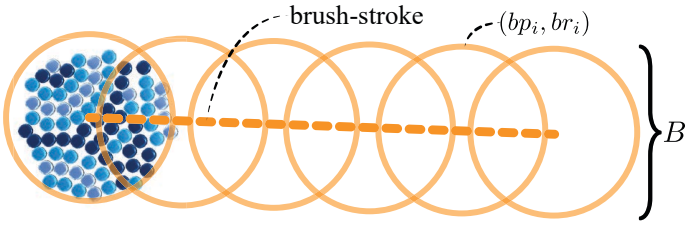
We build on the work of Ma et al. [15], in particular, their optimization framework and neighborhood comparisons with the addition of our fast region-growing algorithm. In contrast to their method, we support multiple example textures, an interactive sketch-based interface, and a dramatic increase in the rate of synthesis. We also demonstrate that region-growing can synthesize textures not possible with Ma et al.'s method (Section 6.1).

Related to discrete element texture synthesis, **statistical synthesis** synthesizes elements randomly to fit example distributions—often represented as histograms. Hurtut et al.

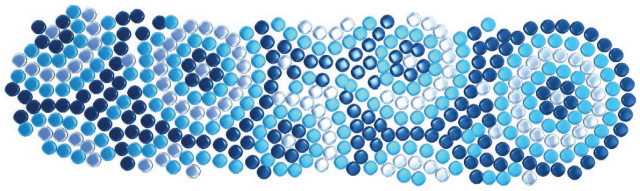
[19] consider the bounding boxes of elements during synthesis, while Landes et al. [20] improve on this with a shape-aware model. Roveri et al. [21] synthesize point distributions with adaptive density and correlations. Recently, Emilien et al. [22] and Gain et al. [23] use sketch-based tools for synthesizing element distributions for building virtual worlds.

The example-palette and sketch-based tools of Emilien et al. [22] are powerful worldbuilding tools. However, the variety of element arrangements that can be synthesized with statistical synthesis methods like this are limited—for example, our glass tilings (Figure 11) would be difficult to achieve with this method. Therefore, we propose the example-palette for discrete element texture synthesis as another set of tools in the virtual-world-builder's toolbox.

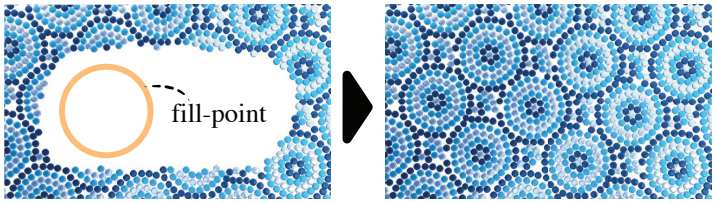
Geometry Synthesis, Model Synthesis and Procedural Modelling Bhat et al. [24] introduce the idea of example-based synthesis for geometric textures on 3D surfaces. Zhou et al. [25] took this a step further and generated quilted surface geometries. This idea was refined by Yuksel et al.'s [26] multi-stage pipeline based on stitch meshes. 3D procedural models have been synthesized based on example models (Merrell and Manocha [27] and Peytavie et al. [28]). Bokeloh et al. [29] build a shape grammar by analyzing an input model for symmetric regions. The shape grammar is used to semi-manually or automatically generate 3D models. Synthesizing structured patterns with space colonization algorithms have been used for modeling trees (Runions et al. [30]). Palubicki et al. [31] use sketch-based interfaces to generate trees. Li et al. [32] guide the growth of grammars across a surface with a user defined-tensor field. More recently, an example-based system for sketching structured decorative patterns was developed by Lu et al. [33]. Finally, Guerrero et al. [34] synthesize patterns with a tool that explores pattern variations.



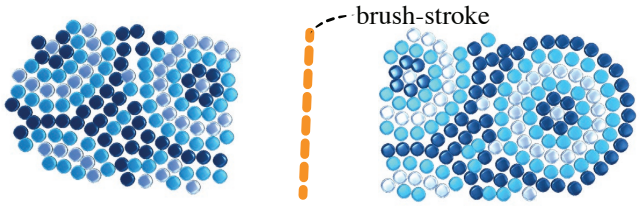
(a) **Generative-Brush** A randomly selected patch from the example-palette selection is copied to the output at the first brush point to seed region-growing along the rest of the brush points in B.



(b) The output domain after synthesizing elements along the brush-stroke in Figure 3a.



(c) **Filler Tool** The empty region is filled with elements starting at the fill-point.



(d) **Eraser** Removing elements along the brush stroke (orange dashed-line).

Figure 3: Some of our sketch-based tools.

3 Overview of our approach

In this section, we provide an overview of our approach and its major components: the example-palette, region-growing and optimization, surface mapping and sketch-based interaction.

An artist uses a paint-palette to combine paint colors before applying them to canvas; A user of our system paints with discrete element textures into a scene or onto an object. In our system, a user creates discrete element textures in the example-palette. Those textures can be selected and then applied to virtual worlds and objects with a generative sketch-based brush—we also support other tools, such as erasers and filler tools. Furthermore, the generative brush can be used to create new textures in the example-palette derived from other discrete element textures in the example-palette (Figure 3), just like an artist mixing paint on a paint-palette. Our generative tools are based on a new fast region-growing algorithm for discrete element texture synthesis. An interleaved optimization step further improves previously synthesized element arrangements.

3.1 Sketch-based tools and the example-palette

We support various sketch-based tools for synthesizing discrete element textures. The *generative-brush* (Figure 3a and 3b) synthesizes new elements in a small region along the brush path. The generative-brush can also be applied to previously synthesized results, in which case it will optimize their appearance relative to a texture selected from the example-palette. For example, one can use this method to repair undesired arrangements on a cobblestone road or even to transform a cobblestone road into a mushroom garden. With the *filler tool* (Figure 3c) the user sets a fill point where there are no elements, and then we synthesize new elements until there is no more room to do so (or the user tells us to stop). Finally, the *eraser* (Figure 3d) removes elements within a certain distance from along a brush path. These tools work on the scene (for example, the bunny-planet) and the example-palette.

The example-palette can be built through manual element placement and attribute assignment. Copy-paste operations

may also be applied. However, a more exciting possibility enabled by our system is to design the example-palette textures using our sketch-based tools and textures selected from elsewhere in the example-palette (Figure 9(a-e)).

3.2 Discrete element texture synthesis

Our texture synthesis method has two interleaved steps. A **generation** step uses a region-growing algorithm to iteratively synthesize new elements (Section 5). The region-growing algorithm synthesizes elements based on an example discrete element texture selected from the example palette. An **optimization** step relaxes the arrangement of newly synthesized elements relative to the example palette selection (Section 5.3).

One of our main challenge is deciding if and where new elements can be synthesized. We track where new elements can be synthesized with so-called free-space points (Section 5.1). We derive free-space points from analysis on the example-palette. We use free-space points as an efficient method to keep the active problem size small, achieving high rates of synthesis as a result.

Region-growing is well suited for sketch-based tools. New elements are synthesized in a region-growing front at each iteration of the algorithm. To synthesize elements along a path, we seed region-growing at the start with a small example copied from the example-palette. Region-growing then iteratively adds new elements—so long as they are within a certain distance of the path—until it reaches the end of the path.

It is possible to use region-growing alone to synthesize elements. However, the local and greedy way that we synthesize new elements can lead to artifacts, such as gaps in the output. To solve this, we took inspiration from Kwatra et al. [7] and Ma et al. [15], and interleave an optimization step to reduce these artifacts. We compare region-growing alone to region-growing with optimization, in our results and analysis (Section 6.1).

We adapt our generation and optimization steps to 3D surfaces through the use of a mapping function (Section 5.7). We compose a local mapping function wherever we want to generate new elements; the function maps from a Cartesian grid to an orientation field over the surface. The orientation field is generated automatically before we start synthesizing elements. The user also has the option to interactively design it by placing singularities, as described by Crane et al. [35].

4 Discrete element texture synthesis

The goal of image texture synthesis is to generate image textures which are maximally similar around every pixel in the output to pixels in the example (Efros and Leung [2]). With discrete element textures, the goal is the same, except we are comparing element positions and attributes in a neighborhood, instead of pixel colors (Ma et al. [15]).

