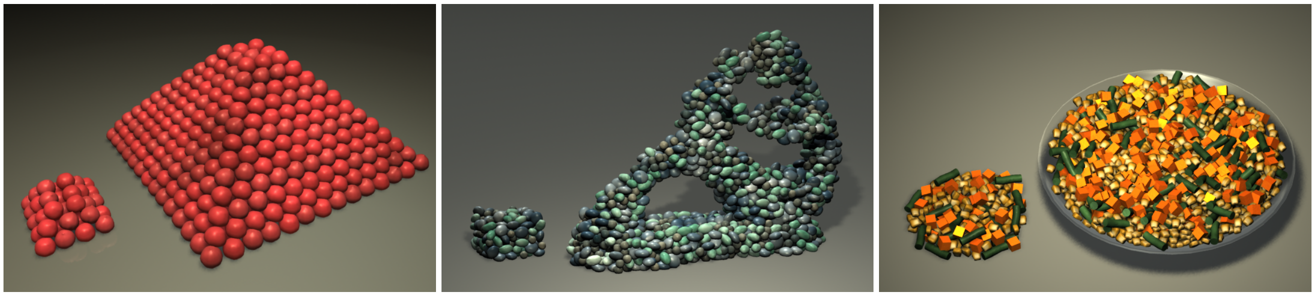
Discrete Element Texture Synthesis



(a) plum stack (b) pebble sculpture (c) dish with corns, carrots, and beans

Figure 1: Discrete element texture synthesis. Given a small input exemplar (left within each image), our method can synthesize the corresponding output with user specified size and shape (right within each image). Our method, being data driven, can produce a variety of effects, including realistic/artistic phenomena, regular/semi-regular/irregular distribution, different number of element types, and variations in element properties including size, shape, and orientation. Case (a) is a semi-regular distribution with the input manually placed as the effect is difficult to achieve via physical simulation. The output in case (b) has very different shape and boundary conditions from the input. Case (c) contains different types of elements and different input and output shapes/boundary-conditions: the input is prepared on a planar board with the output served in a curved plate.

Abstract

A variety of natural and man-made phenomena can be characterized by repetitive discrete elements. Examples include a stack of fruits, a plate of dish, or a stone sculpture. Although certain results can be produced via existing methods based on procedural or physical simulation, these are often designed for specific applications. Some of these methods can also be hard to control.

We present discrete element texture synthesis, a data-driven method for placing repetitive discrete elements within a given large-scale structure. Our main goal is to provide a general approach that works for a variety of phenomena instead of just specific scenarios. We also want it easy to use, as the user only needs to specify an input exemplar for the detailed elements and the overall output structure, and our approach will produce the desired combination. Our method is inspired by texture synthesis, a methodology tailored for generating repetitions. However, existing texture synthesis methods cannot adequately handle *discrete* elements, often producing unnaturally broken or merged elements. Our method not only preserves the individual element properties such as color, shape, size, and orientation, but also their aggregate distributions. It also allows certain application specific controls, such as boundary constraints for physically realistic appearance. Our key idea is a new neighborhood metric for element distribution as well as an energy formulation for synthesis quality and control. As an added benefit, our method can also be applied for editing element distributions.

Keywords: discrete element, texture synthesis, editing

1 Introduction

A variety of natural or man-made phenomena can be characterized by a distinctive large scale structure with repetitive small scale elements. Some common examples include a stash of fruits or vegetables, a tiled house, or a stone sculpture. Due to the potential scale and complexity of such phenomena, it would be desirable to let the user specify only the overall structure while having automatic algorithms to produce the detailed elements.

One common method is physical simulation, for which the user specifies certain input controls (e.g. initial or boundary condition) and simply let the algorithm run course to produce the results [Baraff and Witkin 1997]. The primary advantage of physical simulation is fidelity to realism. However, such methods can be hard to control, as to produce the desired output the user might need to repeatedly tweak the input parameters. Physical simulation might not be suitable for man-made or artistic effects (e.g. see [Cho et al. 2007]). Another possibility is the procedural approach [Ebert et al. 2002]. However, procedural methods are known for their limited generality and only applicable to specific distribution (e.g. Poisson disk [Lagae and Dutré 2005]) or phenomenon (e.g. rocks [Peytavie et al. 2009]). Furthermore, even though many procedural methods offer control via input parameters, tuning these to achieve the desired effects might require significant expertise.

Our main goal is to provide a general approach for placing repetitive discrete elements within a given large-scale structure. By general, we mean the approach should work for a variety of phenomena instead of tied to specific applications. We also want it easy to use, i.e. the user only needs to specify a small exemplary swath of detailed elements and the overall output shape, and the approach will produce the desired combination. In addition, we would like to produce both realistic and artistic effects simply via user specified element swath and output shape, as existing methods might not offer the kind of flexibility and control that a user desires.

We present discrete element texture synthesis, a data-driven approach for placing repetitive discrete elements within a given large-scale structure. We draw inspirations from texture synthesis [Wei et al. 2009], a methodology for producing natural repetitions from given exemplars. But unlike prior methods that might produce broken or merged elements (Figure 2), our method not only preserves the individual elements but also their aggregate distributions. Our key idea is a texture neighborhood definition for elements as well as a metric measuring neighborhood similarity. By ensuring that the input and output have similar texture neighborhoods, we are able to synthesize outputs that not only preserve the individual elements but also resemble aggregate distributions in the input exemplars. Thus, the user can easily achieve the desired result by simply supplying a proper input exemplar as well as the desired overall output size and shape. Since the user has maximum flexibility in specifying both the input exemplar and the output shape, our method is able to achieve a variety of effects, including different dimensions (e.g. 2D or 3D), different element properties (including shapes, sizes, colors) and distributions (e.g. regular/semi-regular/irregular), different number of element types (e.g. a stack of plums or a plate of mixed vegetables), as well as physically realistic or artistic phenomena (e.g. a pile of pebbles or a decorative mosaic pattern).

