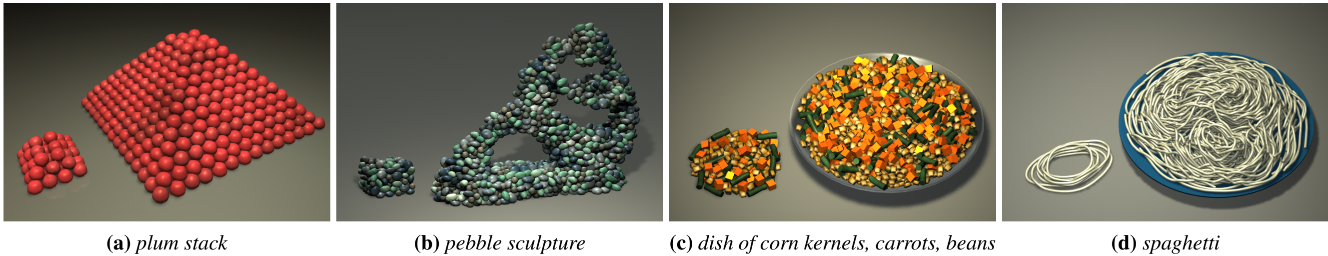
Discrete Element Textures

Chongyang Ma[1,3] [1]Tsinghua University

Li-Yi Wei[2] [2]Microsoft Research

Xin Tong[3] [3]Microsoft Research Asia



(a) *plum stack* **(b)** *pebble sculpture* **(c)** *dish of corn kernels, carrots, beans* **(d)** *spaghetti*

Figure 1: *Discrete element textures. Given a small input exemplar (left within each image), our method synthesizes a corresponding output with user specified coarse-scale domain (right within each image). Using a data driven approach, we can achieve a variety of effects, including (a) regular distribution, (b) output with different domain shape and boundary conditions from the input, (c) mixture of different elements, and (d) deformable and elongated shapes.*

Abstract

A variety of phenomena can be characterized by repetitive small scale elements within a large scale domain. Examples include a stack of fresh produce, a plate of spaghetti, or a mosaic pattern. Although certain results can be produced via manual placement or procedural/physical simulation, these methods can be labor intensive, difficult to control, or limited to specific phenomena.

We present discrete element textures, a data-driven method for synthesizing repetitive elements according to a small input exemplar and a large output domain. Our method preserves both individual element properties and their aggregate distributions. It is also general and applicable to a variety of phenomena, including different dimensionalities, different element properties and distributions, and different effects including both artistic and physically realistic ones. We represent each element by one or multiple samples whose positions encode relevant element attributes including position, size, shape, and orientation. We propose a sample-based neighborhood similarity metric and an energy optimization solver to synthesize desired outputs that observe not only input exemplars and output domains but also optional constraints such as physics, orientation fields, and boundary conditions. As a further benefit, our method can also be applied for editing existing element distributions.

Keywords: discrete element, texture, analysis, synthesis, sampling, editing, data driven

Links: DL PDF

1 Introduction

A variety of phenomena can be characterized by a distinctive large scale domain with repetitive small scale elements. Some common examples include a stack of fresh produce, a plate of spaghetti, or a mosaic pattern. Due to the potential scale and complexity of such phenomena, it is desirable to have a general and efficient method for users to easily specify and synthesize these element distributions for different application scenarios.

Manual placement, which is flexible enough to achieve many effects, can be too tedious with current modeling tools for sufficiently large or complex distributions. An alternative is physical simulation, for which the users specify certain input controls (e.g. initial state and/or boundary conditions) and simply let the algorithm run its course to produce results. The primary advantage of physical simulation is fidelity to realism. However, such methods can be hard to control, since producing the desired output might require the user 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 are only applicable to specific distributions (e.g. Poisson disk [Lagae and Dutré 2005]) or phenomena (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.

To achieve the goal of generality, efficiency, and easy usage, we adopt a data-driven methodology. We call our approach *discrete element textures*, which analyzes an input exemplar and synthesizes the corresponding discrete elements within a given output domain (Figure 2). Unlike prior data-driven methods that might produce undesirable individual elements (Figure 3) or aggregate distributions (Figure 4), our method preserves both. Since the user has maximum flexibility in specifying both the input exemplar and the output domain, 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), complexities (e.g. round and rigid pebbles or elongated and deformable spaghetti) and distributions (e.g. regular/semi-regular/irregular), different numbers of element types (e.g. a plate of mixed vegetables), as well as physically realistic or artistic phenomena (e.g. a physical pile of objects or a decorative mosaic pattern). In particular, even though our method is data driven, it can still produce physical effects (e.g. deformable spaghetti as in Figure 1d).

We observe that overall element distributions are closely related to individual element properties, such as position, size, shape, and orientation. Thus, treating each element as a point sample, as com-

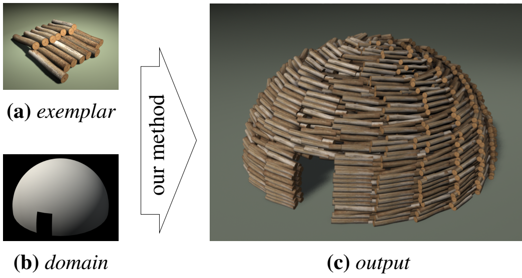


Figure 2: *Given an input exemplar (a) and output domain (b), our method automatically synthesizes the corresponding output (c).*

monly practiced in previous methods, might not work well for sufficiently complex element shapes or distributions. Our key idea is to represent each element by multiple samples so that all relevant attributes including position, size, shape, and orientation can be fully encoded by sample positions only. Upon this core representation, we build a sample-based neighborhood similarity metric as well as an energy formulation to express the desired combinations of individual element samples and their overall distributions. We minimize this energy function through an iterative optimization solver (following [Kwatra et al. 2005]) to produce the desired outputs. A primary advantage of this energy optimization framework is the flexibility in incorporating optional application specific constraints through additional energy terms (e.g. for orientation fields and boundary conditions) or solver steps (e.g. for physics).

