Interactive example-palettes for discrete element texture synthesis

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

Preprint submitted for review/Computers & Graphics (2018)

Abstract

Textures composed of individual discrete elements are found in everything from human-made glass-tilings to forests and tropical coral. We propose an interactive sketch-based system for synthesizing scenes consisting of many discrete element textures. We have implemented an example-palette, a design window where a user can use our sketch-based tools to create discrete element textures and then paint those textures into a scene or back into the example-palette to create new textures. Our interactive sketch-based tools use a new and fast region-growing algorithm that iteratively synthesizes new elements around previously synthesized elements. To support discrete element textures with different scales in the same output, we parameterize our region-growing algorithm on a per-element basis. Our method is capable of synthesizing structured and stochastic example discrete element textures. We explore applications of our system for building virtual worlds (such as for video games) and for sketch-based modeling.

1 Introduction

The 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. Synthesizing virtual worlds composed of discrete elements is an important problem and has applications for video games and film. Our goal is to synthesize these worlds interactively.

Consider the problem of building a hypothetical bunny-planet for a children's fantasy computer game (Figure 1). In our system, we paint cobblestone, forests, crops, meadows, and villages on this planet with an interactive sketch-based system for example-based discrete element texture synthesis. Our system supports multiple examples, at different scales, in a palette.

One challenge for synthesizing virtual worlds is to keep the human in the loop to guide synthesis while removing the tedious task of manual element placement. 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. Such an algorithm must be fast for real-time interactive applications, which is one of the main problems we set out to solve. Another challenge is how to interactively design complex texture exemplars.

Our idea is an interactive, sketch-based discrete element texture synthesis system for designing textures on-the-fly in an example palette. Then, we interactively paints those textures on surfaces in a 3D virtual world (Figure 2). We have developed a fast region-growing algorithm for discrete element texture synthesis that powers our interactive sketch-based generative brushes. In our bunny-planet scenario, we create an example-palette containing different discrete textures using our generative brushes. When one selects an example, like the mushrooms, and sketches that discrete element texture into the scene, our region-growing algorithm synthesizes elements along the brush path in real-time. Examples from the palette can also be used to paint new examples into the palette (Figure 10). With our interactive system, one can create complicated and intricate virtual worlds and objects with little effort.

The example-palette is a design window for arranging the example texture elements. It would be reasonably simple to add an example-palette, with multiple textures, to other systems for discrete element texture synthesis. However, one challenge is how to handle discrete textures at different scales when applying those textures to the same output. In our solution, we parameterize our synthesis algorithm on a per-element basis (Section 4)—in single example systems like Roveri et al. [1] and Ma et al. [2] a global set of parameters is used. One of the most important per-element parameters in our system is neighborhood size. It should be large enough to capture a repetitive pattern in the example texture (an example of different neighborhood sizes are the cobblestone and trees in Figure 3).

Our main contribution is a fast **region-growing algorithm** that iteratively synthesizes new elements based on previously synthesized elements (Section 5). We introduce a method to limit size of the active-problem (Section 5.1). Our fast method enables interactive applications for discrete element texture synthesis, which we demonstrate with an interactive sketch-based system. Our **sketch-based tools** (Section 5.4) guide our region-growing algorithm to synthesize new elements in the example-palette or directly on **3D surfaces** (Section 5.7) without having to worry about surface parameterizations or texture mapping. We also demonstrate applications for sketch-based modeling (Section 5.4). A happy consequence of our interactivity and sketch-based tools is an **example-palette** that allows on the fly design and synthesis of complex discrete texture examples (Section 5.5). The parameters for synthesis are configured on a per-element basis in the example-palette and copied to elements in the output during synthesis, this helps us sketch scenes with discrete element textures that have different scales. We analyze our method in comparison to previous methods in Section 6.

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. This process took about 45 minutes. (top right) A whole view of the bunny-planet. The supplementary material contains a video of the bunny-planet design (filename: bunny_planet.mp4).

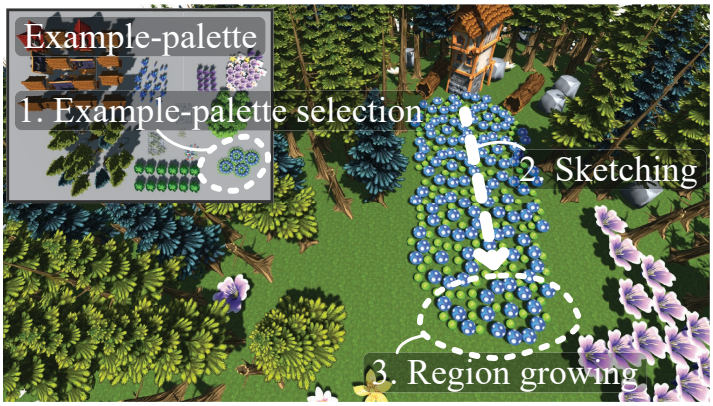


Figure 2: Interactive synthesis of a mushroom garden. 1) The user selects a mushroom garden discrete element texture from the example-palette. 2) The user sketches the garden onto the surface. 3) Our fast region-growing algorithm iteratively synthesizes new elements in the highlighed area as the user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

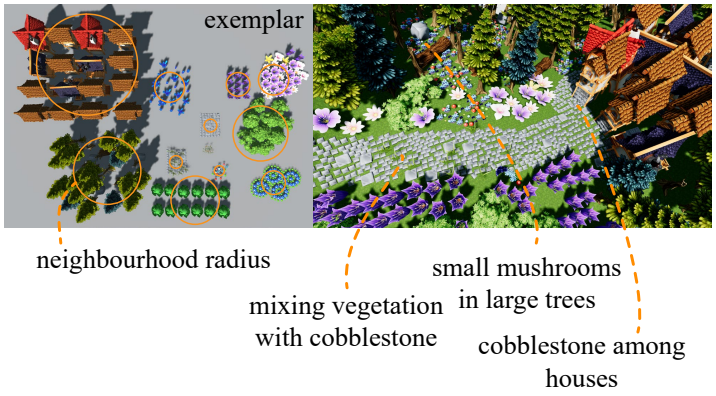


Figure 3: (left) Each example in the example-palette has different parameters for discrete element texture synthesis. Here, the neighborhood radius parameter illustrates the feature size that will be synthesized. (right) Per-element parameters enable us to synthesize scenes at multiple scales. Notice the scale of the discrete mushroom texture compared to the scale of the discrete tree texture.

2 Related work

Pixel-based texture synthesis: Mosaic models (Schachter and Ahuja [3]) and coherent noise (Perlin [4]) were two of the early methods to randomly synthesize pixel-based textures. In example-based texture synthesis, a 2D example image is used to synthesize a large scale texture (Wei et al. [5]) that is perceptually similar to the example (Tamura et al. [6]). Pixel-based approaches commonly choose pixels to add upon neighborhood comparisons between the previously synthesized pixels and example pixels (Efros and Leung [7] and Wei and Levoy [8]). Wei and Levoy [9] synthesize textures over arbitrary manifold surfaces. Later works reduced the number of neighborhood comparisons making synthesis much faster (Ashikhmin [10] and Tong et al. [11]). Dischler et al. [12] decompose an input texture into some representative texture particles that are duplicated and recombined to create larger textures. Kwatra et al. [13] reframe the problem of texture synthesis as one of energy minimization, while Han

et al. [14] build on this with a fast and discrete solver. Cohen et al. [15] use Wang tiles as a fast way to synthesize large non-repeating textures composed of related tiles. Recently Li and Wand [16] achieve real-time texture synthesis with generative neural networks. Texture synthesis has been applied for multi-scale synthesis (Han et al. [17] and Vanhoey et al. [18]).