The main technical challenge of our approach is synthesizing element distributions. Unlike many prior texture synthesis methods where the *domain* information $\mathbf{p}$ is given (e.g. positions of pixels, vertices, or voxels) and only the *range* information $\mathbf{q}$ needs to be determined (e.g. colors of pixels, vertices, or voxels), we have to compute both $\mathbf{p}$ and $\mathbf{q}$ as part of the synthesis process. We achieve this by a carefully designed metric for measuring the similarity between two texture neighborhoods, accounting for both $\mathbf{p}$ and $\mathbf{q}$. Even though there are prior methods targeting specific scenarios of this general problem (e.g. 2D NPAR distribution [Ijiri et al. 2008; Hurtut et al. 2009]), to our knowledge these are not for synthesizing general discrete elements, e.g. 3D or physically realistic effects for which our method can easily handle. We formulate our synthesis algorithm as an energy optimization process [Kwatra et al. 2005] to allow us not only properly determine $\mathbf{p}$ and $\mathbf{q}$ but also satisfy certain specific application demands, such as boundary constraints for physically plausible effects. We choose this optimization framework mainly for its flexibility, as both the basic neighborhood similarity as well as additional application-specific needs can be incorporated as individual energy terms. Adopting a familiar framework of optimization also facilitates easy extension and adoption. However, even though optimization is a common methodology and has been used in many different algorithms, the main challenges are on how to properly design the individual algorithmic components for discrete element synthesis.

As an added benefit, our method can also be applied for editing element distributions. Specifically, the user can change not only $\mathbf{q}$ but also $\mathbf{p}$ of a few elements, and our method will automatically propagate such changes to all other elements with similar texture neighborhoods, relieving the user from the potential tedious chore of manual repetitions. This editing application is possible thanks to the texture neighborhood metric we developed for direct synthesis.

2 Previous Work

Multi-scale synthesis A variety of phenomena consists of small scale repetitions within a distinctive large scale structure. Such phenomena could be computed with better quality or efficiency by applying different methods for different scales; some examples include fluid turbulence [Kim et al. 2008; Narain et al. 2008], hair strands [Wang et al. 2009], crowds [Lerner et al. 2007; Narain et al. 2009], or motion fields [Ma et al. 2009]. Our approach follows this general philosophy and focuses on discrete elements.

Example-based texturing Example-based texturing is a general data-driven methodology for synthesizing repetitive phenomena (see survey in [Wei et al. 2009]). However, the basic representations in most existing texture synthesis methods such as pixels [Efros and Leung 1999], vertices [Turk 2001] or voxels [Kopf et al. 2007] cannot adequately represent individual or *discrete* elements with semantic meanings, such as common objects seen in our daily lives. Without a basic representation that has knowledge of the discrete elements it would be very difficult to synthesize these elements adequately; even though artifacts could be reduced via additional constraints on top of existing methods (e.g. [Zhang et al. 2003; Wu and Yu 2004]), there is no guarantee that the individual elements would be preserved. Thus, the synthesized textures can have elements that are broken or merged with each other (Figure 2). Such artifacts can be quite visible and thus better avoided.

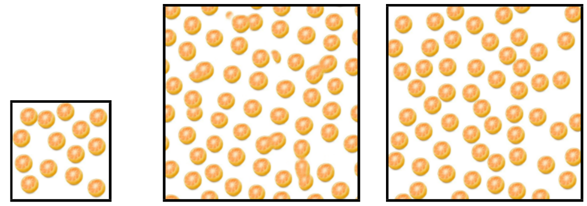


Figure 2: Pixel versus element synthesis. The pixel synthesis result is produced by combining discrete optimization [Han et al. 2006] with texton mask [Zhang et al. 2003].

Geometry synthesis Our method is also related to geometry synthesis, especially those via texture methods such as meshes [Zhou et al. 2006], models [Merrell and Manocha 2008], or terrains [Zhou et al. 2007]. However, similar to other texture synthesis methods these are mainly for continuous patterns and might lack necessary information to preserve or control discrete elements, e.g. broken elements as can be seen in Figure 5b of [Zhou et al. 2006].

Element packing There exist methods that pack a set of discrete elements into a specific domain or shape, such as mosaic tiles [Hausner 2001; Kim and Pellacini 2002] or 3D object collage [Gal et al. 2007]. However, the element distributions in these methods are usually determined via specific procedures or semi-manual user interface, instead of from input exemplars targeted at general distributions as in our approach.

Texture element placement Even though the majority of example-based texturing methods are not suitable for discrete elements, potential solutions have been explored by a few pioneering works. However, despite the promises shown in these techniques, they might fall short in certain aspects. Dischler et al. [2002] and Liu et al. [2009] obtain distribution from input exemplars to place 2D textons, but these techniques are not designed for general discrete elements. Barla et al. [2006] synthesized discrete elements but their positions are determined by Lloyd relaxation, not from the input exemplars. Ijiri et al. [2008] synthesized element positions via a growth method similar to [Efros and Leung 1999] but their method appears to be less general and more complex than ours, e.g. dealing with only 1-ring neighborhoods and requiring triangulation. Thus their method is sufficient for the target 2D NPAR applications but probably not for more general effects such 3D or physically realistic distribution. Jodoin et al. [2002] and Kim et al. [2009] applied texture synthesis for generating stipple distributions, but not general discrete elements. Hurtut et al. [2009] took into account element attributes via area and appearance analysis, but only deals with static 2D non-photorealistic elements, not 3D or physically-realistic phenomena. Our method is inspired by these pioneering works, but aims at synthesizing discrete elements in a general setting, including 2D and 3D distribution, volume and surface synthesis, regular/semi-regular/irregular configuration, variations in number of element types, shapes, sizes, as well as artistic and physically-realistic effects.

3 Core Algorithm

Given an input exemplar $\mathbf{z}$ consisting of a set of elements with the relevant domain/position $\mathbf{p}$ and range/attribute $\mathbf{q}$ information, our goal is to synthesize an output $\mathbf{x}$ that it is similar to $\mathbf{z}$ in terms of both $\mathbf{p}$ and $\mathbf{q}$. We can formulate this synthesis of discrete elements as an optimization problem [Kwatra et al. 2005] by minimizing the following energy function:

$$E_t(\mathbf{x};\mathbf{z}) = \sum_{s\in X^\dagger} |\mathbf{x}_s - \mathbf{z}_s|^2 \tag{1}$$

where E_t measures the similarity between the input exemplar $\mathbf{z}$ and the output $\mathbf{x}$ via local neighborhoods around elements. Specifically, for each output element s, we take a small set of elements near it as the texture neighborhood $\mathbf{x}_s$, find the most similar input neighborhood $\mathbf{z}_s$, and measure their distance $|\mathbf{x}_s - \mathbf{z}_s|$. We repeat this same process for each $s \in X^\dagger$, a subset of all input elements, and sum their squared differences. Our goal is to find an output $\mathbf{x}$ with low energy value. Below we describe details about our energy formulation as well as a solver for this optimization problem. For each reference, we have summarized the algorithm in Pseudocode 1.