As an added benefit, our method can also be applied for editing element distributions. Specifically, users just need to change a few elements, and our method will automatically propagate such changes to all other elements with similar texture neighborhoods, relieving users from the potential tedious chore of manual repetitions. This editing application is possible thanks to the framework we developed for direct synthesis.

2 Previous Work

Multi-scale computation 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], hair strands [Wang et al. 2009], crowds [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; Wei and Levoy 2000], 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]), there is no guarantee that the individual elements would be preserved. Thus, the synthesized textures can have elements that are broken or merged (Figure 3).

Geometry synthesis Our method is also related to geometry synthesis, especially those via example-based texturing methods such as surface meshes [Zhou et al. 2006], volumetric models [Bhat et al. 2004; 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].

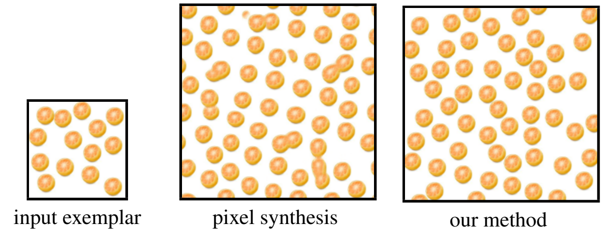


Figure 3: *Comparison with pixel-based synthesis. The pixel synthesis result is produced by combining discrete optimization [Han et al. 2006] with a texton mask [Zhang et al. 2003].*

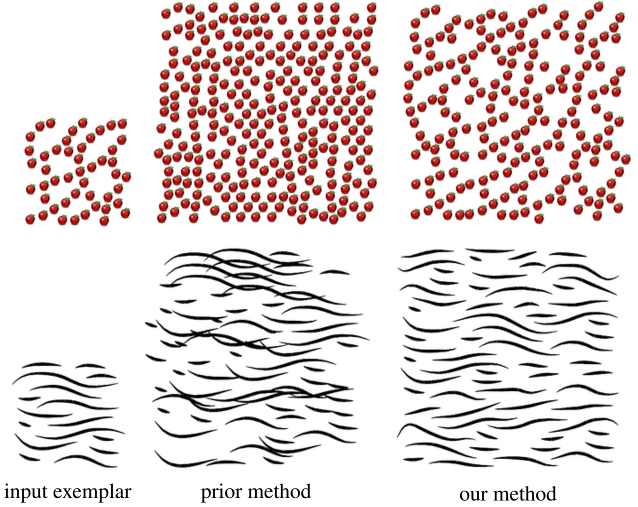
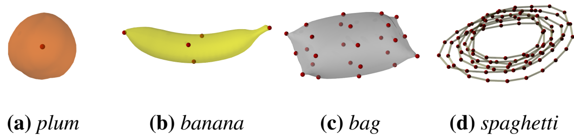


Figure 4: *Comparison with prior element synthesis methods. Results in the middle column are produced by [Dischler et al. 2002] (top) and [Ijiri et al. 2008] (bottom).*

Element packing There exist methods that pack a set of discrete elements into a specific domain or shape, such as geometric elements over a surface [Fleischer et al. 1995; Landreneau and Schaefer 2010], mosaic tiles [Hausner 2001; Kim and Pellacini 2002], stroke patterns [Barla et al. 2006], curves [Merrell and Manocha 2010], 3D object collage [Gal et al. 2007], aggregated particles [Jagnow et al. 2004], or rock piles [Peytavie et al. 2009]. However, the element distributions in these methods are determined via specific procedures or a semi-manual user interface, instead of generally imitating the distributions in input exemplars 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, including 1D strokes [Jodoin et al. 2002], 2D stipples [Kim et al. 2009; Martín et al. 2010], 2D particles/elements [Dischler et al. 2002; Ijiri et al. 2008; Hurtut et al. 2009], and 2D agent motions [Lerner et al. 2007; Ju et al. 2010]. However, these methods treat each element as a single sample without a comprehensive neighborhood metric or a general synthesis solver as in our method; thus they might not faithfully reproduce both overall element distributions and individual element properties, especially shape, orientation, or heterogeneous elements. For example, Dischler et al. [2002] extracted 2D textons from an input exemplar and generated



(a) *plum* **(b)** *banana* **(c)** *bag* **(d)** *spaghetti*

Figure 5: *Examples of elements and samples (shown as red points over each element model).*

the output texture by adding elements from a randomly chosen list of co-occurrences. Since co-occurrence is less general than a full neighborhood metric, this approach might not handle well structural patterns that are neither entirely stochastic nor entirely regular. Ijiri et al. [2008] synthesized 2D distributions by locally growing through 1-ring neighborhoods, but cannot handle elements with complex shapes which are closely correlated with spatial distributions. Hurtut et al. [2009] synthesized 2D non-photorealistic arrangements using a statistical model evaluated between and within different categories. The model is based on pair-wise element distance and is only suitable for isotropic stochastic distributions, not more general ones that can be structural or anisotropic. To our knowledge, the works in [Dischler et al. 2002; Ijiri et al. 2008] present the most advanced algorithm features and the strongest effects among existing methods, but as shown in Figure 4 our method still produces better results.

3 Texture Representation

Here we first describe our representation for discrete elements and texture neighborhoods. Based on these, we then describe our basic synthesis method in Section 4, followed by constrained synthesis with more advanced features in Section 5.

3.1 Element Samples

We represent each element by point samples as exemplified in Figure 5. Each sample records its position $\mathbf{p}$ and various attributes $\mathbf{q}$, including the element id identifying which element it belongs to, as well as the sample id identifying its relative position within the element. We determine the number of samples based on the element properties and place the samples over the element surface via either manual specification or standard mesh simplification with feature preservation. During synthesis, we compute only the sample ids and locations without considering any other information of the original elements, like their geometry and appearance. After synthesis, we recover the output elements by treating the output samples as control points for placing, orienting and possibly deforming the corresponding input element to the output [Shi et al. 2007].