Discrete element texture synthesis, the area to which our work belongs, is similar to pixel-based texture synthesis but instead of directly manipulating pixels, the texture is represented with individual discrete elements. Some methods use discrete elements to synthesize raster textures. For example, texture bombing splatters small texture elements into a larger texture (Glanville [19]) and more recently this has been optimized for on-the-fly generation on GPUs (Wang et al. [20]). Texture sprites can be efficiently applied to surfaces directly on the GPU (Lefebvre et al. [21]). Other methods directly synthesize arrangements of individual elements (such as an arrangement of multiple 3D flower models), using the elements directly and without rasterizing them to a pixel buffer. Ijiri et al. [22] synthesize 2D arrangements of elements by incrementally copying elements from an example to grow a network of interconnected elements and it is closely related to Barla et al.'s [23] method for synthesizing stroke patterns. Ma et al. [24] extend Kwatra et al.'s [13] expectation maximization framework to discrete element textures. Ma et al. [2] synthesize dynamic discrete element textures, such as swimming schools of fish. Xing et al. [25] apply discrete element texture synthesis for drawing autocompletion. Roveri et al. [1] reframe discrete element texture synthesis as a continuous problem and explore the use of gradient descent optimization on an example and output signal for specialized repetitive 3D structure synthesis. Finally, while not example-based, Loi et al. [26] propose a scripting language for technical artists to design a wide variety of element textures.

A major limitation of Roveri et al.'s [1] repetitive structure synthesis is that it requires a repetitive enough example texture (according to their article in Section 7.4, Paragraph 4 [1]). Roveri et al.'s [1] algorithm gets stuck in local optima when synthesizing textures without the necessary repetition. We do not have this limitation (see Section 6.3). In general, we achieve superior run-time performance in our test. However, on textures with enough repetitive structure present, Roveri et al. [1] achieve superior results (see Section 6.3). Concerning interactivity, we also support generative brushes (two other notable systems with generative brushes are Ijiri et al. [22] and Emilien et al. [27]). Our system is designed to build virtual worlds composed of multiple discrete textures, in contrast, Roveri et al.'s system is designed for synthesizing a 3D object from a single example texture.

Our region-growing algorithm is related to the fast and interactive region-growing algorithm in Ijiri et al.'s [22] work. However, the graph topology and heuristics of Ijiri et al. require 1-ring neighborhood comparisons, in contrast, we build on Ma et al.'s [24] point-based representation and neighborhood matching to support larger neighborhoods and thus more complicated texture examples (such as the coral in Figure 9 or the glass tilings in Figure 14). Ma et al.'s [24] method is primarily an offline texture synthesis process, in contrast, we support multiple example textures and per-element texture-synthesis parameters, an interactive sketch-based interface, and a significant increase in the rate of synthesis with our new fast region-growing algorithm. We also demonstrate that region-growing can synthesize textures that are difficult to achieve with Ma et al.'s initialization (Section 6.4).

Many works in discrete element texture synthesis demonstrate results on a single example (Ijiri et al. [22], Ma et al. [24] and Roveri et al. [1]) and it might be simple to extend them to more than one example. However, creating a powerful, flexible and interactive system with support for multiple examples is not so straightforward. Our example-palette is an interactive and persistent space where new textures can be designed. Furthermore, to support examples at different scales, we introduce per-element parameters to control the texture synthesis process (Section 4). Particularly in the case of Roveri et al. [1], it might be difficult to modify their system to support multiple textures with per-element parameters, due to more complicated derivations of their gradient functions used for gradient descent based optimization.

We chose Ma et al.'s [24] point-based representation and optimization framework over other recent works, such as Roveri et al.'s [1], for the variety of textures that this framework can support. In Section 6.3, we demonstrate the limitations of repetitive structure synthesis on some examples. In our experiments, our region growing algorithm achieves superior run-time performance to both Ma et al. [24] and Roveri et al. [1].

Statistics-based discrete element texture synthesis also generate elements based on examples, but in this case, the elements are synthesized by a statistical model derived from an example. Hurtut et al. [28] consider the bounding boxes of elements during synthesis, while Landes et al. [29] improve on this with a shape-aware model. Roveri et al. [30] synthesize point distributions with adaptive density and correlations. Recently, Emilien et al. [27] and Gain et al. [31] use sketch-based tools for synthesizing element distributions for building virtual worlds.

Emilien et al.'s [27] palette stores statistical models learned from examples specified in the scene. In contrast, our example-palette is an environment where the user can design and experiment with example arrangements and then apply those textures to the scene. A limitation of Emilien et al.'s method is synthesizing densely packed and structured textures like the glass tilings in Figure 14.

Geometry synthesis, model synthesis and procedural modeling Bhat et al. [32] introduce the idea of example-based synthesis for geometric textures on 3D surfaces (using a voxelized output domain), with user-designed orientation fields guiding synthesis, we use a similar idea for discrete element textures. Zhou et al. [33] took Bhat et al.'s [32] idea a step further and generated quilted surface geometries. The idea was refined by Yuksel et al.'s [34] multi-stage pipeline based on stitch meshes. In contrast to these techniques, we do not require or rely on the topological relationship between the elements (vertices).

3D procedural models have been synthesized based on example models (Merrell and Manocha [35] and Peytavie et al. [36]). Bokeloh et al. [37] build a shape grammar by analyzing an input model for symmetric regions. The shape gram-

mar 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. [38]). Palubicki et al. [39] use sketch-based interfaces to generate trees. Li et al. [40] guide the growth of grammars across a surface with a user defined-tensor field. In contrast to procedural and grammar based methods, discrete element texture synthesis algorithms (like our method) generate semi-repetitive arrangements of elements that have locally similar features across the output. Grammar-based systems are capable of producing branching structures that our method is not.

An example-based system for sketching structured decorative patterns was developed by Lu et al. [41]. Guerrero et al. [42] synthesize patterns with a tool that explores pattern variations. Tangles are a recursive and repetitive pen-and-ink 2D art style, Santoni and Pellacini [43] define a grammar for procedurally generating them within arbitrary polygons. These specialized algorithms have heuristics designed around stroke and curve synthesis. In contrast, our algorithm's heuristics are based on reproducing the repeating or semi-repeating element arrangements within an example discrete element texture.

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.

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 discrete element textures in the example-palette. We base our generative tools on a new fast region-growing algorithm for discrete element texture synthesis. An interleaved optimization (based on the optimization from Ma et al. [24]) step further improves previously synthesized element arrangements.

3.1 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 5.2).

We track where new elements can be synthesized with so-called free-space points (Section 5.1). We derive free-space points from an analysis step performed on the example-palette. We use free-space points as an efficient method to keep the active problem 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 interleave an optimization step (Kwatra et al. [13] and Ma et al. [24]) to reduce these artifacts. We compare region-growing alone to region-growing with optimization, in our results and analysis (Section 6.4).

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 user designs the orientation field by placing singularities (Crane et al. [44]).

4 Discrete element texture neighbourhood similarity

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 [7]). With discrete element textures, the goal is the same, except we are comparing element positions and attributes in a neighborhood (Ma et al. [24]).

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. We iteratively copy elements from the example next to previously synthesized elements in such a way that they are perceptually-close with the example texture. We use a *neighborhood distance* measure to determine which elements to copy, based on the surrounding elements. The neighborhood distance measures the difference in the relative position and attributes of elements in two neighborhoods.

Let $e \in \mathcal{E}$ be an element in the example-palette and let $o \in \mathcal{O}$ be an element in the 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 an attribute 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.

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) measuring differences between output and exemplar neighborhoods and 2) identifying exemplar elements to copy to the output domain during the region-growing step.

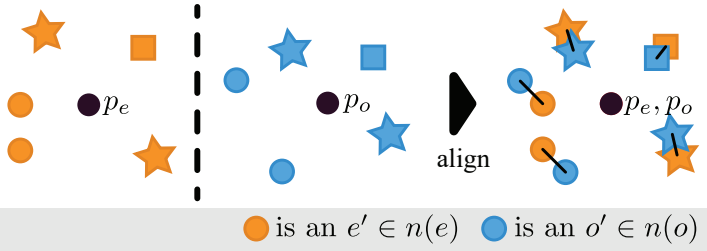


Figure 4: The similarity between two neighborhoods is found by first aligning those neighborhoods at their centroids p_e and p_o. Then, we sum the distance between pairs of elements and the attributes of those elements compared (in this case shape-type, as given by stars, circles, and squares).

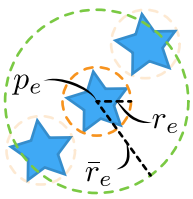


Figure 5: The neigbhorhood $n(e)$ (green circle) around p_e is determined by the neigbhorhood radius $\bar{r}_e$. The bounding sphere (orange circle) around p_e with radius r_e is used to check for overlaps with other elements—whose bounding spheres are shown in light orange.