In our system, the user makes a selection from the example-palette, and our goal is to generate element arrangements that are similar to the selected texture. Elements are synthesized using a region-growing algorithm that uses the selected texture as an example of what to synthesize. New elements are iteratively synthesized next to previously synthesized elements in such a way that they are maximally perceptually-close with the example texture. We calculate *neighbourhood distance* between the example and output neighborhoods from the relative position and attributes of elements in the two neighborhoods. We use neighborhood distance to find elements from the example to copy to the output.

4.1 Neighbourhood Similarity

Let $e \in \mathcal{E}$ be an element in the example-palette and let $o \in \mathcal{O}$ be an element in our output domain. The output domain is the region, typically a surface, where elements from the example-palette will be synthesized. Every element has a position $p_e \in \mathbb{R}^3$, various attributes captured in a vector a_e (such as shape, color or type) and a bounding radius $r_e \in \mathbb{R}$. The position and radius define a bounding sphere that can be used to check for element overlaps quickly. The trees and rocks in our bunny-planet have significantly different sizes (Figure 1). Therefore, the bounding sphere is customizable for each element.

The selection $\mathcal{S}$ is a subset of the example-palette, $\mathcal{S} \subset \mathcal{E}$. We synthesize new elements using the selection as an example. We do not differentiate between textures in the example palette, the elements of those textures all belong to the same set $\mathcal{E}$. For example, the trees, mushrooms, and cobblestone in the bunny-planet example-palette all belong to the same set $\mathcal{E}$ (Figure 1). Synthesis is based on what subset of the example-palette is selected. Our system is flexible because the user can select portions of a texture with a desired arrangement, a whole texture, or even multiple textures (Figure 9(a-e)).

In texture synthesis applications, the similarity between an exemplar and synthesized elements is commonly based on a neighborhood distance function. The goal of this function is to determine how perceptually distant two different neighborhoods are. This function has a low value when two neighborhoods are similar.

We require our function to have two properties: 1) it must measure differences between output and exemplar neighborhoods and 2) it must identify exemplar elements to copy to the output domain during the region-growing step.

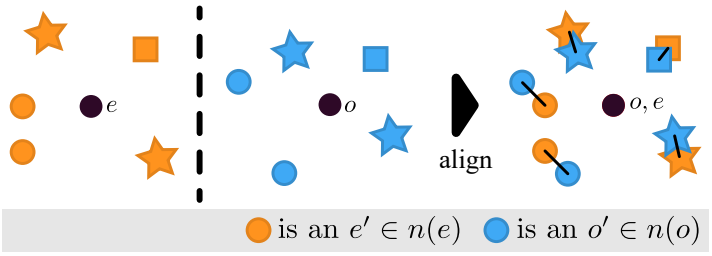


Figure 4: The similarity between two neighborhoods is found by first aligning those neighborhoods at their centroids e and o. Then, the distance between pairs of elements is summed and the attributes of those elements compared (in this case shape-type, as given by stars, circles and squares).

To measure how distant two element neighbourhoods are, we align the centroids of two neighbourhoods and compare the distance between pairs of points as illustrated in Figure

4. Let $e' \in n(e)$ (where $n(e) \subset \mathcal{E}$) be an element in the geometric neighbourhood of e around p_e. The geometric neighbourhood $n(e)$ of e are all those elements in $\mathcal{E}$ within a neighbourhood radius $\bar{r}_o$ of p_e. The customizable element radius r_e and the customizable neihbourhood radius $\bar{r}_o$ allow us to synthesize discrete element textures with different scales (the cobblestone pathways and the trees in Figure 1 have different element and neighbourhood radii). The distance between an output neighbourhood $n(o)$ and an example neighbourhood $n(e)$ is given by:

$$|n(o) - n(e)| = \sum_{o' \in n(o)} |(p_{o'} - p_o) - (p_{e'} - p_e)|^2 + \omega(o', e'). \quad (1)$$

ω is a function that compares the attributes of two elements. o' is an element in the neighbourhood $n(o)$, the pair for that element in $n(e)$ is denoted with e'.

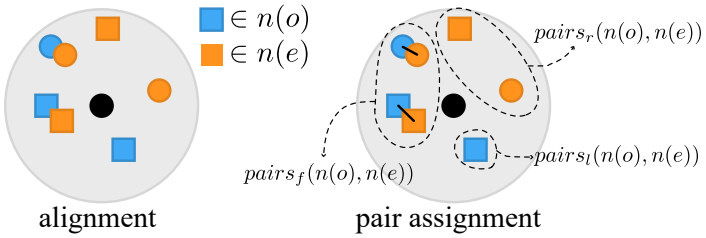


Figure 5: In finding the pairings between two neighbourhoods $n(o)$ and $n(e)$ we align the two neighbourhoods at their centroids. Next, we find the full pairings between elements that are close enough $pairs_f(n(o), n(e))$ and the leftover partial pairings $pairs_l(n(o), n(e)), pairs_r(n(o), n(e))$.

Our generation and optimization steps rely on the notion of full and partial assignment of pairs of points between two neighbourhoods $n(e)$ and $n(o)$ (Figure 5). A partial assignment occurs when we cannot form pairs between all of the elements in $n(e)$ and $n(o)$. There are two sets of partial pairings, the left-partial pairings $pairs_l(n(o), n(e))$ with the form $\{(o', \mathbf{0})\}$) and the right-partial pairings $pairs_r(n(o), n(e))$ of the form $\{(\mathbf{0}, e')\}$). The set of full pairings is $pairs_f$.

We use a greedy pair assignment algorithm. First, we align the input and output neighborhood. For each element in the output neighborhood, we find the nearest input element that has the same attributes. We do not pair elements that are too far apart.

In detail, for two neighbourhoods $n(o)$ and $n(e)$ aligned at p_o and p_e, we sort the elements in the output neighbourhood by distance from p_o. Then, for each sorted element $o' \in n(o)$ in the neighbourhood of the output element, we find the closest element $e' \in n(e)$. If this element is within a certain threshold distance c, where $|(p_{o'} - p_o) - (p_{e'} - p_e)|^2 \leq c$, then we form the full-pair (o', e') and remove e' from further consideration. Otherwise, we form the left-partial pair $(o', \mathbf{0})$. The remaining elements in the exemplar neighbourhood $e' \in n(e)$ form the right-partial pairs $\{(\mathbf{0}, e')\}$.

In our implementation, we construct a k-d tree for the output $\mathcal{O}$ and input $\mathcal{E}$ domains. During pair assignment the cost to find the nearest element $e' \in n(e)$ to an output element $o' \in n(o)$ is an $O(log\ n)$ operation in the size of $\mathcal{E}$. The cost of our pair assignment algorithm is $O(m\ log\ n)$, where m is the largest neighbourhood size in $\mathcal{O}$ or $\mathcal{E}$.

5 Region-Growing and Optimization for Discrete Element Texture Synthesis

Our goal is to minimize the energy (Kwatra et al. [7] and Ma et al. [15]) of neighborhoods in the output $\mathcal{O}$ relative to the most similar neighborhoods in the example-palette selection $\mathcal{S}$:

$$E(\mathcal{O}, \mathcal{S}) = \sum_{o \in \mathcal{O}} |n(o) - n(e)|, e \in \mathcal{S}. \quad (2)$$

We approach this problem by greedily generating new elements that minimize Equation 2. Next, we relax elements in the horizon through re-assignment of positions and attributes. Our region-growing algorithm consists of three main steps: 1) *seed selection* and *generation* , 2) *optimization*, and 3) *free space updating*.

For interactive applications, fast texture synthesis is critical. Therefore, we reduce the difficulty of the synthesis problem by considering a small subset of the output domain, the horizon $\mathcal{H} \subset \mathcal{O}$. The horizon is a region containing recently synthesized elements.

We generate new elements around so-called **seed elements** (Figure 6a). Seed elements are a small subset of previously synthesized elements in the horizon $\mathcal{H}$ that can generate new elements. We can quickly determine if a seed can generate new elements by checking if it has nearby free-space points.