An important benefit of our multi-sample element representation is that it allows us to absorb element shape and orientation information into sample positions. In our earlier design we treat each element as one sample, and thus usually have to incorporate shape and orientation as extra information in $\mathbf{q}$ for our neighborhood metric. This multi-sample representation facilitates a simpler and cleaner algorithm formulation. In particular, we have found that using multiple samples per element is very effective for synthesizing objects that are otherwise difficult to handle via existing element synthesis methods, such as elongated shapes. Furthermore, by using multiple samples per element, our method is able to synthesize convincing physical effects without real physical simulation, e.g. deformable shapes whose output variations are simply mimicked from the input exemplar through data driven synthesis.

As a special case of our multi-sample element representation, we have found that it is sufficient to use only one sample per element for rigid and isotropic shapes (such as the plums, pebbles, and corn kernels). We place the sample at the element centroid, and simply use the input element geometry, orientation and appearance for the corresponding output element.

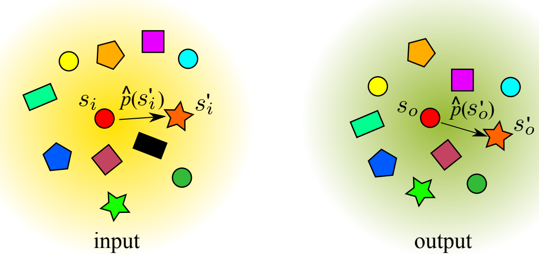


Figure 6: *Illustration for our neighborhood metric. We matched samples based on both their relative positions* $\hat{\mathbf{p}}$ *and attributes* $\mathbf{q}$ *(illustrated by shape and color). Unmatched input samples are shown in black.*

3.2 Neighborhood Metric

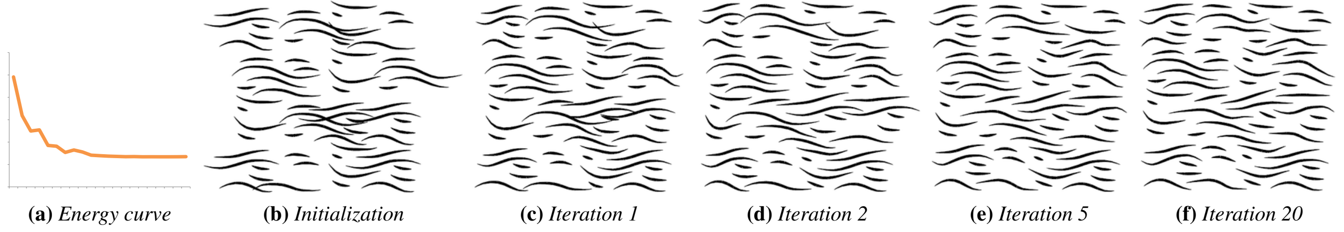
The neighborhood similarity metric is a core component for neighborhood-based texture synthesis algorithms [Wei et al. 2009]. In our method, let $\mathbf{n}(s)$ denote the spatial neighborhood around a sample s, constructed by taking the union of all samples within its spatial extent defined by a user specified neighborhood size. We measure the distance $|\mathbf{n}(s_o) - \mathbf{n}(s_i)|^2$ between the neighborhoods of two samples s_o and s_i via the following formula:

$$|\mathbf{n}(s_o) - \mathbf{n}(s_i)|^2 = \sum_{s'_o \in \mathbf{n}(s_o)} \left|\hat{\mathbf{p}}(s'_o) - \hat{\mathbf{p}}(s'_i)\right|^2 + \alpha \left|\mathbf{q}(s'_o) - \mathbf{q}(s'_i)\right|^2 \tag{1}$$

where s'_o runs through all samples $\in \mathbf{n}(s_o)$, $s'_i \in \mathbf{n}(s_i)$ is the "matching" sample 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 s), and α is the relative weight between the $\mathbf{p}$ and $\mathbf{q}$ distance terms. See Figure 6.

Intuitively, what Equation 1 tries to achieve is (1) align the two neighborhoods $\mathbf{n}(s_o)$ and $\mathbf{n}(s_i)$, (2) match up their samples 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 from further consideration, and repeat the process to find the next pair until $\mathbf{n}(s_o)$ runs out of samples. We prevent $\mathbf{n}(s_i)$ from running out of samples before $\mathbf{n}(s_o)$ by not presetting its spatial extent, essentially giving $\mathbf{n}(s_i)$ an infinite size. We have found that the heuristic above works well in practice, and provides similar quality to a more rigorous but much slower approach that considers all possible pair matchings (s'_i, s'_o) by brute force. We match only samples with identical object id and sample id to avoid changing topologies of element shapes.

Discussion For traditional texture synthesis that has fixed sample (e.g. pixel/voxel/vertex) positions $\mathbf{p}$, the neighborhood measure can be easily defined by either a simple sum-of-squared differences (SSD) of the attributes $\mathbf{q}$ (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 in Equation 1. Our metric also bears similarity to the Earth Mover's Distance (EMD) [Rubner et al. 2000], which measures distances between two groups of "clusters" that are analogous to our "samples". However, EMD is designed for partial matching between clusters whereas we focus on one-to-one mappings between samples.



(a) *Energy curve* **(b)** *Initialization* **(c)** *Iteration 1* **(d)** *Iteration 2* **(e)** *Iteration 5* **(f)** *Iteration 20*

Figure 7: *Iteration process. Here we show the energy curve as well as results after different numbers of iterations using the input exemplar in Figure 4.*

4 Basic Synthesis

Given an input exemplar $\mathcal{I}$ and an output domain with user specified size, shape, and optional properties such as orientation field and boundary condition, our goal is to synthesize an output $\mathcal{O}$ that contains detailed elements similar to $\mathcal{I}$ while observing the output domain properties (Figure 2). We formulate this as an optimization problem [Kwatra et al. 2005] via the following energy function:

$$E(\mathcal{O};\mathcal{I}) = \sum_{s_o \in \mathcal{O}} |\mathbf{n}(s_o) - \mathbf{n}(s_i)|^2 + \Theta(\mathcal{O};\mathcal{I}) \quad (2)$$

where the first term measures the similarity between the input exemplar $\mathcal{I}$ and the output $\mathcal{O}$ via our local neighborhoods metric as defined in Equation 1. Specifically, for each output sample $s_o \in \mathcal{O}$, we find the corresponding input sample $s_i \in \mathcal{I}$ with the most similar neighborhood (according to Equation 1), and sum their squared neighborhood differences. E also contains optional application specific energy terms represented by Θ such as boundary conditions. Our goal is to find an output $\mathcal{O}$ with a low energy value.