To measure two element neighborhoods are we use Ma et al.'s [24] point-based neighborhood similarity measure. This measure aligns the centroids of two neighborhoods and compares the distance between pairs of points as illustrated in Figure 4. Let $n(e)$ (where $n(e) \subset \mathcal{E}$ and $e' \in n(e)$ is an element in the neighborhood of e) be the geometric neighborhood of e, it includes all elements within radius $\bar{r}_e$ of position p_e. $\bar{r}_e$ is the neighborhood radius of e (Figure 5), it is different than the bounding radius r_e, which is used to check to check for collisions or overlaps between elements. The distance between an output neighborhood $n(o)$ and an example neighborhood $n(e)$ 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 neighborhood $n(o)$, the pair for that element in $n(e)$ is denoted with e'.

Our generation and optimization steps rely on the notion of full and partial assignment of pairs of points between two neighborhoods $n(e)$ and $n(o)$ (Figure 6). 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 $pairs_f$.

Unlike Ma et al. [24] we use a greedy pair assignment algorithm instead of a Hungarian pair assignment (Kuhn [45]). 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. Our motivation for a greedy algorithm that excludes points is because example elements

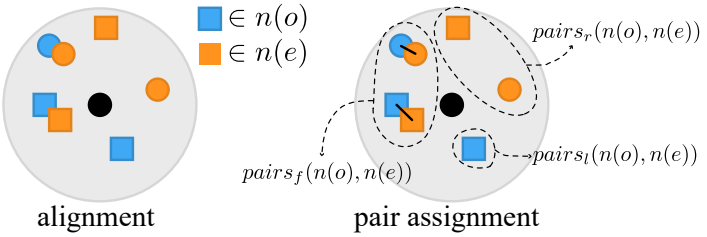


Figure 6: In finding the pairings between two neighborhoods $n(o)$ and $n(e)$ we align the two neighborhoods at their centroids. Next, we find the full pairings between elements that are close enough $pairs_f(n(o), n(e))$ and the leftover partial pairings $pairs_l(n(o), n(e)), pairs_r(n(o), n(e))$.

that do not have an output pairing are excellent candidates to copy to the output.

In detail, for two neighborhoods $n(o)$ and $n(e)$ aligned at p_o and p_e, we sort elements in the output neighborhood by distance from p_o. Then, for each sorted element $o' \in n(o)$ in the neighborhood of the output element, we find the closest element $e' \in n(e)$. If this element is within a certain threshold distance $d_{o'}$, where $|(p_{o'} - p_o) - (p_{e'} - p_e)|^2 \leq d_{o'}$, 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 neighborhood $e' \in n(e)$ form the right-partial pairs $\{(\mathbf{0}, e')\}$. Typically, the threshold distance d_e of an element e is twice the bounding radius, $d_e = 2r_e$.

In our implementation, we construct a k-d tree (Bentley [46]) 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 neighborhood size in $\mathcal{O}$ or $\mathcal{E}$.

4.1 Per-element parametrs for discrete texture synthesis

Each element has parameters—the bounding radius r_e and neighborhood radius $\bar{r}_e$ for an $e \in \mathcal{E}$—used by our element synthesis algorithm to synthesize examples at different scales and with different properties. For example, the neighborhood size of cobblestone in Figure 3 is different than the neighborhood size of a house. Likewise, the bounding radius of cobblestone is much smaller than the bounding radius of a tree. When we synthesize a new element, we copy these parameters to the output. Therefore, every element in the output receives different parameters affecting texture synthesis, this allows us to synthesize many different textures in the same scene and allows those textures to interact with each other—for example, the grass between the cobblestone in Figure 3.

5 Region-growing and optimization for discrete element texture synthesis

The selection $\mathcal{S}$ is a subset of the example-palette, $\mathcal{S} \subset \mathcal{E}$. We synthesize new elements using the selection as input to our region growing algorithm. Our goal is to synthesize elements that minimize the energy (Kwatra et al. [13] and Ma et al. [24]) 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 (Algorithm 1) 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 consider a small active 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 **seed elements** (Figure 7a). 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 7b), 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.

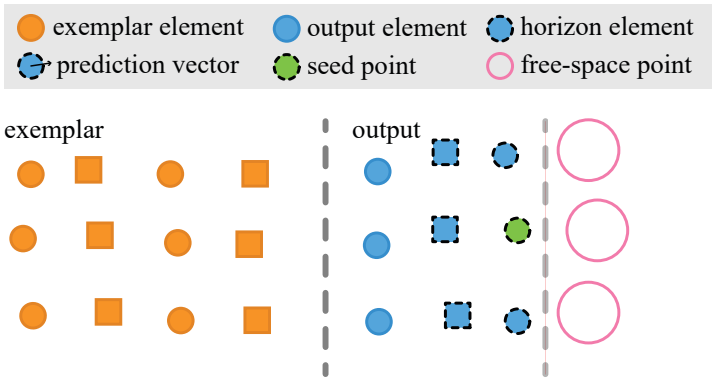
During **optimization** (Figure 7c), we visit each element in the horizon and find the most similar neighborhood in the example-palette selection. The example neighborhoods are aligned with the output neighborhoods and the elements between them paired. The difference in positions and attributes between element pairs are used by the optimization step to adjust the output to look more like the example-palette selection.

Finally, we **update the free-space** points (Figure 7d). 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. Output elements that are not nearby a free-space point are discarded from the horizon. The next iteration of our algorithm starts back at the seed-selection step.

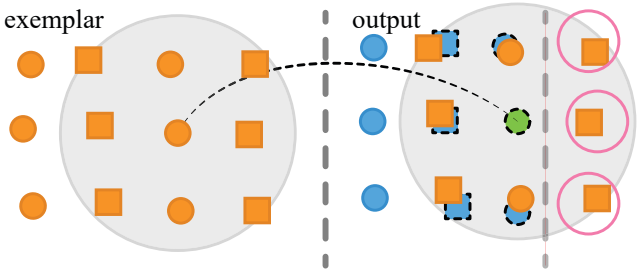
5.1 Generation and free-space

In the generation step, we find a set of seed elements $Seeds \subset \mathcal{H}$ that will generate new elements overlapping with nearby free-space points. Seeds, for example s, are selected so that some fraction of their neighborhood, with radius $0.75\ \bar{r}_s$, does not overlap with other seeds.

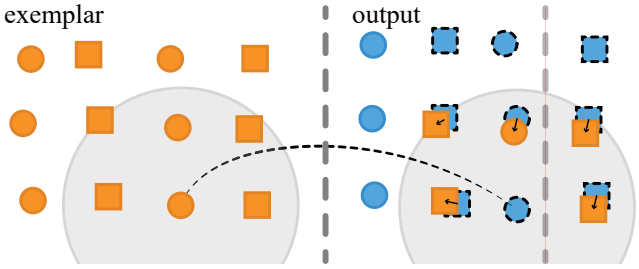
We track where to generate new elements with a set of free-space points F. A free space point $v \in F$ has a bounding



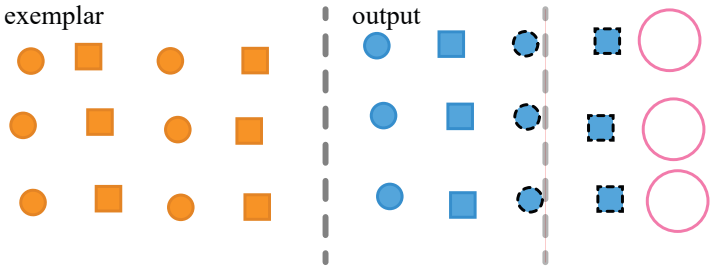
(a) **Seed Selection** A set of seed elements are selected from the horizon (in this example there is only one) such that the seeds also have free-space points nearby.



(b) **Generation** For each seed element, the most similar neighborhood 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 neighborhood in the example-palette selection is found and aligned with the horizon (here only one aligned neighborhood is shown). After pairing the elements in the horizon, the direction vectors between 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 7: 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.