function $\mathbf{x} \leftarrow$ DiscreteElementTextureOptimization($\mathbf{z}$)
  // $\mathbf{x}$*: output distribution*
  // $\mathbf{z}$*: input exemplar*
  $\mathbf{x} \leftarrow$ Initialize($\mathbf{z}$) // *Section 4.2*
  iterate until convergence **or** enough # of iterations reached
    // *search phase, i.e. the "M-step" in [Kwatra et al. 2005]*
    $\left\{\mathbf{z}_s, s \in X^\dagger\right\} \leftarrow$ Search($\mathbf{x}$, $\mathbf{z}$)
    // *assignment phase, i.e. the "E-step" in [Kwatra et al. 2005]*
    Assign($\left\{\mathbf{z}_s, s \in X^\dagger\right\}$, $\mathbf{x}$)
  end
  return $\mathbf{x}$

function $\left\{\mathbf{z}_s, s \in X^\dagger\right\} \leftarrow$ Search($\mathbf{x}$, $\mathbf{z}$) // *Section 3.4*
  foreach element $s \in X^\dagger$ // $X^\dagger$*: a subset of all output elements*
    $\mathbf{x}_s \leftarrow$ output neighborhood around s
    $\mathbf{z}_s \leftarrow$ find most similar neighborhood in $\mathbf{z}$ to $\mathbf{x}_s$
  end
  return $\{\mathbf{z}_s\}$

function Assign($\{\mathbf{z}_s\}$, $\mathbf{x}$) // *Section 3.5*
  foreach output element $s \in X$
    $\mathbf{p}(s) \leftarrow$ weighted combination of predicted positions
      from output neighborhoods that overlap s
    $\mathbf{q}(s) \leftarrow$ select the vote that minimizes the energy function
  end

Pseudocode 1: Discrete element texture synthesis.

3.1 User Inputs

In addition to the input exemplar, the user also needs to supply the following main inputs:

Neighborhood size This is the standard parameter for neighborhood-based texture synthesis [Wei et al. 2009]. The user simply specifies the spatial extent of the neighborhoods, and for each element s, we construct its neighborhood $\mathbf{n}(s)$ by taking the union of all elements within the spatial extent centered at s.

Output shape The user also needs to define the output size and shape. Our algorithm will then attempt to obey it as much as possible, i.e. filling the domain interior with elements while avoiding them spill outside the domain. The algorithm will also try to maintain similarity between input and output boundary element configurations.

Element attributes The user can also specify what kinds of element properties to consider. The element positions $\mathbf{p}$ are mandatory, but the range attributes $\mathbf{q}$ such as element type, geometry, and appearance could be optional depending on the target applications. See Section 3.3 for more details.

3.2 Neighborhood Metric

The neighborhood similarity metric is the core component for neighborhood-based texture synthesis algorithms [Wei et al. 2009]. For traditional texture synthesis that has fixed sample positions $\mathbf{p}$, this can be done easily by either simple sum-of-squared differences (SSD) of the range information $\mathbf{q}$ (such as colors) in a regular setting (e.g. pixels or voxels) or by resampling irregular samples into a regular setting before proceeding with SSD as in the former case (e.g. mesh vertices). However, in our case, since we have to synthesize both $\mathbf{p}$ and $\mathbf{q}$, we need to incorporate both of them into the neighborhood metric. Formally, let $\mathbf{n}(s)$ denote the spatial neighborhood around an element s. We measure the distance $|\mathbf{n}(s_o) - \mathbf{n}(s_i)|^2$ between the neighborhoods of two elements s_o and s_i via the following formula:

$$|\mathbf{n}(s_o) - \mathbf{n}(s_i)|^2 = \sum_{s'_o \in \mathbf{n}(s_o)} \left|\hat{\mathbf{p}}(s'_o) - \hat{\mathbf{p}}(s'_i)\right|^2 + \alpha \left|\mathbf{q}(s'_o) - \mathbf{q}(s'_i)\right|^2 \tag{2}$$

where s'_o is an element $\in \mathbf{n}(s_o)$, $s'_i \in \mathbf{n}(s_i)$ the "matching" element of s'_o (explained below), $\hat{\mathbf{p}}(s') = \mathbf{p}(s') - \mathbf{p}(s)$ (i.e. the relative position of s' with respect to the neighborhood center s), and α the relative weight between domain $\mathbf{p}$ and range $\mathbf{q}$ information.

Intuitively, what Equation 2 tries to achieve is (1) align the two neighborhoods $\mathbf{n}(s_o)$ and $\mathbf{n}(s_i)$, (2) match up their elements in pairs $\{(s'_i, s'_o)\}$, and (3) compute the sum of squared differences of both $\mathbf{p}$ and $\mathbf{q}$ among all the pairs. We determine the pairings by first identifying the pair (s'_i, s'_o) with minimum $|\hat{\mathbf{p}}(s'_o) - \hat{\mathbf{p}}(s'_i)|$, exclude them for further consideration, and repeat the process to find the next pair until $\mathbf{n}(s_o)$ runs out of elements. (We prevent $\mathbf{n}(s_i)$ from running out of elements before $\mathbf{n}(s_o)$ by not presetting its spatial extent, essentially giving $\mathbf{n}(s_i)$ an ∞ size.) We have found that the heuristic above works well in practice, and provides similar quality with a more rigorous but much slower approach that considering all possible pair matching (s'_i, s'_o) in brute force.

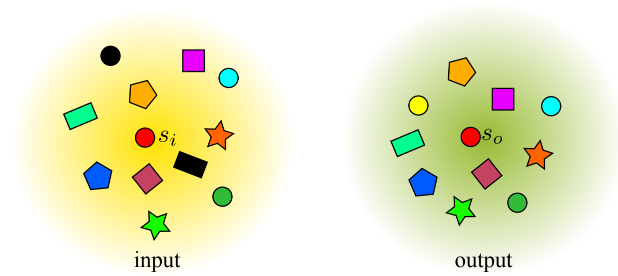


Figure 3: Illustration for our neighborhood metric. Each pair of matched input and output elements has not only similar relative positions (to the center element) but also similar color, shape, and orientation. Unmatched input elements are shown in black.

We use the neighborhood metric above throughout our algorithm, including both the search and the assignment steps. Specifically, in the search step, we use only the scalar distance value computed by Equation 2 to pick the most similar input neighborhood $\mathbf{z}_s$ for each output neighborhood $\mathbf{x}_s$. However, in the assignment step, we need further information for the matching pairs $\{(s'_i, s'_o)\}$ in order to determine the $\mathbf{p}$ and $\mathbf{q}$ values for the output element s. More details can be found in Section 3.4 & 3.5.