Below we describe our basic solver assuming Θ is null, and leave details about constrained synthesis to Section 5. Even though there are multiple potential solvers for Equation 2, we follow the EM-like methodology in [Kwatra et al. 2005] because of its high quality and generality with different options in Θ. As summarized in Pseudocode 1, our basic solver gradually improves the neighborhood similarity term by iterating the two steps: **search** for the most similar input neighborhood for each output sample and **assign** the information from the matched neighborhoods to the output. This will gradually decrease E while improving output quality, as exemplified in Figure 7. The main difference between prior texture synthesis methods and ours is that unlike the former where the position information $\mathbf{p}$ is given (e.g. pixels, vertices, or voxels) and only the attribute information $\mathbf{q}$ needs to be determined (e.g. colors), we have to solve for both $\mathbf{p}$ and $\mathbf{q}$ during synthesis.

4.1 Initialization

Patch-based synthesis is well known to be effective for image textures (see the survey in [Wei et al. 2009]). Here, we apply a similar method for initialization. We first divide the input exemplar into patches, and then randomly copy these patches into the output domain. We set the patch size to be identical to the user selected neighborhood size (Section 3.2). To avoid partial/broken objects, we always copy integral elements. In addition, when copying patches we take into account the user controls (Section 5), such as aligning patches with local orientations as well as preferring input patches with similar boundary conditions to the output region. We have also experimented with other initialization methods such as white noise (random copying elements) and incremental add [Jodoin et al. 2002; Ijiri et al. 2008]. Even though our solver can produce good results from these alternative initializations, they usually exhibit slower convergence than patch copy.

function $\mathcal{O} \leftarrow$ DiscreteElementTextureSynthesis($\mathcal{I}$)
  // *$\mathcal{O}$: output distribution*
  // *$\mathcal{I}$: input exemplar*
  $\mathcal{O} \leftarrow$ Initialize($\mathcal{I}$) // *Section 4.1*
  iterate until convergence **or** enough # of iterations reached
    $\{\mathbf{n}(s_i)\} \leftarrow$ Search($\mathcal{O}, \mathcal{I}$) // *search phase*
    Assign($\{\mathbf{n}(s_i)\}, \mathcal{O}$) // *assignment phase*
    extra solver steps // *Section 5*
  end
  return $\mathcal{O}$

function $\{\mathbf{n}(s_i)\} \leftarrow$ Search($\mathcal{O}, \mathcal{I}$) // *Section 4.2*
  foreach element $s_o \in \mathcal{O}$
    $\mathbf{n}(s_o) \leftarrow$ output neighborhood around s_o
    $\mathbf{n}(s_i) \leftarrow$ find most similar neighborhood for $s_i \in \mathcal{I}$ to $\mathbf{n}(s_o)$
  end
  return $\{\mathbf{n}(s_i)\}$

function Assign($\{\mathbf{n}(s_i)\}, \mathcal{O}$) // *Section 4.3*
  foreach output element $s_o \in \mathcal{O}$
    $\mathbf{p}(s_o) \leftarrow$ least squares from predicted positions
    $\mathbf{q}(s_o) \leftarrow$ select the vote that minimizes the energy function
  end

Pseudocode 1: *Discrete element texture synthesis.*

4.2 Search Step

During the search step, we find, for each output sample s_o, the best matching input sample s_i with the most similar neighborhood, i.e. minimizing the energy value in Equation 1. This search can be conducted by exhaustively examining every input sample, but this can be computationally expensive. Instead, we adopt k-coherence search [Tong et al. 2002] for constant time computation.

The basic idea behind k-coherence search is to build a set of other input samples with neighborhoods similar to each input sample during a pre-process, and use that information to restrict the search space at run time. (See [Wei et al. 2009] for a tutorial.) 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 brute force search step as described above. At run-time, we build the candidate set by collecting the similarity sets from all the neighboring samples, with the offset part properly computed by the recorded sample pairs (Section 3.2).

4.3 Assignment Step

$\mathbf{p}$ assignment Here we determine the output sample positions $\{\mathbf{p}(s_o)\}_{s_o \in \mathcal{O}}$ to minimize Equation 2. At the beginning of the assignment step, we have multiple input neighborhoods $\{\mathbf{n}(s'_i)\}$ overlapping every output sample s_o, where $\mathbf{n}(s'_i)$ is the matching input neighborhood for output sample s'_o as determined in the

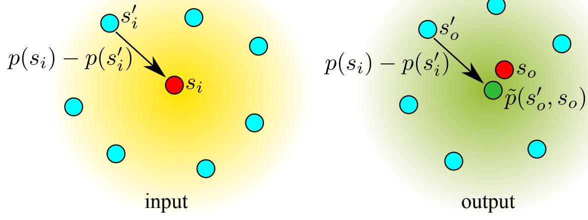


Figure 8: *Illustration for the assignment step. Each neighboring sample s_o' (cyan) of s_o (red) provides a predicted position $\tilde{\mathbf{p}}(s_o, s_o')$ (green) based on its matching sample s_i'. We determine the "desired" output positions $\{\mathbf{p}(s_o)\}_{s_o \in \mathcal{O}}$ to satisfy all such predicted positions in a least squares fashion.*