Algorithm 1 $\mathcal{O} \leftarrow RegionGrowing(\mathcal{E})$

▷

$\mathcal{H} \leftarrow$ copy central patch of the selection $\mathcal{S}$
$\mathcal{O} \leftarrow \mathcal{H}$
while target region is not filled **do**
  // Generation
  select *seeds* from $\mathcal{H}$
  for $s \in seeds$ **do**
    $candidates \leftarrow \{c \in co(s) | p_c$ intersects a free-space point $v \in F\}$
    $e \leftarrow \min_e |n(s) - n(e)|$, $e \in candidates$
    $\mathcal{G} \leftarrow$ copy elements in $pairs_r(s, e)$ not intersecting $\mathcal{O}$
    $\mathcal{H} \leftarrow \mathcal{H} \cup \mathcal{G}$
    // Expand free-space
    $\mathcal{F} \leftarrow \mathcal{F} \cup \{free(g) | g \in \mathcal{G}\}$
  end for

  // Optimize the horizon
  $\mathcal{O} \leftarrow \mathcal{O} \cup \mathcal{H}$
  minimize $E(\mathcal{H}, \mathcal{S})$

  // Update free-space and prune the horizon
  remove freespace points in $\mathcal{F}$ intersecting $\mathcal{O}$
  remove all $h \in \mathcal{H}$ where p_h is too far from all $p_v, v \in \mathcal{F}$
end while

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 overlapping 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 neighborhood 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. Let e_0 be the most similar neighborhood to a seed s and e_i is another similar element neighborhood. If e_0 cannot generate elements, then we consider e_i. However, we need to ignore bad suggestions, so we compare the two neigborhoods e_0 and e_i. If $|n(o) - n(e_0)|/|n(o) - n(e_i)| > c$ for all e_i and for some ratio 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 elements that cannot predict new elements from the horizon, at the start of the generation step. We prune an element in the horizon, $h \in \mathcal{H}$, when there are no free-space points in F that overlap with a bounding sphere around p_h of radius r_h.

Rather than a 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 [7]). *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 close together in the exemplar will also tend to be close together in the output domain, which is *coherence* (Ashikhmin [10] and Tong et al. [11]). As in Ma et al. [24], we use the idea of k-coherence search to reduce the size of the search space to a user-defined k.

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 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. A typical value of k in our system is between 3 and 5, recall that every element in the output has its own set of k-coherent neighbors in the example, so this provides a lot of variation in the search space.

Before deciding what element to copy to the output, we must determine 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 neighborhoods 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 neighborhoods of each $n(c)$ to get the set of vectors $free(e) = \{p_{c'} - p_c | c' \in n(c)$ *and* $c \in co(e)\}$. To reduce the number of vectors in $free(e)$ we use density based clustering (Ester et al. [47]), replacing the points with the centroids of each resulting cluster.

After the optimization step, we update the free-space points relative to the output elements. We derive the free-space points from the relative offset of the free-space vectors from their respective elements. We rebuild 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 (Bentley [46]).

5.2 Optimization

The optimization step reduces the energy of recently synthesized elements in the horizon relative to the example-palette selection. The optimization problem consists of elements in the horizon and nearby elements within a small distance of

the horizon (by default, this distance is the size of the largest neighborhood radius in the example-palette). Expanding the horizon improves coherence between horizon elements and frozen elements in the output domain. We call this expanded horizon $\bar{\mathcal{H}}$, and naturally $\mathcal{H} \subset \bar{\mathcal{H}}$.

We minimize the energy function $E(\bar{\mathcal{H}}, \mathcal{S})$ by arranging the elements in the expanded horizon so that the neighborhood of each 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. We use Ma et al.'s [24] least-squared optimization method to find new positions for these elements from their predictions.

For each $h \in \bar{\mathcal{H}}$ 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 prediction vector $\hat{p}(h, h') = p_{e'} - p_e$ for each $h' \in n(h)$. The new predicted position of each neighboring horizon element is $p'_{h'} = p_{h'} + \hat{p}(h, h')$. We end up with many such new predicted positions for each element. We solve this overdetermined optimization problem using Ma et al.'s [24] least squares optimization method coupled with the Sparse Cholesky Decomposition module implemented in the Eigen matrix library (Guennebaud et al. [48]). In the next section we will look at the weights that are applied to each prediction vector to improve the output results.

5.3 Prediction vectors

A tug-of-war occurs when two neighborhoods provide two different prediction vectors for the position of an element, the result is an averaging of the two predictions. Ma et al. [24] solve this problem by weighting prediction vectors by their length, but this means that if a very good prediction vector has a long length, it will receive a low weight in the optimization. In contrast, we solve this problem by considering the distribution of prediction vectors in the neighborhood of $h \in \bar{\mathcal{H}}$ (Figure 8). Principal component analysis of 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 $\hat{p}(h, h')$ for h and $C = \bar{X}^T \times \bar{X} * 1/m$, the weight α_i that we give to one of the prediction vectors $\hat{p}_i$ for h is

$$\alpha_i = exp\left(\frac{\bar{p_i}^T \times C^{-1} \times \bar{p_i}}{-2}\right). \quad (4)$$

To maintain coherence with output elements not in the horizon, we give low weight to predicted positions for elements not in the horizon while also giving their previous positions a high weight. This weighting allows old elements to move but otherwise constrains the prediction vectors to work on horizon elements.

5.4 Sketch-based interaction

We support various sketch-based tools for synthesizing discrete element textures in our system. The *generative-brush*

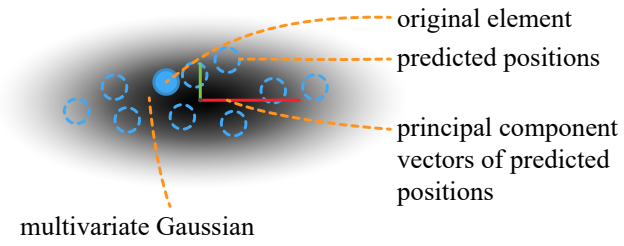


Figure 8: Prediction vectors for an element are weighted by a multivariate Gaussian distribution (the gray function in the background) for the set of predictions. The basis for this Gaussian function is found through principal component analysis on the set of prediction vectors.

(Figures 9a and 9b) synthesizes new elements in a small region along the brush path. We can also apply the generative-brush 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 9c) 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 9d) removes elements within a certain distance from a brush path. These tools work on the scene (for example, the bunny-planet) and the example-palette.

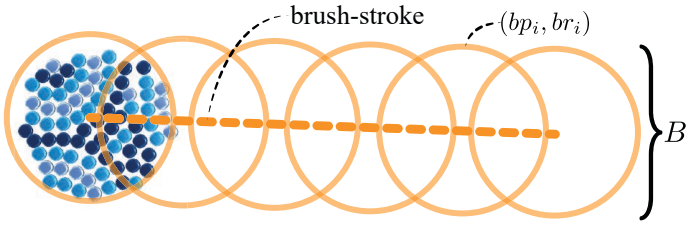
The example-palette can be built through manual element placement and attribute assignment. Copy-paste operations may also be applied. However, a more exciting possibility enabled by our system is to design the example-palette textures using our sketch-based tools and textures selected from elsewhere in the example-palette (see Figures 10a-e for one such example).

We represent generative brush-strokes across a 3D surface in our system as a series of brush-points B 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 set of brush-points B constrains the generation step to nearby brush-points (Figure 9a). 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 from B when there are no free-space points that overlap with it. The eraser tool removes overlapping brush-points from B and any nearby overlapping elements (Figure 9d).

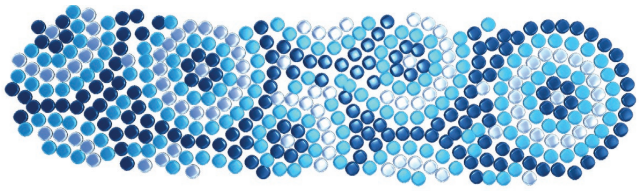
In Figure 10, we demonstrate a sketch-based 3D modeling and texture synthesis application of our system with a **stretch-tool**. The stretch tool deforms the underlying geometry and removes any affected elements. Next, we synthesize new elements to fill the affected region.