Note that even though some prior methods have adopted similar neighborhood metric for discrete elements, they do not entirely satisfy our needs. For example, [Barla et al. 2006] uses Hausdorff distance and thus does not allow explicit control of pair-wise element matching, e.g. the need to avoid duplications, and [Ijiri et al. 2008] considers only 1-ring neighbor positions $\mathbf{p}$ instead of our formulation that allows not only general neighborhoods but also considers both $\mathbf{p}$ and $\mathbf{q}$. Furthermore, our neighborhood definition does not require additional processing such as triangulation in [Ijiri et al. 2008], making our method easier to implement, especially for non-2D applications such as 3D volume or surface synthesis.

3.3 Element attributes $\mathbf{p}$ and $\mathbf{q}$

Here, we describe more details about the element domain $\mathbf{p}$ and range $\mathbf{q}$ information, and how to measure their differences in Equation 2. The $\mathbf{p}$ part is relatively straightforward; it is just the element position, and we measure the difference $\mathbf{p}(s) - \mathbf{p}(s')$ between two elements s and s' via the usual Euclidean metric. The $\mathbf{q}$ part can contain a variety of information depending on the particular application scenario. For the simplest case of point distribution, $\mathbf{q}$ can be empty. Below is a list for more typical applications involving concrete objects as elements:

Orientation The orientation of an element is represented as a normalized quaternion for both 2D and 3D cases. We compute the difference between two quaternions via the standard approach of taking the inverse cosine of their dot product.

Geometry Each element can have geometry with different size and shape from one another. In general, we can measure the difference between two element geometries via Hausdorff distance (after aligning element centers and orientations to avoid double counting their contributions).

Appearance Each element can also have different appearance attributes, including colors and textures. We can measure their appearance differences via color histograms.

Type In general, both the geometry and appearance are parts of the intrinsic element attributes (that remain largely invariant with respect to position and orientation). Beyond geometry and appearance, we can also consider other kinds of intrinsic element attributes depending on the specific application contexts, such as high level semantic meanings. For maximum flexibility, we allow the user to specify the distance metric between intrinsic element properties. In addition, when the number of input elements is sufficiently small or can be grouped into a small number of types, we can pre-compute their intrinsic distances for run time efficiency. For most of our examples, we have found it sufficient to use an integer number to identify the element type, and set the intrinsic distance to be 0 if they are the same, and or 1 if not.

3.4 Search Step

During the search step, we find, for each output element s_o, the best match input element s_i with the most similar neighborhood, i.e. minimizing the energy value in Equation 2. This search can be conducted by exhaustively examining every input element, but this can be computationally expensive. Instead, we adopt k-coherence search for constant time computation, as detailed in Section 4.3.

3.5 Assignment Step

$\mathbf{p}$ assignment At the beginning of the assignment step, we have multiple input neighborhoods $\left\{\mathbf{z}_{s'_o}\right\}$ overlapping every output element s_o, where $\mathbf{z}_{s'_o}$ is the matching input neighborhood for output element s'_o as determined in the search step (Section 3.4) and s'_o is sufficiently close to s_o so that the spatial extent of $\mathbf{z}_{s'_o}$ covers s_o. Each such $\mathbf{z}_{s'_o}$ provides a predicted position $\tilde{\mathbf{p}}(s'_o, s_o)$ for element s_o:

$$\tilde{\mathbf{p}}(s'_o, s_o) = \mathbf{p}(s'_o) + \mathbf{p}(s_i) - \mathbf{p}(s'_i) \tag{3}$$

where s_i/s'_i indicates the matching input element for s_o/s'_o as described in the neighborhood metric (Section 3.2). See Figure 4.

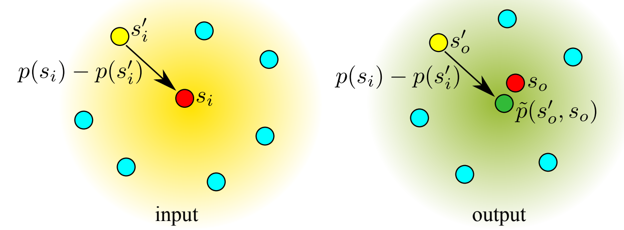


Figure 4: Illustration for the assignment step.

To minimize the energy function E_t in Equation 1, the sample position $\mathbf{p}(s_o)$ is updated as a weighted combination of all $\{\tilde{\mathbf{p}}(s'_o, s_o)\}$ where $\mathbf{z}_{s'_o}$ covers s_o:

$$\mathbf{p}(s_o) = \frac{\sum_{s'_o} \omega(s'_o, s_o) \cdot \tilde{\mathbf{p}}(s'_o, s_o)}{\sum_{s'_o} \omega(s'_o, s_o)} \tag{4}$$

The relative weight ω is determined as

$$\omega(s'_o, s_o) = \frac{1}{\alpha |s'_o - s_o| + 1} \tag{5}$$

where α is a user-specified constant. We have found it sufficient to set $\alpha = 0$ which yields Equation 4 to a simple (equal weighted) average.

$\mathbf{q}$ assignment We assign $\mathbf{q}$ by a simple voting scheme. For each output element s_o, we gather a set of votes $\{\mathbf{q}(s_i)\}$, where each s_i is matched to s_o for a certain overlapping neighborhood determined in the search step (see Figure 4). Then we choose the one that has the minimum sum of distance across the vote set $\{\mathbf{q}(s_i)\}$:

$$\mathbf{q}(s_o) = \arg\min_{\mathbf{q}(s_i)} \sum_{s_{i'} \in \{s_i\}} |\mathbf{q}(s_i) - \mathbf{q}(s_{i'})|^2 \tag{6}$$

where $s_{i'}$ runs through the set of elements $\{s_i\}$ matched to s_o during the search step. Essentially, what we are trying to do is to find a $\mathbf{q}(s_o)$ that is closest to the arithmetic average of $\{\mathbf{q}(s_i)\}$; this is very similar to the use of a discrete solver [Han et al. 2006] for solving a least squares problem [Kwatra et al. 2005].

Discussion In the assignment steps we use blend for $\mathbf{p}$ (Equation 4) but selection for $\mathbf{q}$ (Equation 6). (In some sense, the former is analogous to the least squares solver [Kwatra et al. 2005] while the latter the discrete k-coherence solver [Han et al. 2006].) The main reason is that blend works better than selection for $\mathbf{p}$, but might not be suitable for all $\mathbf{q}$ attributes. For example, the orientation, shape or type information might not be meaningfully blended. Furthermore, to apply k-coherence acceleration (Section 4.3), we will have to copy instead of blending the $\mathbf{q}$ information.

4 Advanced Features

Here, we describe several advanced features of our method beyond the core algorithm presented in Section 3.