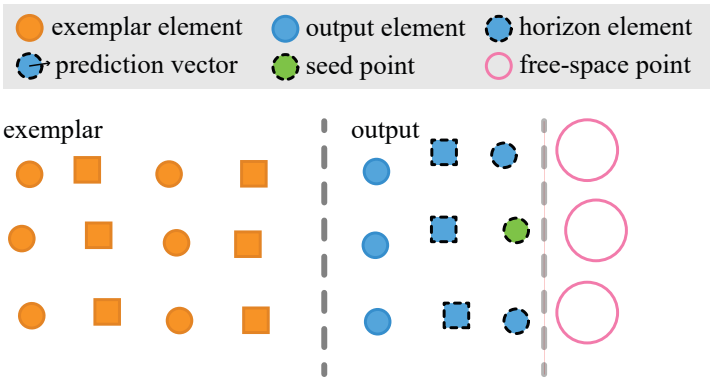
During **generation** (Figure 6b), we visit each seed in the horizon and search the example-palette selection for a neighborhood that is maximally similar to the seed's neighborhood. If any of the elements in the exemplar neighborhood overlap with a free-space point, we copy the elements to the output.

During **optimization** (Figure 6c), we visit each element in the horizon and find the most similar neighbourhood in the example-palette selection. The example neighborhoods are aligned with the output neighborhoods, and the element pairings between them found. The difference in positions and attributes between pairs of elements are used by an optimization step to adjust the output to look more like the example-palette selection.

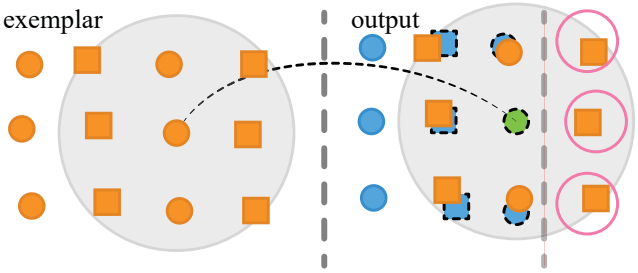
Finally, we **update the free-space** points (Figure 6d). We add new free-space points, derived from an analysis preprocessing step on the exemplar, to the output around the newly synthesized elements. Then, we remove the free-space points that now overlap with the newly synthesized elements. If any of the output elements are not nearby a free-space point, they are removed from the horizon. The next iteration of our algorithm starts back at the seed-selection step.

5.1 Generation and free-space

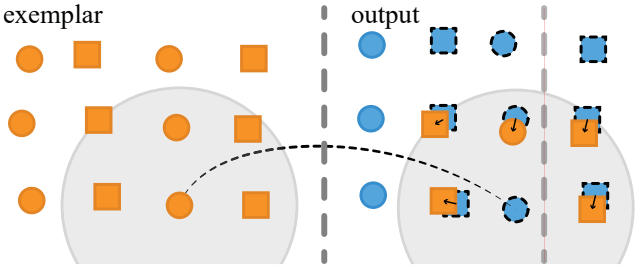
In the generation step, we find a set of seed elements $Seeds \subset \mathcal{H}$ that will generate new elements that overlap with nearby



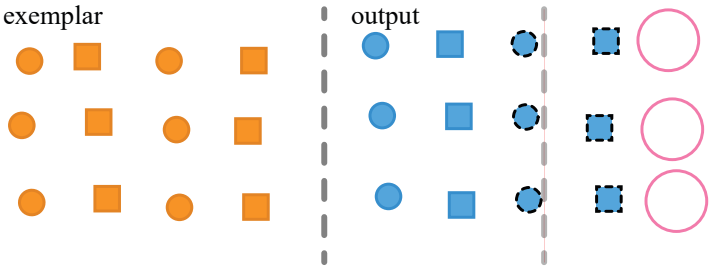
(a) **Seed Selection** A set of seed elements are selected from the horizon (in this example there is only one) such that the seeds also have free-space points nearby.



(b) **Generation** For each seed element, the most similar neighbourhood in the example-palette selection is found whose elements overlap with the free-space points. Those overlapping elements are copied into the output domain and added to the horizon. The free-space points are removed.



(c) **Optimization** For each element in the horizon, the most similar neighbourhood in the example-palette selection is found and aligned with the horizon (here only one aligned neighbourhood is shown). After pairing the elements in the horizon, the difference is positions between the pairs (arrows) are used by the optimization to adjust the positions of the horizon elements.



(d) **Free-Space Updating** The output domain after generation and optimization. Elements are pruned from the horizon and new free-space points are found.

Figure 6: An example of one round of element generation and subsequent optimization. The objective is to expand the elements to the right, past the light-gray dotted line. A legend appears at the top.

free-space points. Seeds are selected so that each seed is separated from its neighbors by some minimum distance d.

We track where to generate new elements with a set of free-space points F. A free space point $v \in F$ has a bounding radius r_v and a position p_v. For each seed $s \in Seeds$, we find the most similar example element s_e such that $n(s_e)$ contains elements that overlap with a free-space point, that is $\exists e' \in n(s_e)$ that overlaps with a free space point $v \in F$. Next, we copy each of those overlapping elements to the horizon as $o_{e'}$ and remove v from F. Intuitively, the new position of $o_{e'}$ is found by transforming the neighbourhood of s_e to align with s and assigning $o_{e'}$ that transformed position. The new position of $o_{e'}$ is $p_{s'_e} = m_{p_s}(p_{s_e} - p_{e'}) + p_s$. Here, m_{p_s} is a mapping at p_s that can be used to map example space onto a 3D surface at that point.

Sometimes, the most similar neighborhood that can generate elements is not the best one to pick—we need to ignore bad suggestions. If we do not, we can introduce artifacts that are difficult to correct with optimization or subsequent rounds of region-growing. If e_0 has the most similar neighborhood to a seed s but does not generate any elements and e_i is another element whose neighborhood does then we compare the similarity of those two neighborhoods. If $|n(o) - n(e_0)|/|n(o) - n(e_i)| > c$ for some constant c we abandon the seed element without adding the found elements.

To keep the size of the optimization and generation problems small, we prune the horizon of all those elements that cannot predict new elements at the start of the generation step. An element $h \in \mathcal{H}$ in the horizon is pruned when there is no free-space point in F that overlaps with a bounding sphere around p_h of radius $\bar{r}_h$.

Rather than using brute-force search of $\mathcal{E}$ to find the most similar example element to an output element, we take advantage of the properties of Markov Random Field textures, namely *coherence* and *locality* (Efros and Leung [2]). *Locality* states that the position and attributes of an element relative to other elements depend only on nearby elements. *Stationarity* states that *locality* is independent of element position. The locality property implies that elements that are together in the exemplar will also tend to be together in the output domain, which is *coherence* (Ashikhmin [5] and Tong et al. [6]). As in Ma et al. [15], we use the idea of k-coherence search to reduce the size of the search space to a user-defined k (typically between 2 and 5).

For each example element $e \in \mathcal{E}$, k-coherence search caches the k most similar examples to e as $co(e)$. The example element that was used to generate a particular output element o is o_e. During synthesis, instead of a brute-force search of $\mathcal{E}$ for the most similar example element to o we can just search through $co(o_e)$. The idea is to exploit the coherence of elements in the output domain. A major benefit of k-coherence search is that increasing the number of elements in an example does not decrease the rate of synthesis.

Before deciding what element to copy to the output, we must decide if and where an element might be placed. We accelerate this decision with free-space points. The idea is to reduce the number of example-palette neighbourhoods that generation has to search through. In a pre-processing step, we perform clustering on the neighborhoods in $co(e)$ for each example-palette element e to reduce the set to a single set

of free-space points. This set of free-space points describes where elements might be synthesized in the output for output elements derived from that element. We use free-space points to search for where elements can be generated, but also to efficiently prune elements from the horizon.

Free-space points are generated by a pre-processing step on the example-palette that exploits the concept of coherence. We consider each example element e in turn, then for each $c \in co(e)$ we take all of the positions of the elements in the neighbourhoods of each $n(c)$ to get the set of vectors $free(e) = \{p_{c'} - p_c | c' \in n(c) \text{ and } c \in co(e)\}$. To reduce the number of vectors in $free(e)$ we use density based clustering, replacing the points with the centroids of each resulting cluster.