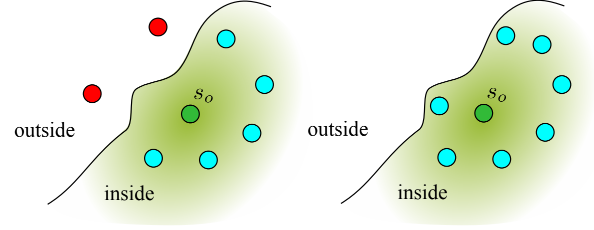


Figure 9: *Illustration for the domain shape. For an output sample s_o near the output domain boundary, we favor matched input neighborhoods with fewer samples outside the output domain. Left: a neighborhood with some samples outside (shown in red). Right: a neighborhood that is completely inside.*

search step and s_o' is sufficiently close to s_o so that the spatial extent of $\mathbf{n}(s_i')$ covers s_o. Each such $\mathbf{n}(s_i')$ provides a prediction $\hat{\mathbf{p}}(s_o, s_o')$ for the relative position between s_o and s_o':

$$\hat{\mathbf{p}}(s_o, s_o') = \mathbf{p}(s_i) - \mathbf{p}(s_i') \quad (3)$$

where s_i,s_i' indicates the matching input sample for s_o,s_o' respectively as described in the neighborhood metric (Equation 1). See Figure 8. We can extract from Equation 2 the $\mathbf{p}(s_o)$ variables for all output samples $s_o \in \mathcal{O}$ into the following energy function:

$$E_p(\{\mathbf{p}(s_o)\}_{s_o \in \mathcal{O}}) = \sum_{s_o \in \mathcal{O}} \sum_{s_o' \in \mathbf{n}(s_o)} \left|(\mathbf{p}(s_o) - \mathbf{p}(s_o')) - \hat{\mathbf{p}}(s_o, s_o')\right|^2 \quad (4)$$

Equation 4 is a quadratic function of $\{\mathbf{p}(s_o)\}$ and can be minimized via least squares, i.e. solving a positive definite sparse linear system.

q assignment We assign $\mathbf{q}$ by a simple voting scheme. For each output sample 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. Then we choose the one that has the minimum sum of distances across the vote set $\{\mathbf{q}(s_i)\}$:

$$\mathbf{q}(s_o) = \arg\min_{\mathbf{q}(s_i)} \sum_{s_{i'} \in \{s_i\}} |\mathbf{q}(s_i) - \mathbf{q}(s_{i'})|^2 \quad (5)$$

where $s_{i'}$ runs through the set of samples $\{s_i\}$ 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 the closest to the arithmetic average of $\{\mathbf{q}(s_i)\}$.

Discussion In the assignment steps we use blending for $\mathbf{p}$ (Equation 4) but selection for $\mathbf{q}$ (Equation 5). In some sense, the former is analogous to the least squares solver [Kwatra et al. 2005], and the latter to the discrete k-coherence solver [Han et al. 2006]. The main reason is that blending works better than selection for $\mathbf{p}$, but might not be suitable for all $\mathbf{q}$ attributes. For example, the type information might not be meaningfully blended. Furthermore, to apply k-coherence acceleration (Section 4.2), we will have to copy instead of blend the $\mathbf{q}$ information.

5 Constrained Synthesis

Even though our basic synthesis method can produce stationary outputs, for realistic effects it is usually desirable to control certain aspects of the output. Here, we describe several synthesis controls that we have found useful in producing various application specific effects. Due to our energy optimization framework, these controls can be easily achieved as additional energy terms Θ in Equation 2 or extra solver steps in Pseudocode 1 without changing our core algorithm in Section 4.

Domain shape To ensure that the synthesis output is distributed within the user specified domain shape, we incorporate the following extra energy term into our basic neighborhood metric in Equation 1:

$$\mathbf{c}(s_o, s_i) = \sum_{s_i' \in \mathbf{n}(s_i)} |1 - c(s_o, s_i, s_i')|^2 \quad (6)$$

where c is a function within the range $[0, 1]$ for which a higher value indicates a higher probability of being inside the output domain, and $\{c(s_o, s_i, s_i')\}$ are values of function c sampled at positions $\{\mathbf{p}(s_o) + \mathbf{p}(s_i') - \mathbf{p}(s_i), s_i' \in \mathbf{n}(s_i)\}$. Essentially, we shift the entire input neighborhood $\mathbf{n}(s_i)$ to the center location $\mathbf{p}(s_o)$ and query c at shifted sample positions. In the search step, we find the input neighborhood $\mathbf{n}(s_i)$ that minimizes the sum of the usual texture (dis)similarity term $|\mathbf{n}(s_o) - \mathbf{n}(s_i)|^2$ and the additional term $\lambda\mathbf{c}(s_o, s_i)$ with a user specified relative weight λ. For s_o well inside the output domain, the corresponding $\mathbf{n}(s_i)$ is likely to be completely inside as well, causing Equation 6 to have value of 0 and thus no effect on the search step. However for s_o near the output boundary, Equation 6 will favor those s_i whose neighborhood $\mathbf{n}(s_i)$ has a compatible occupancy to $\mathbf{n}(s_o)$, i.e. those near similarly local boundaries (Figure 9).

As an implementation detail, we specify the function c by voxelizing the entire output region into a binary-valued "inside-outside" texture of resolution 128^3 according to the constraint from a closed mesh [Crane et al. 2007].

Local orientation The user can also optionally specify a local orientation field 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]. We specify the local orientation field either manually or via simple procedures. Algorithmically, incorporating local orientation can be easily achieved by using the local instead of the global frame at each sample throughout all the steps of our algorithm, including the initialization, search, and assignment steps. In the search step for example, with the local orientation $\mathbf{o}(s)$ specified at a neighborhood center s, the relative position in Equation 1 should be computed as

$$\hat{\mathbf{p}}(s') = \mathbf{o}(s)^{-1}(\mathbf{p}(s') - \mathbf{p}(s)) \quad (7)$$

where the symbol $\mathbf{o}(s)^{-1}$ means to rotate the vector by the inverse orientation of $\mathbf{o}(s)$. Note that the incorporation of local frames into a texture optimization framework has been done in prior methods, e.g. [Ma et al. 2009].