5.5 Example-palette

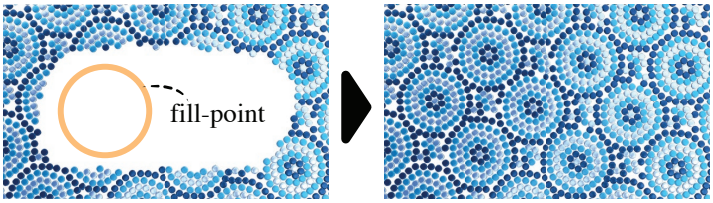
The user can manually design element arrangements in the example-palette or use our interactive tools to synthesize el-



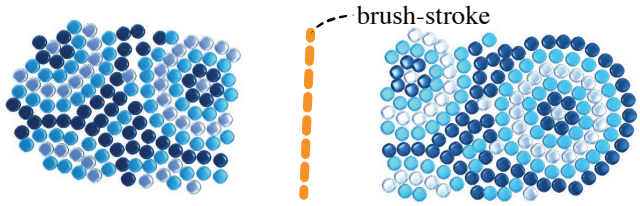
(a) **Generative-Brush** A randomly selected patch from the example-palette selection is copied to the output at the first brush point to seed region-growing along the rest of the brush points in B.



(b) The output domain after synthesizing elements along the brush-stroke in Figure 9a.



(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 9: Our sketch-based tools.

ements back into the example-palette. When designing an example arrangement, the user can assign the neighborhood radius $\bar{r}$ and bounding radius r to each element (or a group of elements) using a property editor user interface. When we modify elements in the example-palette, we recalculate the k-coherent neighbors for the rest of the example palette and update the affected k-coherence caches in the output.

5.6 Filling voids in the exemplar

Our region-growing algorithm tends to fill sparse regions greedily (Figure 11). To counter this, we let the user place invisible elements in those regions (Figure 12). 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

With surface synthesis, prediction vectors tell us in which direction to 'walk' across the mesh. We use Crane et al. [44] to find an orientation for the output domain mesh. Crane et al.'s method is closely related to Ray et al.'s [49]. The orientation field can be automatically computed (it takes less than one second), or the user can design it by placing singularity points on the mesh as described by Crane et al. [44].

For an element $h \in \bar{\mathcal{H}}$ and a prediction vector $\bar{p}$ in its neighborhood, 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. We use this integral curve walking for mapping new elements, prediction vectors, and free-space points onto the surface—this is a similar idea to the orientation fields in Bhat et al. [32]. It is also similar to Wei and Levoy [9], except we do not compute the orientation field on-the-fly as we synthesize, it is interactively designed and precomputed.

5.8 View based synthesis

In a 3D scene, many of the components of that scene may be occluded from view (Figure 13). 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). When all visible elements have been generated, we start synthesizing elements at nearby non-visible free-space points.

6 Results and discussion

6.1 Implementation and parameters

We have open-sourced our implementation and experiments at https://github.com/REDACTED FOR ANONYMOUS REVIEW under a permissive MIT license. Our implementation was developed as an editor module within the Unreal Game Engine (Epic Games, Inc. [50]). We use the Unreal Editor property editor interface to configure element parameters.

The element bounding radius r_e defines a bounding sphere around an element e and is used to check for overlaps with other elements, if it is too small, it will lead to crowded output. The neighborhood size of an element is controlled by $\bar{r}_e$, it should be large enough to capture the patterns in the example (Figure 3 contains some example neighborhood radii). A too large neighborhood radius will decrease performance. In general, we found a 3-ring or less neighborhood sufficient for our results. A 3-ring neighborhood is one where $\bar{r}_e = 3a$, where a is the average spacing between elements in an example texture. A low value for k in k-coherence search will improve the speed of synthesis, but it will reduce variation in the output. We found a value of five, for k, to be sufficient

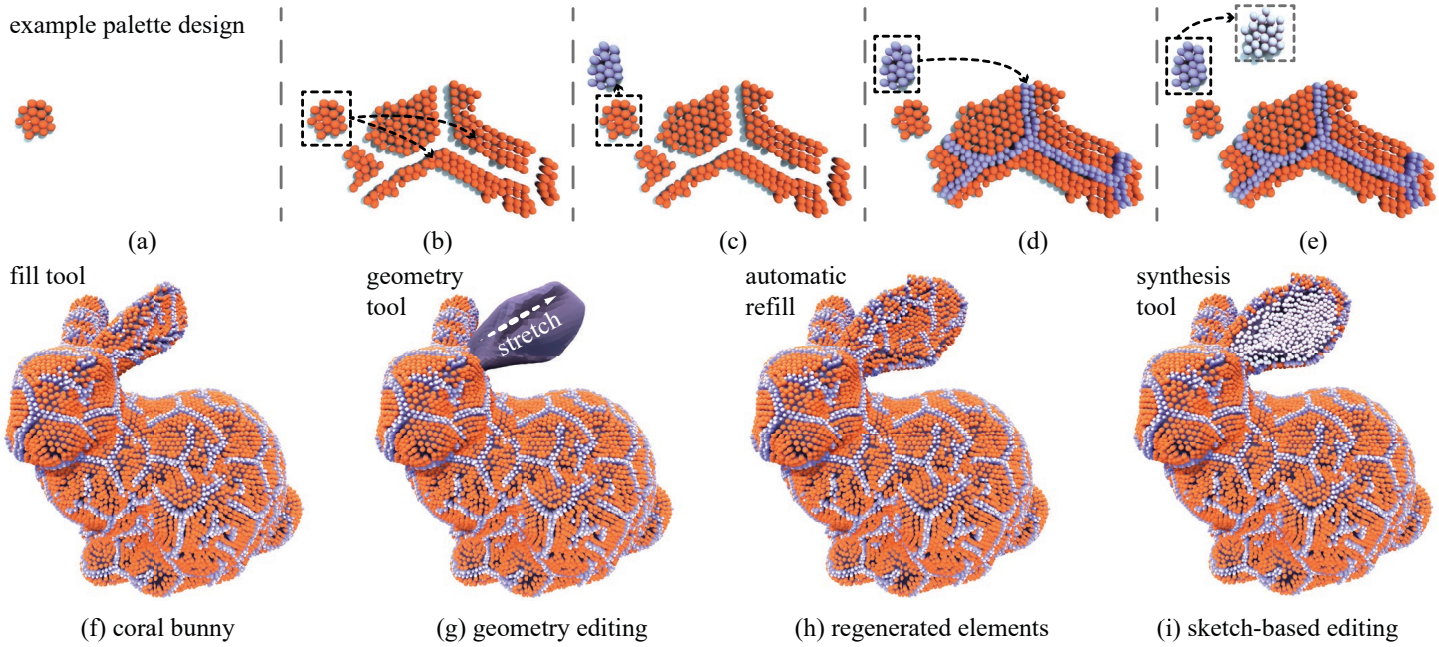


Figure 10: This figure demonstrates sketch-based 3D modeling and discrete element texture synthesis. (a) Example-palette design: a small example arrangement is created by manually placing elements. (b) The example from (a) is used to sketch a more complicated texture. (c) A new example is created. (d) The new example 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) We erase elements on the ear and replace them with white elements (e). The supplementary material contains a video showing the interactive design of the bunny (filename:sketching.mp4).

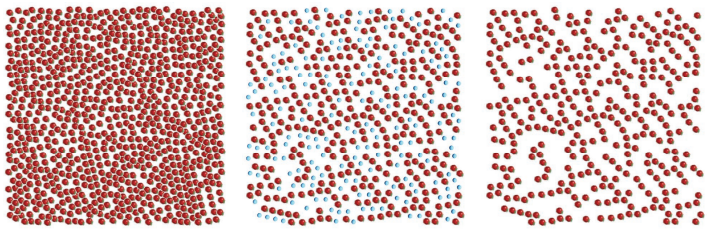


Figure 11: (a) Our region-growing algorithm greedily fills in the space between apples in Figure 12a. (b) Using the modified apple example in Figure 12b our algorithm synthesizes the correct arrangement, with the void elements shown here for illustration. (c) Void elements are hidden by default.