After the optimization step, we need to update the free-space points relative to all the output elements. We derive the free-space points from the relative offset of the free-space vectors from their respective elements. We re-build the set of all free space positions F in the output domain by offsetting all of the free space vectors for all elements by the position of those elements:

$$F = \{m_{p_o}(v) + p_o | v \in free(o_e) \text{ and } o \in O\}. \quad (3)$$

If a free-space point $v \in F$ overlaps with an element at p_v we remove v from F. As an implementation detail, we keep track of free space positions in a k-d tree to enable efficient queries.

5.2 Example-palette

The example-palette is a set of elements. The user can manually design element arrangements or use our interactive tools to synthesize elements from the example-palette back into the example-palette. We take the user's selection from the example-palette and pass it as input to our region-growing algorithm to synthesize new elements.

Scenes composed of elements are often composed of elements at different scales, for example, the groves of trees and stone walkways in Figure 1. Therefore, we capture these different scales by allowing different element arrangements to use different neighborhood radii. If the neighborhood radius is too small, it will not capture the features of an element arrangement. Likewise, if it is too large, it can lead to slow rates of synthesis.

Each element $e \in \mathcal{E}$ has a neighbourhood radius attribute $\bar{r}_e$. When designing an example arrangement, the user can assign the neighborhood radius to each element in the arrangement. For example, the cobble-stone and trees in Figure 1 have different neighborhood sizes.

The synthesized elements must derive from the example-palette selection. Therefore, the neighborhoods that we search through for new elements must be limited to the example-palette selection. Therefore, we recalculate the k-coherent neighborhoods of elements in an example-palette selection every time it changes.

The user can select elements from multiple textures in the example-palette. The user can even select subsets of elements from different textures. This makes the example-palette selection a versatile tool for combining features from different discrete element textures into a new texture.

5.3 Optimization

The optimization step (Figure 6c) reduces the energy of previously synthesized elements in the horizon relative to the example-palette selection $E(\mathcal{H}, \mathcal{S})$. The goal is to arrange the elements in $\mathcal{H}$ so that the neighborhood of each horizon element aligns as closely as possible with the similar example-palette selection neighborhoods. Each of these corresponding exemplar neighborhoods provides predicted positions for the elements in the output domain. Each element in the horizon will have multiple predictions. Inspired by Ma et al. [15], we find new positions for the elements in the horizon from their predictions using a least-squares method.

For each $h \in \mathcal{H}$ we find the nearest example-palette selection element $e = nearest(h, \mathcal{S})$. The full pair assignments $pairs_f(h, e) = \{(h', e')\}$ provide us with a direction vector $\hat{p}(h, h') = p_{e'} - p_e$. This direction vector, is the direction between h' and h as predicted by the exemplar neighbourhood.

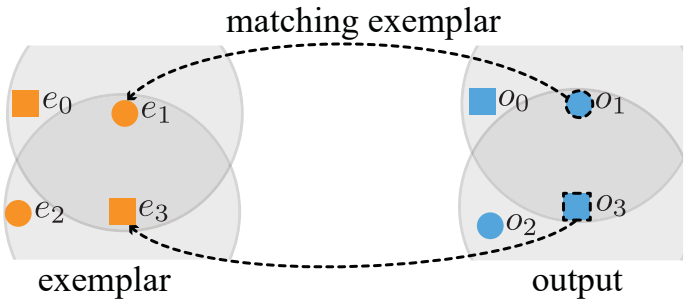


Figure 7: In this example, the nearest exemplar selection element to o_1 is found to be e_1 and the nearest exemplar selection element to o_3 is e_3. Therefore, we have the following pairings: $pairs_f(o_1, e_1) = \{(o_0, e_0), (o_3, e_3)\}$ and $pairs_f(o_3, e_3) = \{(o_2, e_2), (o_1, e_1)\}$.

We motivate the formation of a system of equations for optimizing element arrangements with the example in Figure 7. We can form a system of equations from the pairings $pairs_f(o_1, e_1) = \{(o_0, e_0), (o_3, e_3)\}$ and $pairs_f(o_3, e_3) = \{(o_2, e_2), (o_1, e_1)\}$ between the example and output neighbourhoods:

$$\begin{aligned} & Ax = b \\ \hline & p_{o,0} - p_{o,1} = p_{e,0} - p_{e,1} \\ & p_{o,3} - p_{o,1} = p_{e,3} - p_{e,1} \\ & p_{o,2} - p_{o,3} = p_{e,2} - p_{e,3} \\ & p_{o,1} - p_{o,3} = p_{e,1} - p_{e,3}. \end{aligned} \quad (4)$$

The right side of the system of equations forms a vector b. The left side forms a matrix A and unknowns $x = (p_{o,0}, p_{o,1}, p_{o,2}, p_{o,3})^T$. The system $A\ x\ =\ b$ for an output domain and exemplar is a positive definite sparse linear system that can be solved quickly with a Sparse Cholesky Decomposition—we use an implementation from the Eigen matrix library (Guennebaud et al. [36]).

5.4 Prediction vectors

A tug-of-war occurs when two neighborhoods provide two different prediction vectors for the position of an element. Both predictions will have equal weights in the system of equations, and we will end up with a position that is the average of the two. Ma et al. [15] solve this problem by weighting the prediction vectors by their length, so predictions closer to the original position will have more weight. The disadvantage of this approach is that very long prediction vectors will have a low weight in the system of equations (even if they are a good prediction). It will take more optimizations steps to move those elements to a minimized position.

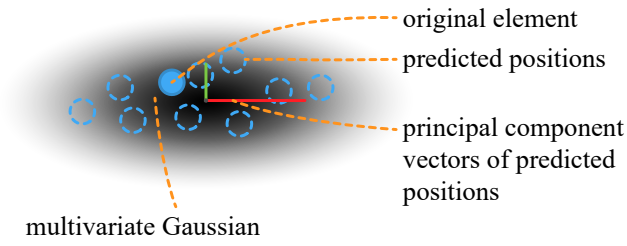


Figure 8: Prediction vectors for an element are weighted by a multivariate Gaussian distribution (the gray function in the background) for the set of predictions. The basis for this Gaussian function is found through principal component analysis on the set of prediction vectors.

We solve this problem by considering the distribution of prediction vectors in the neighborhood of $h \in \mathcal{H}$ (Figure 8). Principal component analysis on the prediction vectors in a neighborhood gives us three orthogonal vectors that along with the center of the distribution, we can use to weight the prediction vectors. Specifically, we use a multivariate Gaussian function to weight the prediction vectors in a neighborhood. If $\bar{X}$ is the mean of the prediction vectors for h and $C = \bar{X}^T \times \bar{X} * 1/m$, the weight given to a prediction vector $\bar{p}$ is

$$exp(\frac{\bar{p}^T \times C^{-1} \times \bar{p}}{-2}). \quad (5)$$

Since we do optimization on the horizon and not on the entire output domain, we have to be careful to maintain coherence with output elements that are not in the horizon. Therefore, in the linear optimization problem, we give low weight to predicted positions for elements not in the horizon while also giving their previous positions a high weight. This allows for old elements some movement but otherwise constrains the prediction vectors to work on horizon elements.

5.5 Sketch-based modelling

The generative-brush generates element arrangements in a radius around the brush-points of a brush-path. We represent brush-strokes across a 3D surface in our system as a series of brush-points composed of a position and radius (bp_i, br_i). The user controls the brush radius with a slider. We use ray-casting to map the brush-points from screen-space onto the surface.

The generative-brush adds brush-points to a set B. This set constrains the generation step (Figure 6b) to nearby brush-points (Figure 3a). To limit generation to the brush strokes, we discard free-space points that do not overlap with any of the brush-points in B. This has the effect of limiting generation to the area around the brush point. If there are no elements near the first brush-point, a random patch from the example-palette selection is copied to the output domain to seed synthesis. We remove brush points when there are no free-space points that overlap with it. The eraser tool does the opposite; it removes overlapping brush-points from B and any nearby overlapping elements (Figure 3d).

We also use the generative-brush an optimization tool to improve the appearance of the output domain after synthesis relative to the example-palette selection (Figure 3a). Rather than creating new elements, the overlapping elements are returned to the horizon, where the next optimization step will affect them. The tool also has exciting applications for designing new textures in the example-palette (Figure 9e).