Constrained selection For certain application scenarios it might be desirable to maintain specific constraints. In these cases, we have

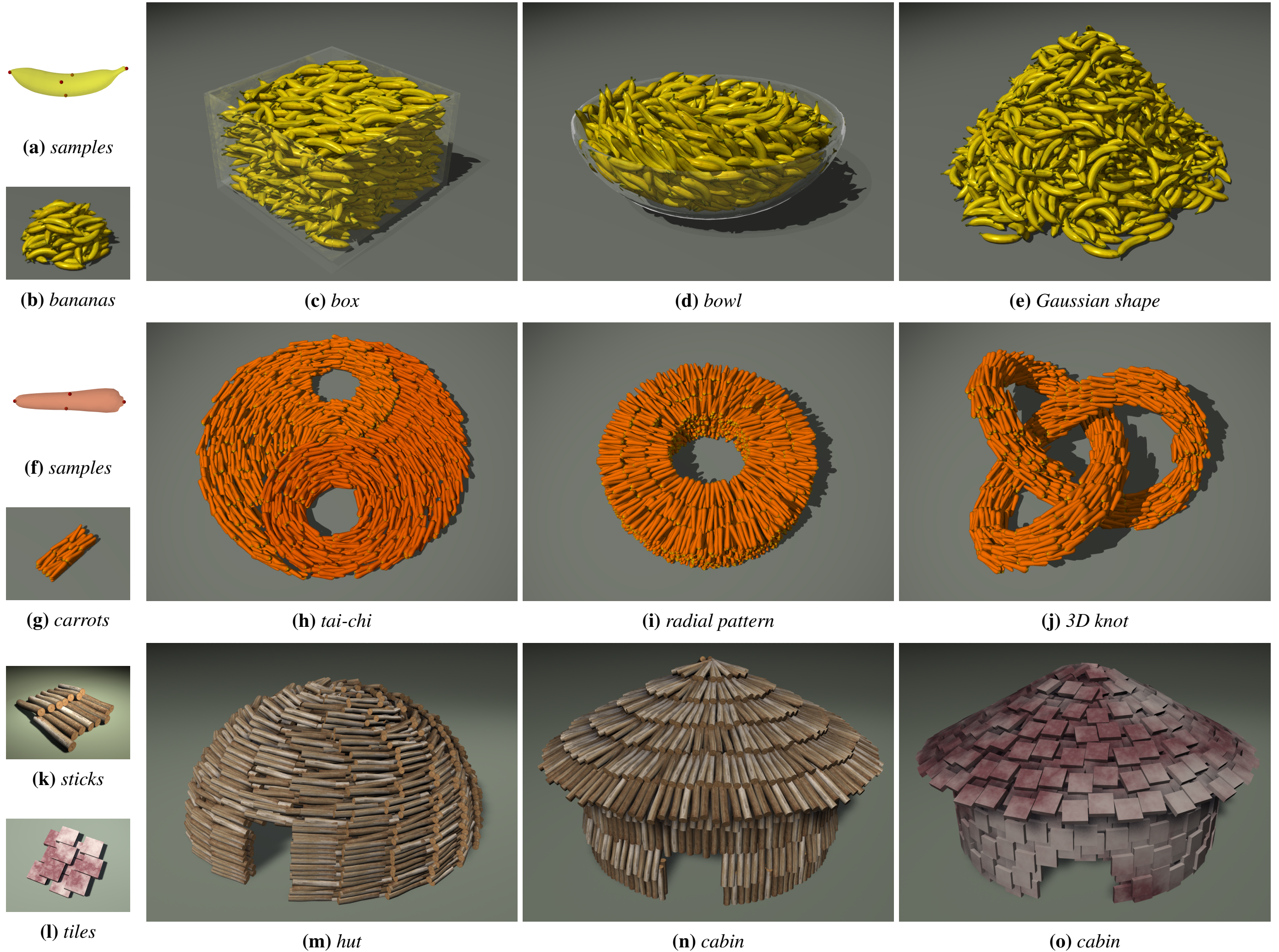


Figure 10: *Element synthesis results. The input exemplars are shown as smaller images, with the corresponding synthesis results shown as larger ones. Each exemplar in (b), (g), and (k) is used to produce multiple outputs with different sizes, shapes, or orientation fields. The same output model is used to produce different results in (n) and (o) via different exemplars in (k) and (l). For inputs with more than one sample per element, we also visualize the corresponding sample positions as red dots over the element models.*

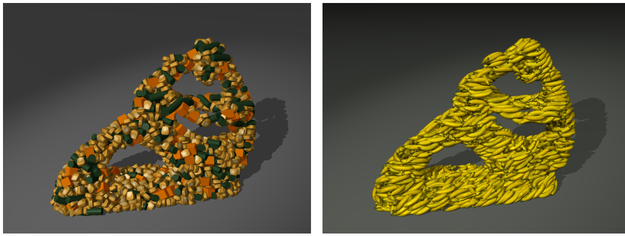
found it effective to constrain the kinds of input elements that can be transferred into the constrained output regions, which is a commonly used method in texture synthesis, such as volumetric layers [Owada et al. 2004]. For example, to reduce the chance of elements floating in mid-air, during the search 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 4.3.

Interleaved physics solver To further control or constrain synthesis effects we can add additional terms to our basic energy function. However, for common physical effects we have found a simpler and more effective method via an *implicit* term achieved through an interleaved solver. For example, to impose physical constraints such as avoiding penetration and obeying gravity, one possibility is to add additional energy terms for each one of these constraints. But this is not only tedious but also can lead to solver issues in quality and speed [Shi et al. 2007]. Instead, we propose to resolve this issue by simply interleaving a few physics based simulation sub-steps within each iteration of our main solver, after the search and assignment steps as indicated in Pseudocode 1. In our current implementation we use the open source Bullet Physics Library [Coumans 2009] even though other solvers can also be used. In a sense, this interleaved solver achieves implicit energy terms and thus keeps the formulation clean while achieving better quality and convergence speed compared to solving additional explicit energy terms.

6 Results

6.1 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 10 and Figure 11). This



(a) *corn kernels, carrots and beans* **(b)** *bananas*

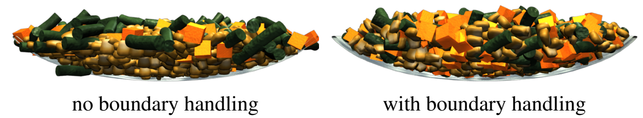
Figure 11: *Element distributions with the same output domain but different input exemplars. Here we combine the control shape in Figure 1b with input exemplars in Figure 1c and 10b to synthesize different outputs.*

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 and Figure 10. We begin with the simplest but also very common case of one type of element, e.g. Figure 1a, 10b, and 10g. 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 10h, 10i, and 10j). Notice that these examples cannot be easily 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 10b) 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 distributions can be produced by multiple types of elements with different sizes and shapes, e.g. a dish containing corn kernels, diced carrots, and green beans (Figure 1c).