4.1 Synthesis control

Even though texture synthesis can automatically produce a stationary output, for realistic effects, it is usually desirable to control certain global aspects of the synthesis process. This synthesis control has appeared in prior methods, e.g. controllable [Lefebvre and

Hoppe 2005] or globally-varying [Zhang et al. 2003; Wei et al. 2008] synthesis. Here, we describe several synthesis controls that we have found useful in producing our results.

Overall shape Given a user specified output domain shape (Section 3.1), we would like the synthesis process to comply with this as much as possible, i.e. put elements inside instead of outside the overall shape, and transfer boundary/interior output elements from similarly configured boundary/interior input elements. We can achieve this with a density map $\mathbf{c}$ that shaped as the output domain with values within the range $[0, 1]$, where higher value indicates larger probability of element appearance.

In the search step, we find the input neighborhood $\mathbf{z}_s$ that minimizes not only the usually texture (dis)similarity $|\mathbf{x}_s - \mathbf{z}_s|^2$ but also an additional term $\lambda|\mathbf{c}_s - \mathbf{z}_s|^2$, where λ is a relative weight and $\mathbf{c}_s$ the sampled density value of $\mathbf{x}_s$. Specifically,

$$|\mathbf{c}_s - \mathbf{z}_s|^2 = \sum_{s'_i \in \mathbf{z}_s} |c(s'_i) - 1|^2 \tag{7}$$

where $\{c(s'_i)\}$ are sampled density $\mathbf{c}$ values at positions $\{\mathbf{p}(s_o) + \mathbf{p}(s'_i) - \mathbf{p}(s_i), s'_i \in \mathbf{z}_s\}$. Essentially, we shift the entire input neighborhood $\mathbf{z}_s$ to the center location $\mathbf{p}(s_o)$ and sample $\mathbf{c}$ at the shifted element positions.

Local orientation The user can also optionally specify a local orientation of the output texture so that the output patterns are aligned with the user choice instead of the default global coordinate frame. This allows the production of more interesting results, e.g. oriented flow patterns as in [Ijiri et al. 2008]. Algorithmically, this can be easily achieved by using the local instead of the global frame at each element during each step of our algorithm, including neighborhood metric, search, assignment, and initialization. Note that the incorporation of local frames into a texture optimization framework has been done in prior methods, e.g. [Ma et al. 2009].

Constraints For certain application scenarios it might be desirable to maintain specific constraints, e.g. minimize penetrations for physical elements or avoid elements floating in the mid air. Even though texture synthesis cannot completely guarantee all these constraints, it can usually be tuned to produce visually plausible results. For inter-penetration, we have found that minimizing neighborhood dissimilarity in Equation 2 would also lead to less penetrations. For other kinds of constraints, we have found it effective to restrain the kinds of input elements that can be transferred to the constrained output regions. (This is a commonly used method in texture synthesis, e.g. for volumetric layers [Owada et al. 2004].) For example, to reduce the chance of elements floating in the mid air, during the search step we only select input floor elements for output floor elements. During the assignment step, we maintain the vertical elevation of these floor elements to be invariant while minimizing other energy terms as described in Section 3.5. Similar mechanisms can be applied to other kinds of constraints, as we will show in Section 5.

4.2 Initialization

White noise This is perhaps the simplest and most flexible initialization method, by randomly copying elements from the input to the output domain. One downside of such a white noise initialization, though, is that it may require an excessive number of iterations to converge via our optimization procedure. It could also get stuck in a local minimum, causing unsatisfactory element distribution in certain regions of the output.

Patch copy To address the deficiencies of white noise initialization, we have found another strategy, patch copy, which works quite well. Patch-base synthesis has demonstrated to be effective for image textures (see e.g. [Liang et al. 2001; Efros and Freeman 2001] and the survey in [Wei et al. 2009]). Here, we apply a similar method for initialization. We first divide the input exemplar and output region into uniform grids, with each grid cell corresponding to a patch of elements, and then randomly copy input patches into output grids, just like patch-based image synthesis. In addition, when copying patches we take into account the user controls (Section 4.1), such as aligning patches with local orientations as well as preferring input patches with similar boundary conditions to the output region.

4.3 Acceleration by k-coherence

Since our method copies the $\mathbf{q}$ information from input to output elements, we can apply k-coherence [Tong et al. 2002] throughout our entire algorithm. The main difference between our method and the original k-coherence method is that we have to deal with irregularly placed samples. However, this problem has been addressed in the context of irregular mesh vertices [Han et al. 2006], and we could adopt a similar strategy here. Specifically, during the pre-process, we can build a similarity-set for each input sample via our search-step as described in Section 3.4. At run-time, we build the candidate set by collecting the similarity sets from all the neighboring elements, with the offset part properly computed by the recorded element pairs (Section 3.2).

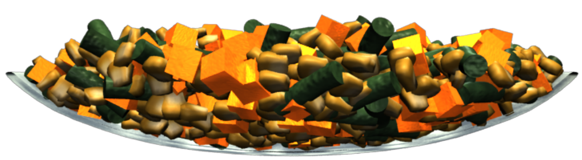
5 Results

5.1 Element distribution

Our method can produce a variety of element distributions with different attributes, such as dimensionality (2D/3D), volume/surface synthesis, regular/semi-regular/irregular distribution, number of element types, variations in element size/shape/color/texture, output domain size/shape/orientation, and artistic/realistic phenomena. Since our method is data driven, we can handle all these by simply using different input exemplars and output domains. We wish to emphasize that the input and output specifications are more or less de-coupled, i.e. the same input exemplar can be used for different output domains, and vice versa (see Figure 6). This is a key factor facilitating easy and flexible usage of our method.

Input exemplar properties Using input exemplars with different properties, our method can produce a variety of different results as shown in Figure 1 & Figure 6. We begin with the simplest but also very common case of one type of elements, e.g. Figure 1a, 6a, and 6e. But even such one-element-type distributions may have certain properties that cannot be easily captured by procedural or physical simulation methods. For example, the user might prefer to arrange a stack of plums in a near-regular configuration (Figure 1a), or a collection of carrots in specific orientations (Figure 6f, 6g, and 6h). Notice that these examples cannot be produced by physical simulation (e.g. dropping objects until they come to rest) as the outputs are unlikely to reach the desired user intention. One possibility is to manually place the elements, but this could quickly become very tedious for sufficiently large outputs. Using our method, the user only needs to manually place a small input exemplar and our method will automatically produce the desired output. The bananas (Figure 6a) present another interesting case due to their unique long and curvy shapes. For this case, we generated the input via physical simulation to show that our method can produce visually realistic outputs via physically validated input. More interesting distribution can be produced by multiple types of elements with different sizes and shapes, e.g. a dish containing corns, diced carrots, and green beans (Figure 1c).