The filler tool triggers region-growing to synthesize new elements at a point selected by the user (Figure 3d). It does this by removing all brush-points from B and disabling the free-space overlap checks with elements in B. If there are no elements nearby the selected point a random patch of elements from the example-palette selection is copied to the output domain at that point to seed generation.

In Figure 9, we demonstrate some of the geometry and texture synthesis applications of our system with a **stretch-tool**. The stretch tool deforms the underlying geometry and removes any affected elements. Next, we use the filler tool to synthesize the affected region of the surface with the current example-palette selection.

5.6 Filling voids in the exemplar

Many arrangements that one might reproduce with our system may contain voids. However, our region-growing algorithm will often select elements to fill those empty spaces. Our solution is to fill space in the example-palette with invisible elements. The benefit of this idea is that it can reuse the existing machinery for element synthesis while giving the user fine-grained control over what should and should not be empty in the example.

To fill space in the exemplar, the user places invisible elements. The user can use manual placement or the generative-brush with a source texture of invisible elements. During this process, we toggle invisible elements to a visible state.

5.7 Surface synthesis

In planar synthesis, prediction vectors tell us where to place a given element on the plane, but in surface synthesis, the prediction vectors tell us in which direction to 'walk' on the surface of a mesh.

We use Crane et al. [35] to find an orientation for the output domain mesh. Crane et al.'s method is closely related to Ray et al.'s [37]. The orientation field can be automatically computed, or the user can design it by placing singularity points on the mesh as described by Crane et al. [35]. Once the orientation field has been created (one second or less), the user can

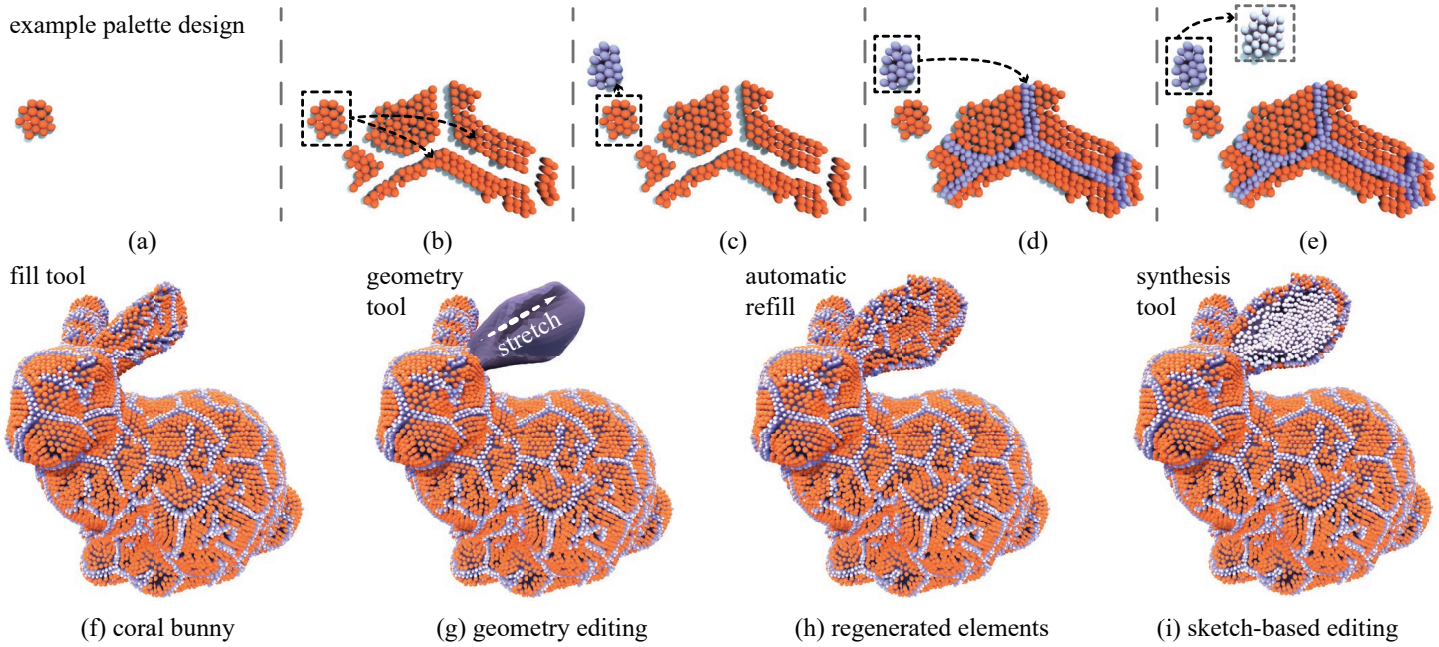


Figure 9: (a) Example-palette design: a small example arrangement is created by manually placing elements. (b) The example-palette selection together with the generative-brush is used to generate a new more complicated texture indicated by the arrows. (c) A new arrangement is introduced. (d) The new arrangement is used to fill in the gaps. (f) The filler tool is applied to the Stanford bunny whose ear is subsequently stretched (g), erasing the affected elements. (h) The elements on the ear are regenerated. (i) The eraser and generative-brush are used to replace the ear elements with white elements from the example-palette (e). The supplementary material contains a video showing the interactive design of the bunny (filename:sketching.mp4).

start generating discrete element textures for that surface.

For an element $h \in \mathcal{H}$ and a prediction vector $\bar{p}$ in its neighbourhood, p_h is already on the surface of the mesh. We look up the orientation at that point $o(p_h)$ and walk along the integral curve on the surface passing through p_h and in the world direction $o(p_h) \times \bar{p}$ until we have travelled $|\bar{p}|$ units along that curve.

To compare an exemplar and output neighborhood, the exemplar neighborhood positions are mapped onto the surface using an integral curve mapping, as described above. The pairing process and distance metric work the same as before, using the mapped positions.

Like Wei and Levoy [4], we perform synthesis directly over an arbitrary manifold surface, avoiding discontinuities. Unlike Wei et al., we do not compute the orientation field as we go. Instead, we allow the user to design it first (if they choose) by placing a few singularity points.

5.8 View based synthesis

In a 3D scene, many of the components of that scene may be occluded from view (Figure 10). Therefore, during synthesis, we focus on just those regions that are visible. We maintain two queues, one with elements visible to the camera and another with elements occluded by geometry in the scene. We prioritize synthesis to elements in the visible queue. In our implementation, we make use of the GPU's depth buffer to test whether a free space point is occluded or not. If a free-space point is not visible, the generation step can skip generating a new element from that free-space point (Section 5.1).

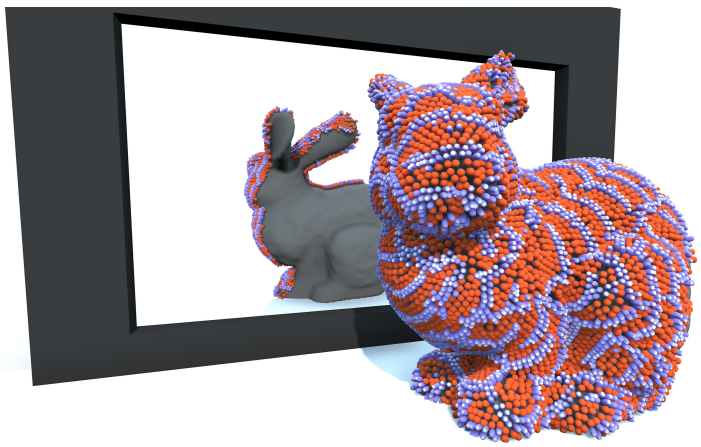


Figure 10: Elements that are not occluded are generated with priority over those that are occluded. Here one can see the backside of the bunny in the mirror does not have many elements generated yet.

However, if all of the visible free-space points have generated elements, then we can take advantage of the freed up CPU resources to generate elements for occluded free-space points. If the camera changes its vantage point, then this scheme may have already generated elements for those newly revealed regions.

6 Results and Discussion

Previous offline processes Ma et al. [15] and Landes et al. [20] can generate hundreds of elements with running times in the minutes. Our method can generate thousands of elements per second. Furthermore, we demonstrate results that are not possible with Ma et al.'s patch initialization scheme (Section 6.1).