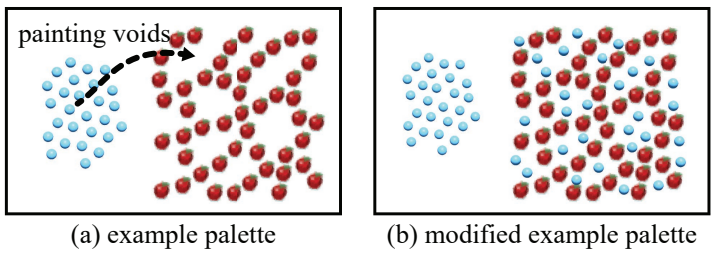


Figure 12: (a) A sparse arrangement of apples. One can paint the void example arrangement into the space between the apples to create the example in (b).

for our results. Five may seem like a low value, but recall, every element in the output has its own set of k-coherence candidates in the exemplar, in practice this leads to lots of variation. The constant c (Section 5.1) improves output quality by rejecting poor candidate neighborhoods in the generation step; it should be greater than one, the default is two (in Figure 17 we used a value of 1.4). The value d_e is used during neighborhood element pair assignment, a value twice the element bounding radius is the default. The neighborhood size $\bar{r}_e$ and r_e are the parameters we change the most.

6.2 Comparisons

Previous offline processes (Ma et al. [24] and Landes et al. [29]) can generate hundreds of elements with running times in the minutes. Our method can generate thousands of elements per second, which we demonstrate in an interactive sketch-based system. In Section 6.4 we demonstrate the limitations of patch-initialization as used by Ma et al. [24]. Landes et al.'s [29] shape-aware synthesis engine is very expensive to compute, and so we did not consider it as the basis for our interactive system.

In comparison to Roveri et al.'s [1] repetitive structure synthesis algorithm, our system supports a wider variety of textures that lack a repetitive enough pattern for their method. We also achieve superior runtime performance (see Section 6.3 for a detailed comparison).

Ijiri et al. [22] employ a fast region-growing method that

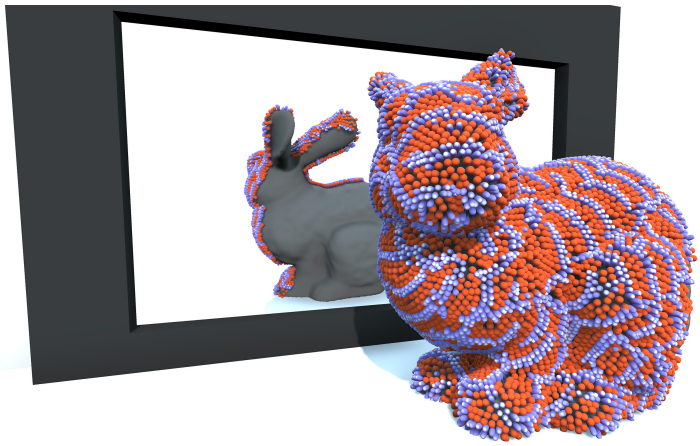
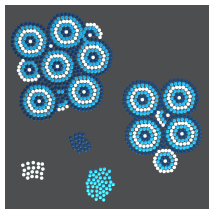


Figure 13: Elements that are not occluded are generated with priority over those that are occluded. Here one can see the backside of the bunny in the mirror does not have many elements generated yet.

(a) Real-life bottle



(b) Exemplar

(c)

(d)

Figure 14: (a) Photograph of glass tilings on a bottle by Flickr user Laras Anjung. (d) We reproduce the photo using our interactive sketch-based system and the exemplar in (b). (c) an under construction view.

relies on regularity in the topology of a network of elements (found through Delaunay triangulation). However, their approach is limited to textures expressible with 1-ring neighborhoods. In contrast, we support larger neighborhoods, we do not rely on regularity in the example texture nor do we try to maintain a network topology over the generated elements. In general, we support a wider variety of texture examples.

Emilien et al. [27] 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 multiple histograms of the pair-wise distance between points. These histograms are used for Metropolis-Hastings sampling (Geyer and Møller [51] and Hurtut et al. [28]) to synthesize new elements. A limitation of this sampling approach is high point density and synthesizing densely packed element textures. Furthermore, the histograms can have difficulty capturing relationships between more than two elements (the histograms are pair-wise), in contrast, a neighborhood similarity measure like Ma et al.'s [24] (which we also use in Section 4) can capture those relationships. For these two reasons, synthesizing the tightly packed glass tilings in Figure 14 would be difficult with Emilien et al.'s statistical synthesis method. A limitation of our method in comparison to Emilien et al. is that we cannot easily adapt the density of our synthesized arrangements relative to terrain features.

6.3 Comparisons to repetitive structure synthesis

We compare our work in detail to Roveri et al.'s [1]. While the systems were designed for different purposes, the synthesis algorithms are related. We used a single core on an Intel 5960x processor (3.0 GHz) to compare our algorithm to Roveri et al.'s [1] implementation on the examples in Figure 15. We stopped each algorithm once it had filled the output region and record the results in Table 1. We gave their algorithm additional time, up to twenty minutes, to compare their best result against the best result achieved by our algorithm (which stops when it has filled the output). We found that our algorithm can synthesize a wider variety of discrete textures that do not have a repetitive enough pattern for Roveri et al.'s [1] algorithm (Figures 15a and 15c). On a repetitive structure textures (Figure 15b), Roveri et al. [1] achieve higher quality results than our method. In each test, we have significantly better runtimes than Roveri et al. [1] (Table 1).

Roveri et al. mention that their example textures require a repetitive enough pattern to avoid bad local minima which would distort the synthesized structures (Roveri et al. [1] Section 7.4 paragraph 4). The texture in Figure 15a is a pathological input for Roveri et al.'s [1] algorithm. There are numerous local optima in which their gradient descent based neighborhood matching step (between the example and the output) can get stuck (the rings of the texture). We selected a small neighborhood from the center of the output (left result in Figure 15a) and traced the path that matching neighborhoods took (Figure 16). Many of those paths travel a short distance before getting stuck in local minima. The result is garbled output that lacks coherence. Even on a simple vertical stripe pattern (Figure 15c), there are too many local optima (Figure 16). We even tried a very high number of random starts for the initial positions in their gradient descent, without any improvement in the results, but with significantly worse runtimes. Our region-growing algorithm combined with k-coherence search (Ashikhmin [10] and Tong et al. [11]) selects output element neighborhoods that have good coherence

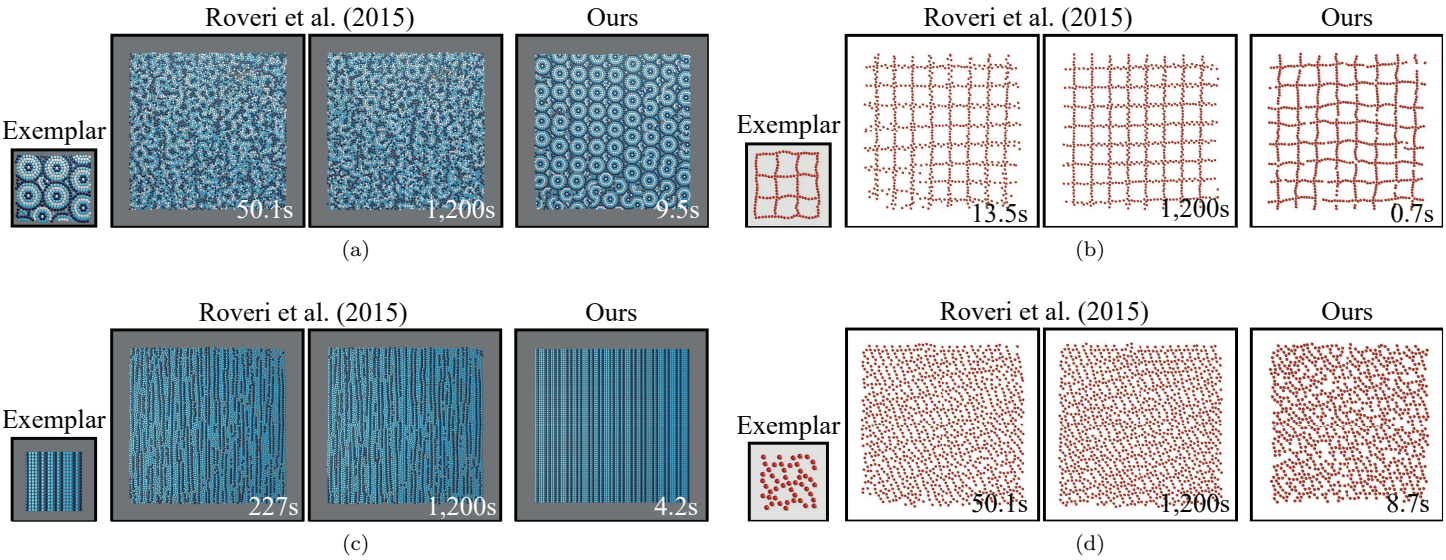


Figure 15: A comparison of Roveri et al.'s [1] repetitive structure-synthesis algorithm to our algorithm. The timings are for how long it took each algorithm to fill the given output region. When our algorithm has filled a region, it stops. After Roveri et al. [1] filled the output region, we gave it up to an additional twenty minutes (middle figures) so that we could compare their best results to ours. Our algorithm is significantly faster for each of these results.