Output domain properties In addition to the input exemplar properties like element type and distribution, the user can also specify the output domain properties, including size, shape, and orientation field, to achieve different effects. Beyond physically plausible shapes like a stack, a box, a pile, or a bowl as shown in Figure 1 and 6, the user can also specify a more complex or interesting shape as a sculpture (Figure 1b), a tai-chi pattern (Figure 6f), a knot (Figure 6h), or a building (Figure 6k). Our method can also be applied to both volume (e.g. Figure 1) and surface/shell (e.g. Figure 6k, 6l, and 6m) synthesis. Note that these results span both physically realistic as well as artistic effects. As noted in [Cho et al. 2007], physical simulation might produce output distributions that look flat or boring. To produce visually more appealing effects, it is often desirable to have the output in a physically unstable or implausible configuration. Cho et al. [2007] achieved this via certain ad-hoc approaches, e.g. stopping physical simulation in the middle prior to completion (Figure 10 in [Cho et al. 2007]) or using repeated skimming and an up-side-down collision mesh (Figure 15 in [Cho et al. 2007]). Our method can easily produce the desired effect in a more principled and more controllable manner by simply using the proper output domains.



(a) with boundary handling

(b) no boundary handling

Figure 5: Boundary condition comparisons. Shown here are the profile views for the texture in Figure 1c.

Boundary handling Properly boundary handling is important to produce satisfactory results for certain discrete element textures that exhibit different distributions near and away from the overall boundaries, e.g. floor or box sides. Our experimental results indicate that these boundary conditions can be adequately handled by our control mechanisms described in Section 4.1. Without such mechanisms, the synthesis results might exhibit poor boundary conditions, as shown in Figure 5. We wish to emphasize that our method does not require all possible output boundary configurations to be present in the input exemplar; as shown in Figure 1 and 6, even though the output can contain different boundary shapes and orientations not present in the simpler input exemplars, the combination of local orientation and boundary handling can still produce satisfactory results.

5.2 Distribution editing

As an added benefit, our method can also be applied for editing discrete element textures, for not only individual element properties $\mathbf{q}$ but also their distributions $\mathbf{p}$. All these can be achieved by the very same algorithms that we have built for synthesizing discrete element textures, especially the neighborhood metric. Texture editing has been shown to be useful for a variety of application scenarios (see e.g. [Brooks and Dodgson 2002; Matusik et al. 2005; Zhou et al. 2006; Liu et al. 2009; Cheng et al. 2010]). Our method follows this line of thinking, but can achieve certain effects that may benefit from the explicit knowledge of the discrete elements.

Figure 7 demonstrates a potential example. Given an input pattern consisting of discrete elements, we aim to use our method to edit the element properties $\mathbf{q}$ and distributions $\mathbf{p}$ to produce more versatile effects. The user may simply select a typical element, performs some edits, and our method will automatically propagate relevant edits to all other elements with similar neighborhoods to the user interacted element. Note that without our automatic propagation, it would be quite tedious for the user to manually repeat the same edits to all relevant elements.

5.3 Usage and parameters

Input preparation Unlike other texture synthesis applications where the input exemplars can be obtained directly (e.g. downloading an image), for discrete element textures the user would have to do some work to produce the input exemplars, including both the individual elements and their distribution. For the results shown in this paper, we prepare the elements via standard modeling tools (e.g. Maya) and distribute them either manually or by simple simulation. For the modeling part, we have found it sufficient to make just one element for each type and the quality seems to work quite well for human perceptions [Ramanarayanan et al. 2008]. If additional element prototypes are desired, we have found it sufficient to slightly perturb the prototype element properties (e.g. geometry or color) via procedural noise. For the distribution part, since the input exemplar is usually quite small, manual placement seems quite feasible (e.g. the inputs for Figure 1a, 6e & 6i). It is also possible to use simple physical simulation for the input distribution for more random or physically realistic effects, even for outputs that might not be easy to produce via simulation (e.g. Figure 1b).

Parameters Similar to prior texture synthesis methods, one of the most important parameters is the neighborhood size. In our results we have found it sufficient to use a neighborhood size containing roughly 1- to 3-ring neighbors ($\sim 3^n$ to 7^n neighborhood in n-D pixel synthesis) depending on whether the pattern is more stochastic or structured. Other important parameters include α (for Equation 2) and λ (for Equation 7), for which we set to be of the same order of magnitude as the average distance between elements. (For example, if the average element distance is 0.01 we just set α and $\lambda \in [0.005\ 0.05]$.)

Regarding speed, our current implementation takes about seconds to minutes to generate each result, containing number of elements in the range $500 \sim 2000$. We have found this fast enough to produce results shown in the paper, even though we have not attempted any further speed optimization beyond the basic k-coherence introduced in Section 4.3.

6 Limitations and Future Work

Even though our method can produce visually plausible results, it cannot guarantee certain domain specific properties, e.g. complete obedience to physical laws like gravity or shape penetration. If such properties need to be more strongly enforced, one possibility is to add them as extra energy terms into our current framework.

Our approach synthesizes element distribution only but not the individual elements, for which we rely on user inputs. It will be interesting to devise methods that can more automatically obtain the individual elements, e.g. 2.1d textons [Ahuja and Todorovic 2007], vector primitives [Hurtut et al. 2009], or even 3D geometry.