Ijiri et al. [13] employ a fast region-growing method that relies on regularity in the topology of a network of elements (found through Delaunay triangulation). However, their method is limited to textures expressible with 1-ring neighborhoods and that has the required topological regularity. In contrast, we do not rely on regularity in the example texture nor do we try to maintain a network topology on the generated elements. Instead, we use geometric neighborhoods of arbitrary size and a closest-point algorithm to find pairs of elements between the example and output neighborhoods (this avoids the need for topological regularity and a network of elements). As such, we support textures that can only be expressed with greater than 1-ring neighborhoods. We demonstrate synthesis with 3-ring neighborhoods —larger neighborhoods can be used but at the cost of interactive rates of synthesis.

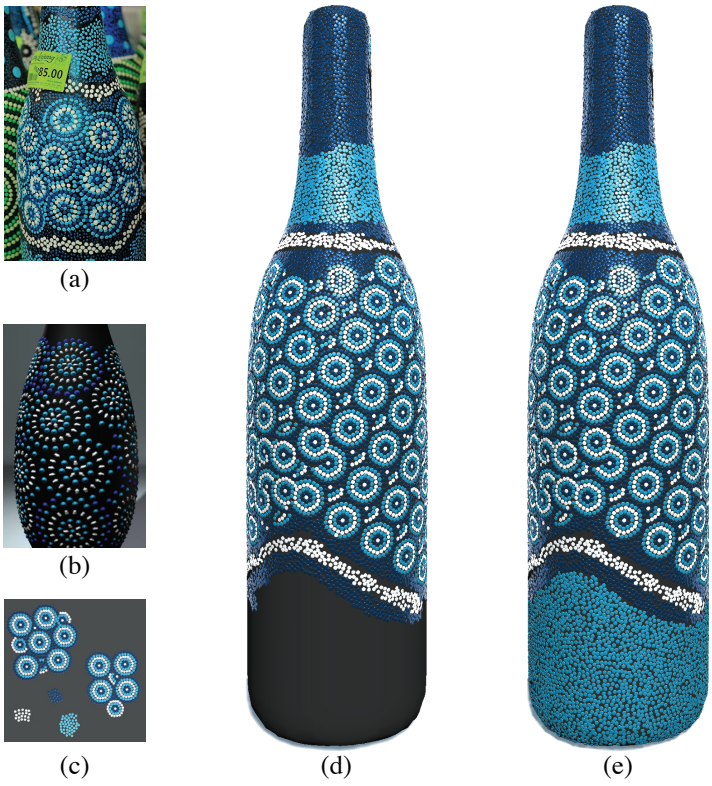


Figure 11: (a) Photograph of glass tilings on a bottle. (b) Ma et al. [15] use their system to adjust element properties on a manullay created arrangement of elements. (e) We fully reproduce the photo from (a) using our interactive sketch based system. We used the example-palette in (c). (d) an under construction view.

Emilien et al. [22] have a similar example-palette idea for synthesizing distributions of elements on virtual-world terrains. However, they apply techniques from statistical synthesis rather than example-based texture synthesis. Their system learns the spatial distribution of points with a histogram and uses Metropolis-Hastings sampling (Geyer and Møller [38] and Hurtut et al. [19]) together with the histogram to synthesize new distributions. For example, this method is well suited for synthesizing distributions with varying density. However, the histogram is insufficient for capturing semantic relationships between elements. In contrast, our method is based on discrete element texture synthesis and can synthesize discrete element textures with structure and semantic relationships between elements in the exemplar. For example, synthesizing the glass tilings in Figure 11 would be impossible with Emilien et al.'s statistical synthesis method. Our method is not intended for only synthesizing stochastic arrangements of elements, but also discrete element textures that have structure and patterns—such as we commonly find in human creations. We imagine that a hybrid system with both discrete element texture synthesis and statistical synthesis would complement the strengths and weaknesses of either approach for constructing virtual worlds.

Our system has applications for the design of virtual worlds, such as those used in film or video games. We have designed a scenario with a large example-palette and a large world. Our bunny-planet is composed of 30 element types (Figure 1). In the final scene, there are 24,063 elements, which took 45 minutes to design. Rates of synthesis vary between 2,050 elements/s (the groves of trees) to 3,200 elements/s (stone walk-way). To accommodate differences in scale—such as between trees and flowers—we use different element neighborhood radii ($\hat{r}_e$).

Glass tilings are one of the many examples of element arrangements found in our world. Ma et al. [15] demonstrate how their method could be used to change the attributes of an intricate glass tiling inspired by a photograph (Figure 11a). However, with evidence from Section 6.1 we suspect that their method (using patch-initialization) cannot be used to produce the initial arrangement of elements inspired by that photo. We have reproduced all of the different element arrangements in that photo using our interactive tools (Figure 11). Our bottle contains 21,806 elements synthesized at an average rate of 1,100 elements/s. It took about 30 minutes to design our glass-tiled bottle. We capture the circle patterns using 3-ring neighborhoods. There are 1,061 elements in the example-palette.

Our system also has application for modeling. We demonstrate this with a Stanford coral-bunny (Figure 9). An artist may deform the mesh, such as by stretching the bunny ears. We search through the affected triangles and gather all of the elements on those triangles. We discard those elements and regenerate free-space points for the neighboring elements, which triggers region-growing in the affected region. We use 2-ring neighborhoods and achieve about 2,100 elements/s. In this example, we also demonstrate example-palette design using the generative brush.

6.1 Analysis

In this section we explore the impact that different optimization and generation schemes have on the empirical quality