Output domain properties In addition to 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 10, the user can also specify a more complex or interesting shape such as a sculpture (Figure 1b), a tai-chi pattern (Figure 10h), a knot (Figure 10j), or a building (Figure 10m). Our method can also be applied to both volume (e.g. Figure 1) and surface/shell (e.g. Figure 10m, 10n, and 10o) 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.

Boundary handling Proper boundary handling is important to produce satisfactory results for certain inputs that exhibit different distributions for elements with different distances to domain boundaries, e.g. floor or containers. Our experimental results indicate that these boundary conditions can be adequately handled by our control mechanisms described in Section 5. Without such mechanisms, the synthesis results might exhibit poor boundary conditions, as shown in Figure 12. 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 10, 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.



no boundary handling with boundary handling

Figure 12: *Boundary condition comparisons. Shown here are the profile views for the output in Figure 1c.*

Complex elements Our element sample representation (Section 3.1) helps in synthesizing complex elements. Figure 13 shows several such results, including a deformable volumetric case and a deformable elongated case. Our algorithm works quite effectively, despite its simplicity (by just using multiple samples per element). Note that our method is applicable to scenarios both physically plausible (e.g. Figure 13c) and implausible (e.g. 13d). As discussed in [Cho et al. 2007], physically implausible configurations are often desired in real production scenarios. Figure 13 demonstrates potential cases which cannot be produced by physical simulation (not stable) even with the assistance of frozen elements during piling [Hsu and Keyser 2010]. We have also found such complex elements very difficult to synthesize well with only one sample per element as in our basic algorithm (see Figure 14) and prior data-driven element synthesis methods which, to our knowledge, predominantly use a single sample per element.

Interleaved physics solver By adding interleaved physics simulation steps into our solver (Section 5), we are able to produce physically more realistic effects without increasing the complexity of our algorithm. See Figure 15 for a comparison. However, we have found that our basic texture solver can already achieve sufficient quality most of the time, and all results shown in the paper are produced without this physics solver unless noted otherwise.

6.2 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 overall 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 appears sufficient for human perception [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 is generally feasible (e.g. the inputs for Figure 1a, 10g and 10k). It is also possible to use 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).

We wish to emphasize that even though our method needs user

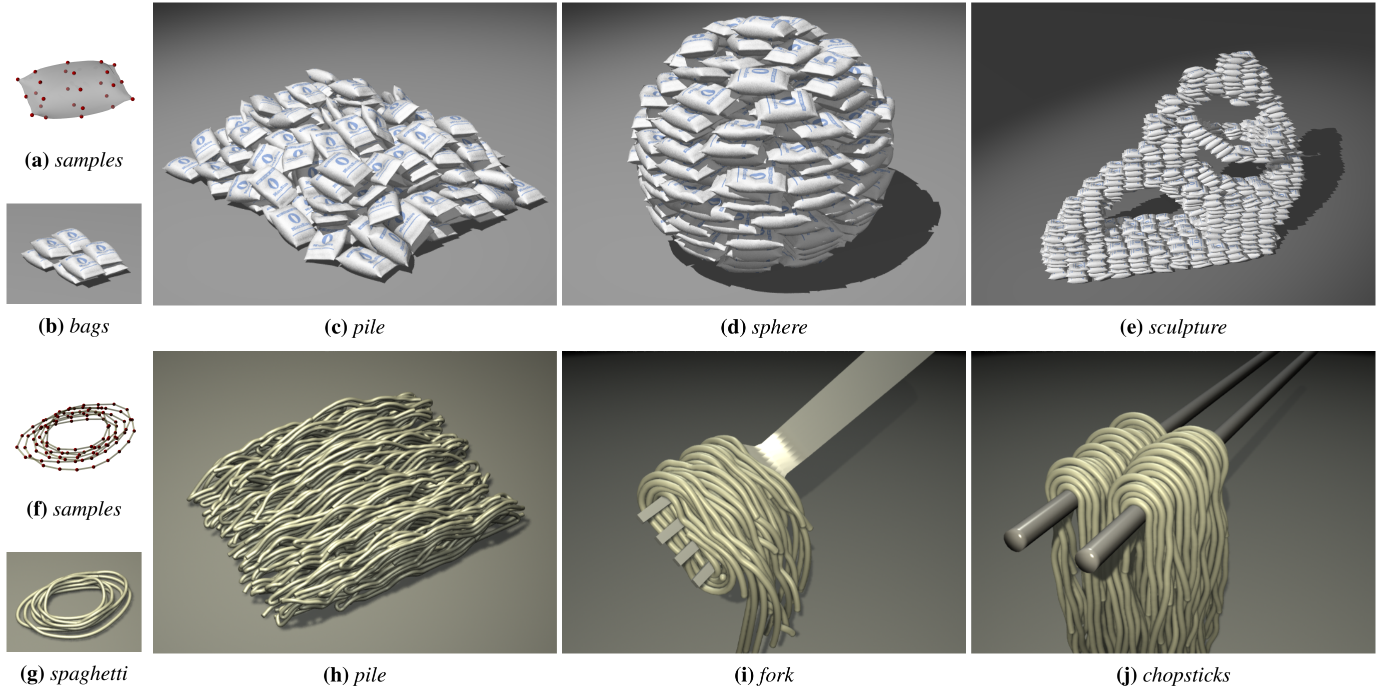


Figure 13: *Complex element synthesis results. Here are two deformable shapes, volumetric bags and elongated spaghetti, with a variety of output domains.*

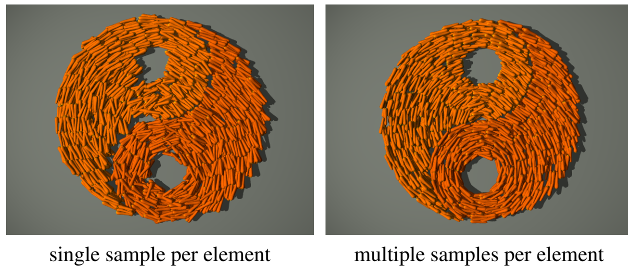


Figure 14: *Comparison between single- and multi-sample element synthesis. Using a single sample per element might not be enough to handle objects with very complex shapes; notice the larger amount of penetration and less conforming boundary conditions as shown on the left. Using multiple samples per element produces better effects as shown on the right.*