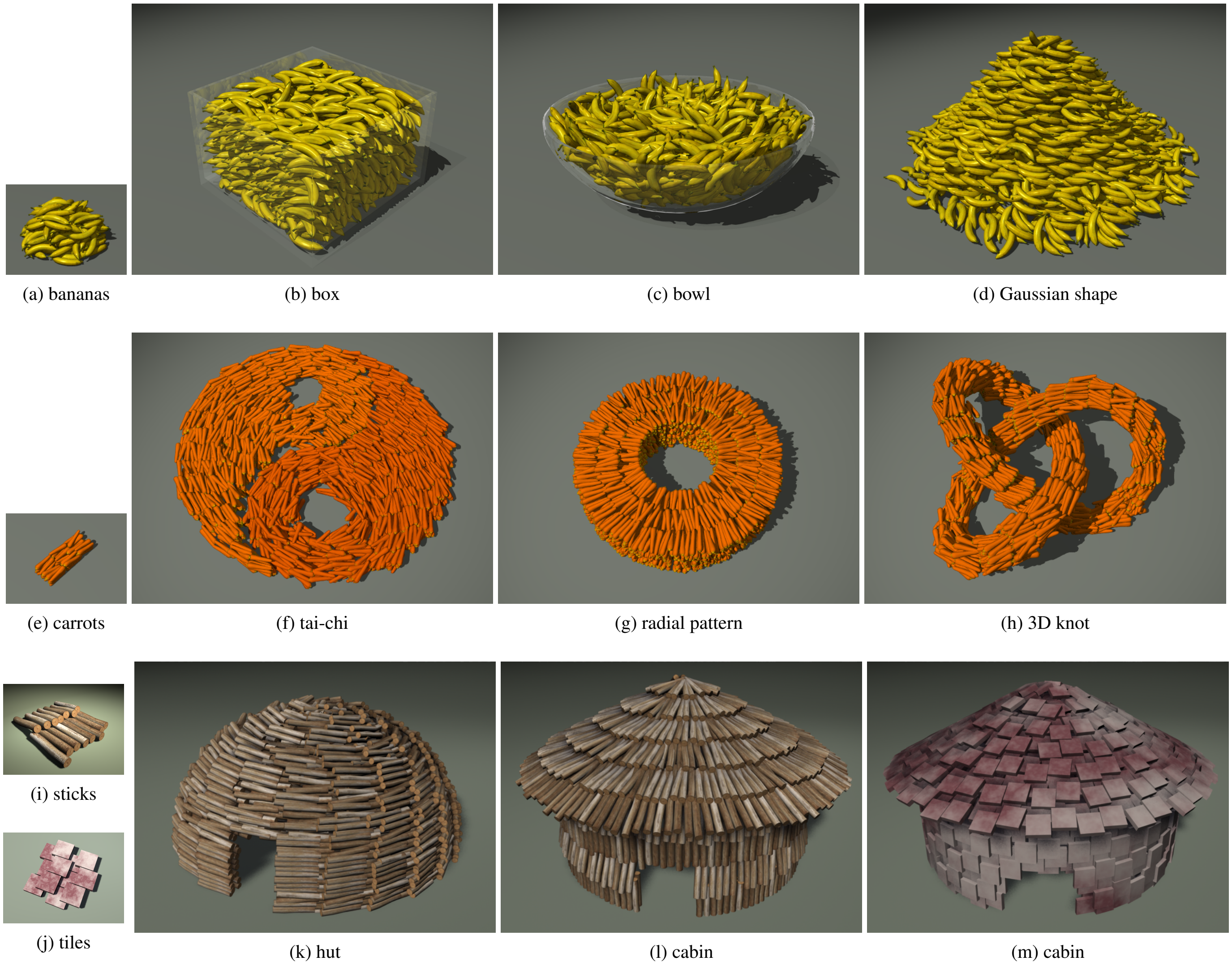


Figure 6: Element distribution. The input exemplars are shown as smaller images, with the corresponding synthesis results shown on larger ones. Each exemplar in (a), (e), and (i) is used to produce multiple outputs with different sizes, shapes, or orientation fields. The same output model is used to produce different results in (l) and (m) via different exemplars in (i) and (j).

We also rely on the user input for the overall output shape. On one hand this provides the flexibility for the users to choose whatever shapes they like, but on the other hand it may be a nuisance if the users do not feel like doing so. For the latter case it would be interesting to apply more automatic methods to determine the output shape, e.g. what would the output shape be for a pile of potatoes?

We have only tried to apply our method to static but not dynamic element distributions. Based on texture optimization, we believe that our basic framework can be applied for frame coherent animation effects as in [Kwatra et al. 2005; Kyriakou and Chrysanthou 2008]. The really interesting issue here is on what kinds of input exemplars to specify; dynamic inputs would be easier for our method to work with, but static inputs might be more convenient and practical to obtain.

References

AHUJA, N., AND TODOROVIC, S. 2007. Extracting texels in 2.1d natural textures. *ICCV 0*, 1–8.

BARAFF, D., AND WITKIN, A. 1997. Physically based modeling: Principles and practice. In *SIGGRAPH '97 Course*.

BARLA, P., BRESLAV, S., THOLLOT, J., SILLION, F., AND MARKOSIAN, L. 2006. Stroke pattern analysis and synthesis. In *EUROGRAPH '06: EUROGRAPH 2006 papers*, vol. 25, Citeseer, 663–671.

BROOKS, S., AND DODGSON, N. 2002. Self-similarity based texture editing. In *SIGGRAPH '02: SIGGRAPH 2002 Papers*, 653–656.

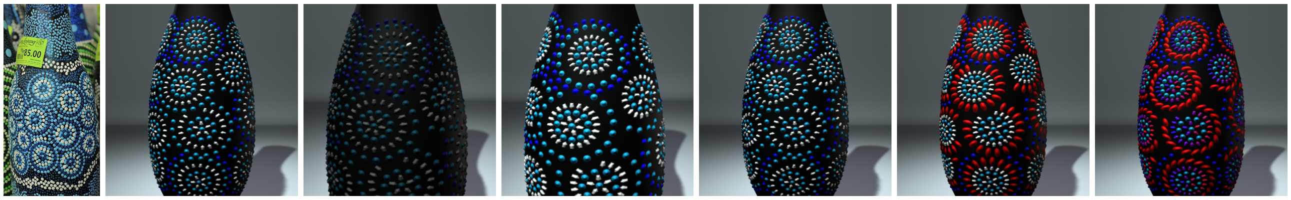
CHENG, M.-M., ZHANG, F.-L., MITRA, N. J., HUANG, X., AND HU, S.-M. 2010. Finding approximately repeated scene elements for image editing. In *SIGGRAPH '10*.

CHO, J. H., XENAKIS, A., GRONSKY, S., AND SHAH, A. 2007. Course 6: Anyone can cook: inside ratatouille's kitchen. In *SIGGRAPH 2007 Courses*.

DISCHLER, J., MARITAUD, K., LÉVY, B., AND GHAZANFARPOUR, D. 2002. Texture particles. *Computer Graphics Forum 21*, 401–410.

EBERT, D. S., MUSGRAVE, K. F., PEACHEY, D., PERLIN, K., AND WORLEY, S. 2002. *Texturing & Modeling: A Procedural Approach*. Morgan Kaufmann.

EFROS, A. A., AND FREEMAN, W. T. 2001. Image quilting for texture synthesis and transfer. In *SIGGRAPH '01: SIGGRAPH 2001 Papers*, 341–346.