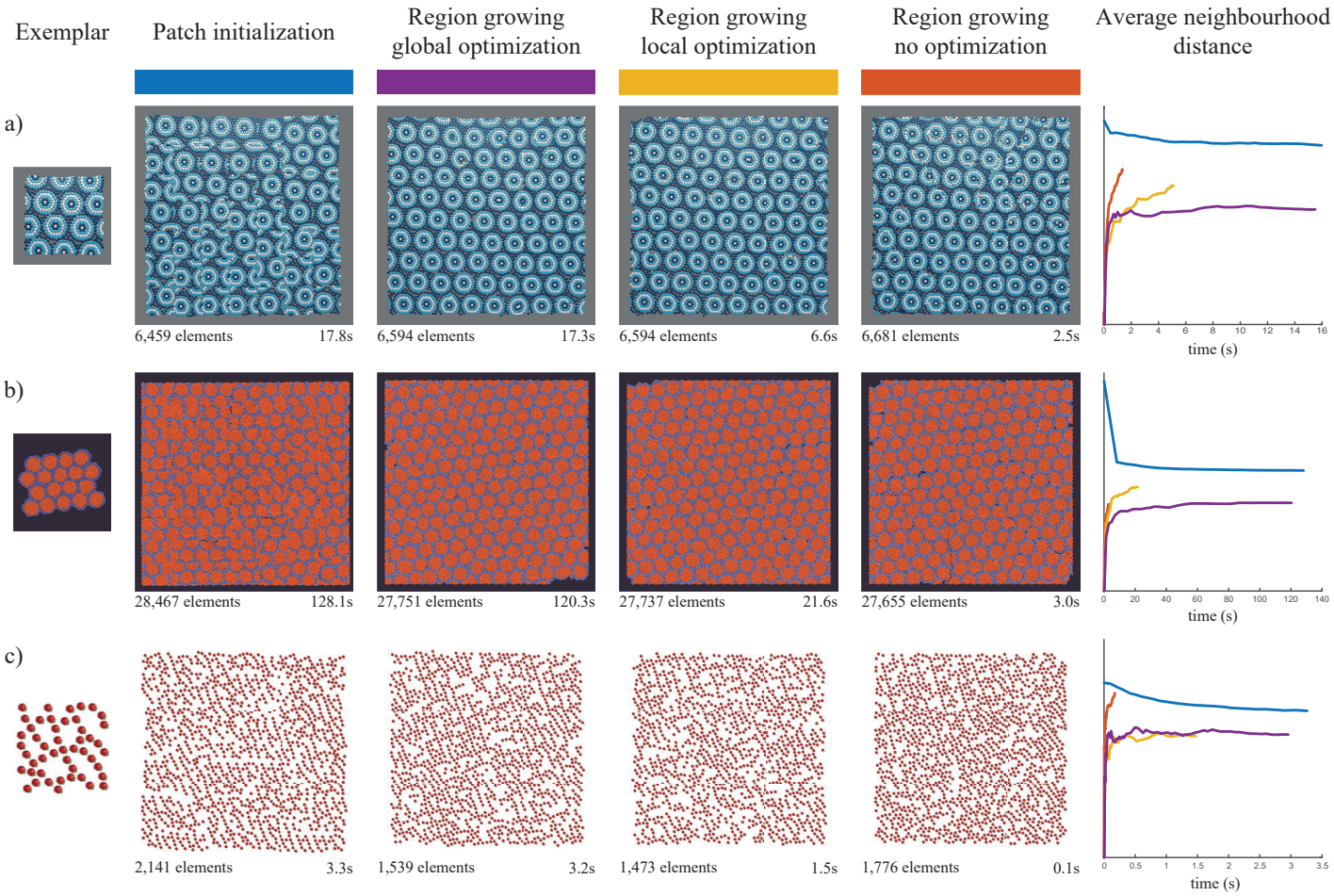


Figure 12: A comparison of Ma et al.'s Ma et al. [15] patch-initialization as implemented in our system versus global, local and no optimization with region-growing. The exemplar for each row is given on the left, and a plot of the average neighborhood similarity (Equation 6.1) for each row is given in the right column. In the bottom of each screenshot, we give the final number of elements generated along with how long generation took or how long the algorithm was given to run, in the case of patch initialization. Lower values in the plot are better; lower values mean that the output is more similar to the exemplar.

of the generated output domain (Figure 12). In general, we find that region-growing alone generates results superior to Ma et al.'s [15] patch-initialization. When coupled with local optimization of horizon elements or global optimization on all elements, the empirical quality of the results is further improved.

To evaluate our method against Ma et al. [15]'s, we implemented their patch-initialization scheme in our system but used our optimization step. Patch-initialization as described by Ma et al. [15] divides the example-palette and output domain into a grid of fixed size cells. Cells are selected randomly from the example-palette and copied to the empty cells in the output domain. Our goal with this evaluation is to determine what impact region-growing and-or optimization has on the quality of the output.

To measure the empirical quality of an output arrangement of elements relative to an example-palette, we track average neighborhood distance:

$$\frac{\sum_{o \in \mathcal{O}} |n(o) - n(e)|}{|\mathcal{O}|}. \tag{6}$$

In a perfect system, we expect the average neighborhood distance to remain constant and not to increase with the addition of the elements. Tracking average neighborhood similarity with respect to time, allows us to compare patch-initialization (where all elements are generated beforehand) to region-growing (where elements are generated incrementally over time).

In our experimental setup, we compare patch-initialization, region-growing with global optimization, region-growing with local optimization and region-growing without any optimization on three different exemplars. We restrict each output domain to a fixed region. We terminate the region-growing variants when the supplied region is filled. Patch-initialization is given as much time to run as the longest-running region-growing variant.

We choose the three exemplars in Figure 12 to include a complex semi-repeating arrangement of elements (the glass tilings in Figure 12a) to a more stochastic arrangement of elements derived from an example by Ma et al. [15] (the apple's in Figure 12c). Artifacts are left present in all of the following results for comparison purposes—however, these can

be easily corrected using our sketch-based tools.

We found that for each exemplar, patch-initialization starts with a high average neighborhood distance that improves and levels off. In all cases, the final average neighborhood distance is significantly higher than the region-growing variants. We attribute this to the way in which optimization works; it is essentially a blending of greedy predictions that can get stuck in local optima. Therefore, a good starting configuration is essential to achieve a low final average neighborhood distance. Patch-initialization can make abysmal choices (random) that are impossible for the optimization steps to correct (it gets stuck in local optima). Furthermore, patch-initialization is very sensitive to the size of cells dividing the exemplar and the output domain. We use regular grids, which cannot capture the underlying patterns found in many of our exemplars. Perhaps irregular cells could achieve this, but then, so can our simple region-growing scheme.

Meanwhile, region-growing adds new elements before each optimization step, using greedy choices. Therefore, the starting configuration for optimization is better (it is not random), and so the local minima that optimization converges on should also be better. In all of the region-growing variants, the average neighborhood distance starts at zero and increases over time as new elements are added.

Initially, region-growing can make some perfect decisions. The first perfect decision is to copy a small seed patch from the exemplar, the average neighborhood distance of that neighborhood is zero. For a perfectly regular exemplar, we would expect to continue finding perfect choices. However, for the exemplars that we choose we eventually run out of perfect choices and so the average neighborhood distance increases. The dips in average neighborhood distance are the result of local choices in the region-growing step. Some sort of generation algorithm that made global choices could avoid this problem.

In the examples that we explored, we observe that the average neighborhood distance for region-growing (without optimization) had not leveled off yet (the orange plots in Figure 12). Poor choices accumulate, eventually the coherence between neighboring elements becomes low, and so the choices as found by neighborhood matching alone become poor. For other settings (with optimization) an equilibrium was eventually found sooner, and the average neighborhood distance leveled off.

Optimization helps prevent the accumulation of poor choices and prevents a loss in coherence between neighboring elements. We see this in the leveling off of the local and global optimization curves. However, as we observe in Figure 12a and 12b, local optimization leads to higher average neighborhood similarities than does global optimization. We attribute this to 'shearing' artifacts between the horizon elements and 'frozen' elements, where predictions are only generated for the horizon elements, and their positions as a whole are shifted relative to the frozen elements. To combat this, we include frozen elements within a constant distance of the horizon elements, in the optimization problem.

The last exemplar (Figure 12c) is interesting. Local optimization produced the lowest final average neighborhood distance. We suspect that global optimization was overfitting to a single region of the example-palette. Indeed, in some of our experiments, we found that letting global optimization run for too long can produce very long straight artifacts running through the entire output domain. From the narrow and greedy view of the optimization problem, those long straight neighborhoods are the best fit in many situations.

Our conclusion from this analysis is that patch-initialization can produce poor initial configurations that are impossible for our optimization step to recover. Furthermore, patch-initialization must be aligned to features in the exemplar, which is not possible with all exemplars (such as in Figure 12). An interleaved region-growing and optimization strategy consistently produces arrangements that have good average neighborhood distance. This analysis demonstrates a significant improvement in quality over Ma et al. [15] with our interleaved generation and optimization method.

We demonstrate the performance and the quality of our method in comparison to patch-based initialization with global optimization in Figure 12. To obtain these results, we added a random patch-copy initialization scheme as described in Ma et al. [15] to our implementation. Next, we compare the results achieved by our method using region-growing alone, region-growing with global optimization on all elements synthesized so far, and region-growing with horizon-based optimization.

7 Conclusion and future work

Our interactive discrete element texture palettes enable the construction of both stochastic and structured element arrangements. We demonstrate results that are a significant improvement over previous results but also results that are not possible with those systems (Ijiri et al. [13], Ma et al. [15]). While Ma et al. [15] measure their results with hundreds of elements per minute, we measure our results in thousands of elements per second. These speeds for discrete element textures are comparable to other state-of-the-art techniques for statistical synthesis [22]. Our example-palette enables the construction of scenes composed of a variety of element arrangements. Our fast region-growing algorithm enables interactive rates of synthesis suitable for sketch-based modeling. The key to our interactive rates of synthesis is a technique for efficient pruning of the region-growing horizon.

We have implemented a variety of sketch-based tools, including our generative-brush that can be used for both synthesizing new elements but also for relaxing previously synthesized results. However, there are many more tools that we would like to develop, such as a context-aware eraser to easily erase around structures in an element arrangement.

Nevertheless, there are some limitations with our system. For example, significant differences in element size are not supported by our neighborhood matching scheme. A multi-scale approach would be required to reconcile elements and arrangements at different scales.

A limitation with neighborhood matching is complexity. The more elements there are in a neighborhood, the more expensive the computation. We demonstrate up to 3-ring neighborhoods in Figure 11, but beyond this, our system would lose interactivity.

Our system is limited to generating structured element ar-

rangements that can be captured by the radius used during neighborhood matching. However, there are many structured arrangements with much more extensive features that we would like to synthesize.