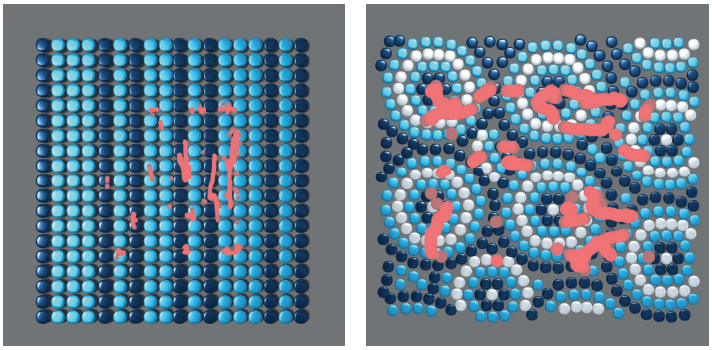


Figure 16: A trace of Roveri et al.'s [1] matching points in the exemplar as their gradient descent based optimization runs. The matching points were selected from the quadrature points in a small region of the output for Figures 15c and 15a respectively (for the 1200s results). In both cases, the matching points get stuck in local optima on either the rings or the stripes. This leads to poor coherence in Roveri et al.'s [1] output textures in Figures 15c and 15a.

on these examples.

Random sampling control for synthesizing new elements is another limitation of Roveri et al.'s [1] method in comparison to our algorithm. First, multiple false starts that result in discarded elements reduces the speed of their method. Second, randomly chosen positions can throw off the alignment of subsequently synthesized elements—manifested as branching artifacts in 15c.

Roveri et al.'s [1] method was designed for synthesizing 3D objects composed of repetitive structures. Not surprisingly, when we give their method a repetitive enough structure they achieve higher quality results than our method (Figure 15b). Their output lacks the variation of our method. However, it achieves a more accurate distribution of element densities—we have gap and alignment artifacts in our output. On Figure 15d, we achieve more variation, but once again, Roveri et al. [1] achieve a better distribution of element densities. However, on these results, we achieve significantly faster runtimes.

In general, our system is a robust and comprehensive system for synthesizing elements on 3D surfaces. We support multiple textures at multiple scales. Furthermore, we allow the interactive design of new discrete texture exemplars in an example-palette and the interactive design of our surface orientation fields. In contrast, Roveri et al.'s [1] real-time interactive system, is designed for synthesizing a 3D object from a single repetitive structure texture. They support generative brushes (seen in many other works, such as Ijiri et al. [22], Ma et al. [2], Lu et al. [41] and Emilien et al. [27]), but lack features like an example-palette and support for textures at different scales that are critical for building virtual worlds like our bunny-planet. To support large orientation fields, they either use the principal curvature direction of the underlying mesh (without user control) or allow the user to place control points to define the orientation field, for a large output this is tedious and time-consuming.

On an algorithmic level, we have demonstrated superior performance and support for a wider variety of texture examples. While in principle, it might be possible to add multiple texture support to Roveri et al.'s [1] method, supporting textures at different scales, with per-element parameters, is a challenge. It is not immediately apparent how this would affect the derivation of their gradient functions; it would also have implications for the sample density of their background grid (some adaptive technique may be required for performance). We suspect these changes would be non-trivial and would need further research.

Example	Time Roveri et al.	# output Roveri et al.	Time us	# output us
Glass tiles Fig. 15a	50.1s	6143	9.5s	6451
Grid Fig. 15b	13.5s	765	0.7s	789
Stripes Fig. 15c	200.6s	6083	4.2s	6241
Apples Fig. 15d	26.5s	1142	8.7s	1154

Table 1: The timings and number of elements to fill the output regions in Figure 15 for Roveri et al. [1] and our algorithm.

6.4 Comparisons to global optimization and patch initialization

In this section we explore our decision to use local horizon optimization and region-growing versus patch-initialization and global optimization (Figure 17). In general, we find that region-growing alone generates results superior to Ma et al.'s [24] 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.'s [24], we implemented their patch-initialization scheme in our system. Patch-initialization as described by Ma et al.'s [24] 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 empty cells in the output domain. To measure the empirical quality of an output arrangement of elements relative to an example-palette, we track average neighborhood distance:

$$\frac{\sum_{o \in \mathcal{O}} |n(o) - n(e)|}{|\mathcal{O}|}. \tag{5}$$

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). A perfect system would see this distance remain constant, or decrease, 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 chose the three exemplars in Figure 17 to include a complex semi-repeating arrangement of elements (the glass tilings in Figure 17a) to a more stochastic arrangement of elements derived from an example by Ma et al. [24] (the apples in Figure 17c). Artifacts are left present in 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. 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. Patch-initialization can make abysmal choices (random) that are impossible for the optimization steps to correct (it gets stuck in local optima). Therefore, a good initial configuration (as created by region-growing) is very important. Furthermore, the regular cell shape used by patch-initialization cannot capture the underlying patterns found in many of our exemplars.

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.

Initially, region-growing can make some perfect decisions (so average neighborhood distance is near zero). Eventually, we run out of perfect greedy and local 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. A generation algorithm that made global decisions (across the horizon) 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 17). Poor choices accumulate, eventually the coherence between neighboring elements becomes low, and so the choices found by neighborhood matching 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 Figures 17a and 17b, 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. During optimization, we combat this by expanding the horizon to include elements within a constant distance of the horizon elements.

The last exemplar (Figure 17c) 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 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 the optimization step to recover. Furthermore, it is not possible to align patch-initialization cells to some of our examples, such as in Figure 17. An interleaved region-growing and optimization strategy consistently produces arrangements that have good average neighborhood distance. This analysis demonstrates a significant improvement in qual-