(a) photo (b) input (c) user edit 1 element (d) propagated change (e) edit $\mathbf{p}$ (f) change element (g) edit $\mathbf{q}$

Figure 7: Discrete element texture editing. Inspired by a real-world example (a), we aim to enhance the pattern quality from the input (b). Since the original pattern is a bit boring with only little dots, the user first changes one element shape. Our method then automatically propagates that change to all other elements with similar neighborhoods. The user then goes on to edit other element properties, including both positions $\mathbf{p}$ and attributes $\mathbf{q}$ such as color, size, and shape.

Efros, A. A., and Leung, T. K. 1999. Texture synthesis by non-parametric sampling. In *ICCV '99*, 1033.

Gal, R., Sorkine, O., Popa, T., Sheffer, A., and Cohen-Or, D. 2007. 3d collage: expressive non-realistic modeling. In *NPAR '07: Proceedings of the 5th international symposium on Non-photorealistic animation and rendering*, 7–14.

Han, J., Zhou, K., Wei, L.-Y., Gong, M., Bao, H., Zhang, X., and Guo, B. 2006. Fast example-based surface texture synthesis via discrete optimization. *Vis. Comput. 22*, 9, 918–925.

Hausner, A. 2001. Simulating decorative mosaics. In *SIGGRAPH '01: SIGGRAPH 2001 Papers*, 573–580.

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

Ijiri, T., Mech, R., Igarashi, T., and Miller, G. 2008. An example-based procedural system for element arrangement. In *EUROGRAPH '08: EUROGRAPH 2008 papers*, vol. 27, 429–436.

Jodoin, P.-M., Epstein, E., Granger-Piché, M., and Ostromoukhov, V. 2002. Hatching by example: a statistical approach. In *NPAR '02: Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering*, 29–36.

Kim, J., and Pellacini, F. 2002. Jigsaw image mosaics. In *SIGGRAPH '02: SIGGRAPH 2002 Papers*, 657–664.

Kim, T., Thürey, N., James, D., and Gross, M. 2008. Wavelet turbulence for fluid simulation. In *SIGGRAPH '08: SIGGRAPH 2008 Papers*, 50:1–6.

Kim, S., Maciejewski, R., Isenberg, T., Andrews, W. M., Chen, W., Sousa, M. C., and Ebert, D. S. 2009. Stippling by example. In *NPAR'09*.

Kopf, J., Fu, C.-W., Cohen-Or, D., Deussen, O., Lischinski, D., and Wong, T.-T. 2007. Solid texture synthesis from 2d exemplars. In *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*, 2.

Kwatra, V., Essa, I., Bobick, A., and Kwatra, N. 2005. Texture optimization for example-based synthesis. In *SIGGRAPH '05: SIGGRAPH 2005 Papers*, 795–802.

Kyriakou, M., and Chrysanthou, Y. 2008. Texture synthesis based simulation of secondary agents. In *Motion in Games*, 1–10.

Lagae, A., and Dutré, P. 2005. A procedural object distribution function. *ACM Trans. Graph. 24*, 4, 1442–1461.

Lefebvre, S., and Hoppe, H. 2005. Parallel controllable texture synthesis. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, 777–786.

Lerner, A., Chrysanthou, Y., and Lischinski, D. 2007. Crowds by example. *Computer Graphics Forum 26*, 655–664.

Liang, L., Liu, C., Xu, Y.-Q., Guo, B., and Shum, H.-Y. 2001. Real-time texture synthesis by patch-based sampling. *ACM Trans. Graph. 20*, 3, 127–150.

Liu, Y., Wang, J., Xue, S., Tong, X., Kang, S. B., and Guo, B. 2009. Texture splicing. In *Pacific Graphics '09*.

Ma, C., Wei, L.-Y., Guo, B., and Zhou, K. 2009. Motion field texture synthesis. In *SIGGRAPH Asia 2009*, 110:1–8.

Matusik, W., Zwicker, M., and Durand, F. 2005. Texture design using a simplicial complex of morphable textures. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, ACM, New York, NY, USA, 787–794.

Merrell, P., and Manocha, D. 2008. Continuous model synthesis. In *SIGGRAPH Asia '08: ACM SIGGRAPH Asia 2008 papers*, 1–7.

Narain, R., Sewall, J., Carlson, M., and Lin, M. 2008. Coupling physically-based and procedural methods for animating turbulent fluids. In *SIGGRAPH Asia '08: ACM SIGGRAPH Asia 2008 Papers*.

Narain, R., Golas, A., Curtis, S., and Lin, M. 2009. Aggregate dynamics for dense crowd simulation. In *SIGGRAPH Asia '09*.

Owada, S., Nielsen, F., Okabe, M., and Igarashi, T. 2004. Volumetric illustration: designing 3d models with internal textures. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, 322–328.

Peytavie, A., Galin, E., Merillou, S., and Grosjean, J. 2009. Procedural generation of rock piles using aperiodic tiling. In *Pacific Graphics '09*.

Ramanarayanan, G., Bala, K., and Ferwerda, J. A. 2008. Perception of complex aggregates. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, 60:1–10.

Tong, X., Zhang, J., Liu, L., Wang, X., Guo, B., and Shum, H.-Y. 2002. Synthesis of bidirectional texture functions on arbitrary surfaces. In *SIGGRAPH '02: SIGGRAPH 2002 Papers*, 665–672.

Turk, G. 2001. Texture synthesis on surfaces. In *SIGGRAPH '01: SIGGRAPH 2001 Papers*, 347–354.

Wang, L., Yu, Y., Zhou, K., and Guo, B. 2009. Example-based hair geometry synthesis. In *SIGGRAPH '09: SIGGRAPH 2009 Papers*, 56:1–9.

Wei, L.-Y., Han, J., Zhou, K., Bao, H., Guo, B., and Shum, H.-Y. 2008. Inverse texture synthesis. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, 1–9.

Wei, L.-Y., Lefebvre, S., Kwatra, V., and Turk, G. 2009. State of the art in example-based texture synthesis. In *Eurographics '09 State of the Art Report*, 93–117.

Wu, Q., and Yu, Y. 2004. Feature matching and deformation for texture synthesis. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, 364–367.

Zhang, J., Zhou, K., Velho, L., Guo, B., and Shum, H.-Y. 2003. Synthesis of progressively-variant textures on arbitrary surfaces. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, 295–302.

Zhou, K., Huang, X., Wang, X., Tong, Y., Desbrun, M., Guo, B., and Shum, H.-Y. 2006. Mesh quilting for geometric texture synthesis. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, 690–697.

Zhou, H., Sun, J., Turk, G., and Rehg, J. M. 2007. Terrain synthesis from digital elevation models. *IEEE Transactions on Visualization and Computer Graphics 13*, 4, 834–848.