We suspect that pair-wise similarity measures are perhaps an insufficient measure for element arrangement similarity. In our framework, there are issues when pairs cannot be found, or an over the assignment of pairings is possible. Pair-wise similarity does not capture the perceptual similarity of element arrangements in an intuitive way. Therefore, we think there is promise in addressing some of the limitations in recent approaches, such as Roveri et al. [18] that reconsider element arrangements in a continuous domain.

Finally, we applied a local strategy for generating elements. A global strategy for generating new elements could avoid some of the artifacts that arose in our results from local decisions made by our region-growing algorithm.

References

[1] Wei, LY, Lefebvre, S, Kwatra, V, Turk, G. State of the art in example-based texture synthesis. In: Eurographics 2009, State of the Art Report, EG-STAR. Eurographics Association; 2009, p. 93–117.

[2] Efros, AA, Leung, TK. Texture synthesis by non-parametric sampling. In: Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on; vol. 2. IEEE; 1999, p. 1033–1038.

[3] Wei, LY, Levoy, M. Fast texture synthesis using tree-structured vector quantization. In: Proceedings of the 27th annual conference on Computer graphics and interactive techniques. ACM Press/Addison-Wesley Publishing Co.; 2000, p. 479–488.

[4] Wei, LY, Levoy, M. Texture synthesis over arbitrary manifold surfaces. In: Proceedings of the 28th annual conference on Computer graphics and interactive techniques. ACM; 2001, p. 355–360.

[5] Ashikhmin, M. Synthesizing natural textures. In: Proceedings of the 2001 symposium on Interactive 3D graphics. ACM; 2001, p. 217–226.

[6] Tong, X, Zhang, J, Liu, L, Wang, X, Guo, B, Shum, HY. Synthesis of bidirectional texture functions on arbitrary surfaces. In: ACM Transactions on Graphics (TOG); vol. 21. ACM; 2002, p. 665–672.

[7] Kwatra, V, Essa, I, Bobick, A, Kwatra, N. Texture optimization for example-based synthesis. In: ACM Transactions on Graphics (TOG); vol. 24. ACM; 2005, p. 795–802.

[8] Han, J, Zhou, K, Wei, LY, Gong, M, Bao, H, Zhang, X, et al. Fast example-based surface texture synthesis via discrete optimization. The Visual Computer 2006;22(9-11):918–925.

[9] Cohen, MF, Shade, J, Hiller, S, Deussen, O. Wang tiles for image and texture generation; vol. 22. ACM; 2003.

[10] Li, C, Wand, M. Precomputed real-time texture synthesis with markovian generative adversarial networks. In: European Conference on Computer Vision. Springer; 2016, p. 702–716.

[11] Han, C, Risser, E, Ramamoorthi, R, Grinspun, E. Multiscale texture synthesis. In: ACM Transactions on Graphics (TOG); vol. 27. ACM; 2008, p. 51.

[12] Vanhoey, K, Sauvage, B, Larue, F, Dischler, JM. On-the-fly multi-scale infinite texturing from example. ACM Transactions on Graphics (TOG) 2013;32(6):208.

[13] Ijiri, T, Mech, R, Igarashi, T, Miller, G. An example-based procedural system for element arrangement. In: Computer Graphics Forum; vol. 27. Wiley Online Library; 2008, p. 429–436.

[14] Barla, P, Breslav, S, Thollot, J, Sillion, F, Markosian, L. Stroke pattern analysis and synthesis. In: Computer Graphics Forum; vol. 25. Wiley Online Library; 2006, p. 663–671.

[15] Ma, C, Wei, LY, Tong, X. Discrete element textures. In: ACM Transactions on Graphics (TOG); vol. 30. ACM; 2011, p. 62.

[16] Ma, C, Wei, LY, Lefebvre, S, Tong, X. Dynamic element textures. ACM Transactions on Graphics (TOG) 2013;32(4):90.

[17] Xing, J, Chen, HT, Wei, LY. Autocomplete painting repetitions. ACM Transactions on Graphics (TOG) 2014;33(6):172.

[18] Roveri, R, Öztireli, AC, Martin, S, Solenthaler, B, Gross, M. Example based repetitive structure synthesis. In: Computer Graphics Forum; vol. 34. Wiley Online Library; 2015, p. 39–52.

[19] Hurtut, T, Landes, PE, Thollot, J, Gousseau, Y, Drouillhet, R, Coeurjolly, JF. Appearance-guided synthesis of element arrangements by example. In: Proceedings of the 7th International Symposium on Non-Photorealistic Animation and Rendering. ACM; 2009, p. 51–60.

[20] Landes, PE, Galerne, B, Hurtut, T. A shape-aware model for discrete texture synthesis. In: Computer Graphics Forum; vol. 32. Wiley Online Library; 2013, p. 67–76.

[21] Roveri, R, Öztireli, AC, Gross, M. General point sampling with adaptive density and correlations. In: Computer Graphics Forum; vol. 36. Wiley Online Library; 2017, p. 107–117.

[22] Emilien, A, Vimont, U, Cani, MP, Poulin, P, Benes, B. Worldbrush: Interactive example-based synthesis of procedural virtual worlds. ACM Trans Graph 2015;34(4):106:1–106:11.

[23] Gain, J, Long, H, Cordonnier, G, Cani, MP. Ecobrush: Interactive control of visually consistent large-scale ecosystems. In: Computer Graphics Forum; vol. 36. Wiley Online Library; 2017, p. 63–73.

[24] Bhat, P, Ingram, S, Turk, G. Geometric texture synthesis by example. In: Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing. ACM; 2004, p. 41–44.

[25] Zhou, K, Huang, X, Wang, X, Tong, Y, Desbrun, M, Guo, B, et al. Mesh quilting for geometric texture synthesis. ACM Transactions on Graphics (TOG) 2006;25(3):690–697.

[26] Yuksel, C, Kaldor, JM, James, DL, Marschner, S. Stitch meshes for modeling knitted clothing with yarn-level detail. ACM Transactions on Graphics (TOG) 2012;31(4):37.

[27] Merrell, P, Manocha, D. Model synthesis: a general procedural modeling algorithm. Visualization and Computer Graphics, IEEE Transactions on 2011;17(6):715–728.

[28] Peytavie, A, Galin, E, Grosjean, J, Mérillou, S. Procedural generation of rock piles using aperiodic tiling. In: Computer Graphics Forum; vol. 28. Wiley Online Library; 2009, p. 1801–1809.

[29] Bokeloh, M, Wand, M, Seidel, HP. A connection between partial symmetry and inverse procedural modeling. In: ACM Transactions on Graphics (TOG); vol. 29. ACM; 2010, p. 104.

[30] Runions, A, Lane, B, Prusinkiewicz, P. Modeling trees with a space colonization algorithm. NPH 2007;7:63–70.

[31] Palubicki, W, Horel, K, Longay, S, Runions, A, Lane, B, Měch, R, et al. Self-organizing tree models for image synthesis. ACM Transactions on Graphics (TOG) 2009;28(3):58.

[32] Li, Y, Bao, F, Zhang, E, Kobayashi, Y, Wonka, P. Geometry synthesis on surfaces using field-guided shape grammars. Visualization and Computer Graphics, IEEE Transactions on 2011;17(2):231–243.

[33] Lu, J, Barnes, C, Wan, C, Asente, P, Mech, R, Finkelstein, A. Decobrush: drawing structured decorative patterns by example. ACM Transactions on Graphics (TOG) 2014;33(4):90.

[34] Guerrero, P, Bernstein, G, Li, W, Mitra, NJ. Patex: exploring pattern variations. ACM Transactions on Graphics (TOG) 2016;35(4):48.

[35] Crane, K, Desbrun, M, Schröder, P. Trivial connections on discrete surfaces. In: Computer Graphics Forum; vol. 29. Wiley Online Library; 2010, p. 1525–1533.

[36] Guennebaud, G, Jacob, B, et al. Eigen v3. http://eigen.tuxfamily.org; 2010.

[37] Ray, N, Vallet, B, Li, WC, Lévy, B. N-symmetry direction field design. ACM Trans Graph 2008;27(2):10:1–10:13.

[38] Geyer, CJ, Møller, J. Simulation procedures and likelihood inference for spatial point processes. Scandinavian Journal of Statistics 1994;:359–373.