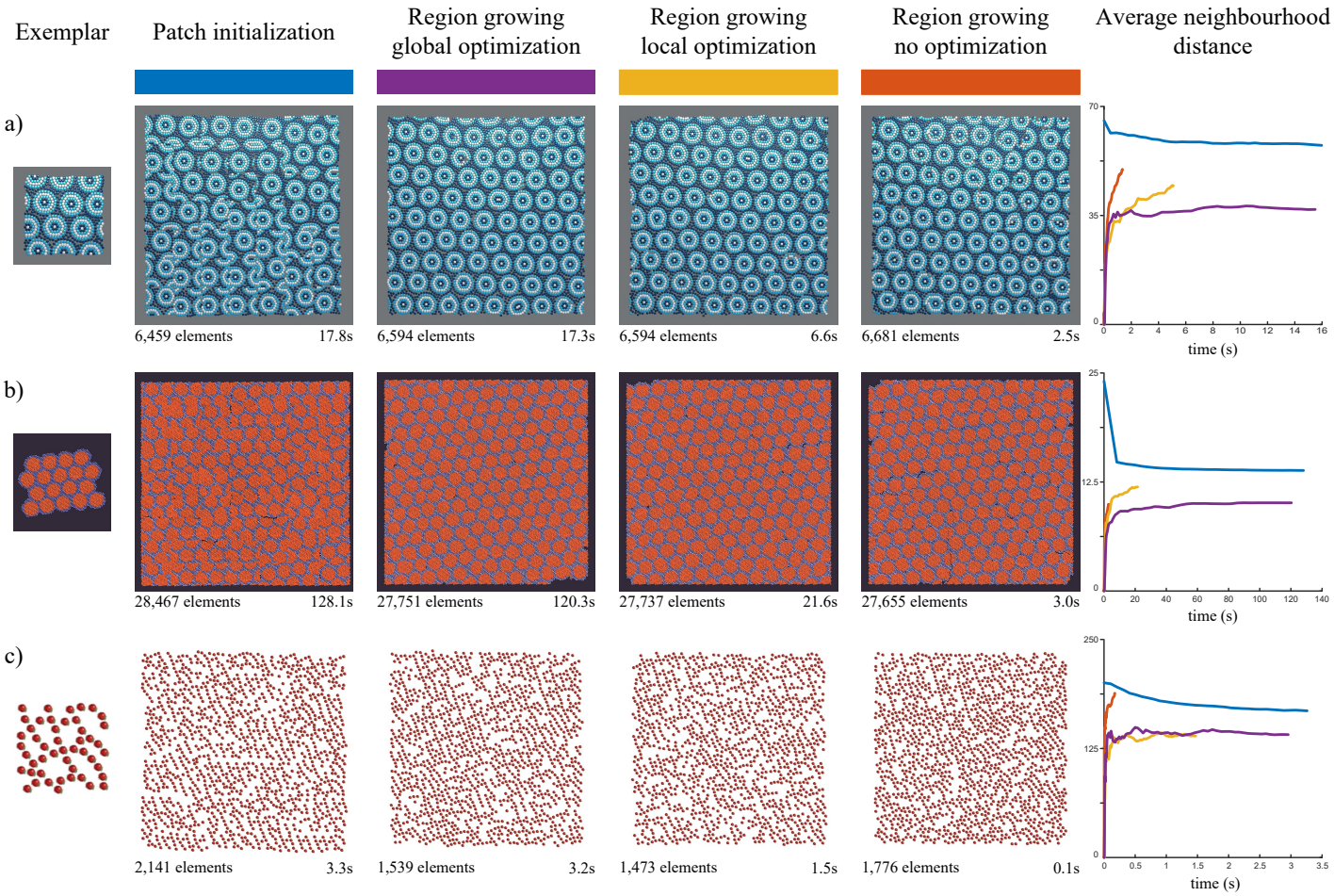


Figure 17: A comparison of Ma et al.'s [24] patch-initialization as implemented in our system versus global, local and no optimization with region-growing. The exemplar for each row is given on the left, and a plot of the average neighborhood similarity (Equation 6.4) for each row is given in the right column. In the bottom of each screenshot, we provide 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.

ity over Ma et al. [24] with our interleaved generation and optimization method.

6.5 Results

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, elements in the palette have different neighborhood radii. For example, trees have a large neighborhood while the rocks have a small neighborhood (see Figure 3).

We demonstrate the application of our system to 3D modeling with a Stanford coral-bunny example (Figure 10). When we deform the mesh, such as by stretching the bunny ears, we discard elements in the stretched areas and then automatically fill in the 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.

We used our system to design a virtual object inspired by a photo of a glass-tiled bottle (Figure 14). Ma et al. [24] demonstrated their method to change the attributes of the glass tiling, however with evidence from Section 6.4 we suspect that patch-initialization cannot synthesize the initial state of the bottle. Our bottle contains 21,806 elements synthesized at an average rate of 1,100 elements/s. It took about 30 minutes to design. We capture the circle patterns using 3-ring neighborhoods. There are 1,061 elements in the example-palette.

7 Conclusion and future work

Our example-palette enables the interactive sketch-based design and synthesis of virtual worlds composed of a variety of element arrangements. Our fast region-growing algorithm allows interactive rates of synthesis suitable for sketch-based modeling. Another aspect to our interactive rates of synthesis is a technique for efficient pruning of the region-growing horizon. We demonstrate results that are an improvement

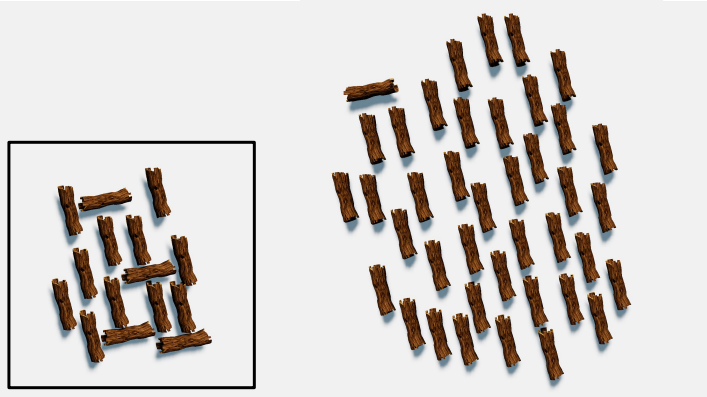


Figure 18: (inset) An exemplar containing oriented logs. The logs are composed of multiple samples to capture their long shape (as in Ma et al. [24]). Our region growing algorithm greedily chooses the horizontal logs over the vertical logs.

over previous work, and our system as a whole achieves new results that would be difficult with those methods (Ma et al. [24] Ijiri et al. [22], and Roveri et al. [1]). While Ma et al. [24] 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 (Emilien et al. [27]).

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.

There are limitations to our system. For example, significant differences in element size within a single exemplar are not supported by our neighborhood matching scheme. A multi-scale approach would be required to capture multi-scale features in a single example discrete texture, and this is one possible direction for future work.

Our method supports multi-sample elements, which can be used to capture more complicated shapes (Ma et al. [24]). However, we greedily copy all of an element's samples at once, leaving no room for other better choices. In Figure 18 the vertically oriented logs dominate as they tend to get copied first and then there is no room to copy horizontal logs. Fixing this would require a less greedy selection process, perhaps one with backtracking.

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 14, but beyond this, our system would lose interactivity.

Our pair-wise neighborhood similarity measure does not capture perceptual similarity intuitively. In future work, we think there is promise in using something like Roveri et al.'s [1] continuous function representation to compare neighborhoods of elements.

Finally, we applied a local and greedy strategy for generating elements. A global strategy that considers the entire horizon could avoid some of the artifacts that we observed in our results.

References

[1] 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.

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

[3] Schachter, B, Ahuja, N. Random pattern generation processes. Computer Graphics and Image Processing 1979;10(2):95–114.

[4] Perlin, K. An image synthesizer. ACM Siggraph Computer Graphics 1985;19(3):287–296.

[5] 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.

[6] Tamura, H, Mori, S, Yamawaki, T. Textural features corresponding to visual perception. IEEE Transactions on Systems, man, and cybernetics 1978;8(6):460–473.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] Dischler, JM, Maritaud, K, Lévy, B, Ghazanfarpour, D. Texture particles. In: Computer Graphics Forum; vol. 21. Wiley Online Library; 2002, p. 401–410.

[13] 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.

[14] 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.

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

[16] 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.

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

[18] 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.

[19] Glanville, S. Texture bombing. GPU Gems: Programming Techniques, Tips, and Tricks for 2004;.

[20] Wang, L, Shi, Y, Chen, Y, Popescu, V. Just-in-time texture synthesis. In: Computer Graphics Forum; vol. 32. Wiley Online Library; 2013, p. 126–138.

[21] Lefebvre, S, Hornus, S, Neyret, F. Texture sprites: Texture elements splatted on surfaces. In: Proceedings of the 2005 symposium on Interactive 3D graphics and games. ACM; 2005, p. 163–170.

[22] 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.

[23] 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.

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

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

[26] Loi, H, Hurtut, T, Vergne, R, Thollot, J. Programmable 2d arrangements for element texture design. ACM Transactions on Graphics (TOG) 2017;36(3):27.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

[36] 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.

[37] 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.

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

[39] 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.

[40] 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.

[41] 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.

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

[43] Santoni, C, Pellacini, F. gtangle: a grammar for the procedural generation of tangle patterns. ACM Transactions on Graphics (TOG) 2016;35(6):182.

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

[45] Kuhn, HW. The hungarian method for the assignment problem. Naval research logistics quarterly 1955;2(1-2):83–97.

[46] Bentley, JL. Multidimensional binary search trees used for associative searching. Commun ACM 1975;18(9):509–517. URL: http://doi.acm.org/10.1145/361002.361007. doi:10.1145/361002.361007.

[47] Ester, M, Kriegel, HP, Sander, J, Xu, X. A density-based algorithm for discovering clusters in large spatial databases with noise. In: Kdd; vol. 96. 1996, p. 226–231.

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

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

[50] Epic Games, Inc., . Unreal engine 4 game engine. 2018.

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