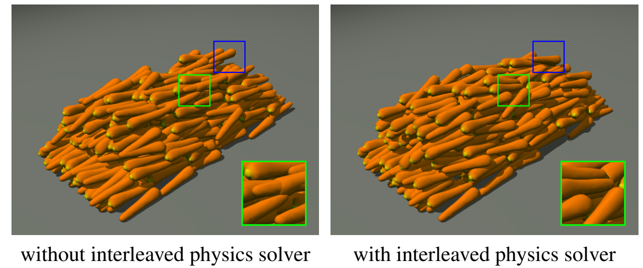


Figure 15: *Effects of interleaved physics solver. Our interleaved physics solver can reduce physics artifacts, such as interpenetrations (as shown in the green frames) or floating elements (as shown in the blue frames). For clarity we also show the zoom-in rendering of each green frame at the lower right corner of the corresponding image.*

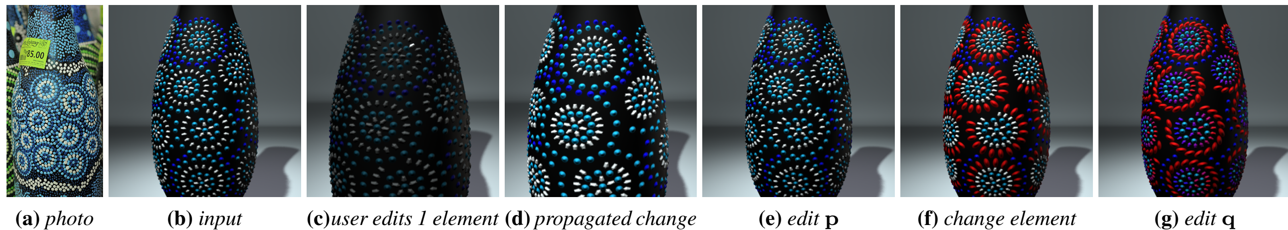
preparation of geometry elements, it is still much easier to use than current modeling tools especially for large, complex, or unusual (e.g. non-physically-based) output distributions. As an informal user study, our professional artists estimate that it can take them hours to generate each single output in our paper themselves, versus minutes via our approach.

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 1) and λ (for Equation 6), 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 $\lambda \in [0.005\ 0.05]$. For multi-sample elements, we usually consider sample id during neighborhood matching, except for the spaghetti case for which we found it unnecessary to distinguish individual sample ids. Regarding speed, a single iteration of our solver takes about 5 seconds for 1000 output elements on a PC with an Intel Xeon X5355 2.66GHz CPU and 4GB RAM. All the results are produced with 10 iterations of our optimization solver. When the interleaved physics solver is used, we run it with 5 sub-steps following the search and assignment steps in each iteration. See Table 1 and Figure 17 for more detailed settings and statistics for our results.

6.3 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; Cheng et al. 2010]). Our method follows this line of thinking, but can achieve certain effects that may benefit from explicit knowledge of the discrete elements.

Figure 16 demonstrates a potential example. Given an input pattern consisting of discrete elements, we aim to use our method to

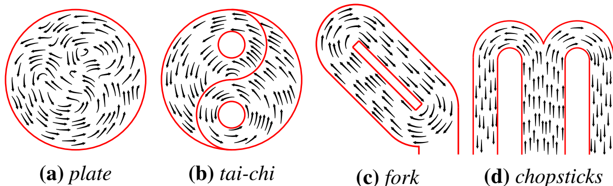


(a) *photo* **(b)** *input* **(c)***user edits 1 element* **(d)** *propagated change* **(e)** *edit* $\mathbf{p}$ **(f)** *change element* **(g)** *edit* $\mathbf{q}$

Figure 16: *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 position, then our method 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.*

demo	# sample per element	additional attributes	physics solver	neighbor size	run time (min)
plums	1	none	no	0.3	1
pebbles	1	type id	no	0.2	2
dish	1	type id	no	0.2	2
bananas	5	sample id	no	0.15	$2 \sim 3$
carrots	4	sample id	no	0.1	$3 \sim 4$
house	1	none	no	0.2	$1 \sim 2$
bags	26	sample id	yes	0.25	5
spaghetti	$20 \sim 100$	none	yes	0.1	$5 \sim 10$

Table 1: *Parameter settings for our results. The neighborhood size is measured relative to the bounding box of the input exemplar with normalized size of* 1.



(a) *plate* **(b)** *tai-chi* **(c)** *fork* **(d)** *chopsticks*

Figure 17: *Output orientation fields. We visualize the flow directions via black arrows and the output domain boundaries via red lines. The vector fields in (a), (b), (c) and (d) are used for the results in Figure 1d, Figure 10h (top view), Figure 13i and Figure 13j (side view) respectively. The shape and vector field for Figure 10j are obtained according to the parametric Trefoil knot. The output domain shapes and orientation fields (if used) should be self evident for all other results.*

edit the element properties $\mathbf{q}$ and distributions $\mathbf{p}$ to produce more versatile effects. The user may simply select a typical element and perform some edits, and then our method will automatically propagate relevant edits to all other elements with similar neighborhoods to the edited 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.

7 Limitations and Future Work

The speed of our current implementation is suitable only for batch synthesis but not real-time computation. We would like to further improve its efficiency so that our method can provide immediate feedbacks for interactive authoring, even for large outputs.

Our approach synthesizes an 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. 2D textons [Ahuja and Todorovic 2007], or 3D geometry [Pauly et al. 2008]. Another related direction is to design a user-interface tool that facilitates semi-manual creation of elements from a 2D image or 3D geometry inputs. Finally we are interested in figuring out the minimum possible input to produce the desired output. This can potentially be achieved via summarization or inverse synthesis [Simakov et al. 2008; Wei et al. 2008].

We also rely on 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 [Hsu and Keyser 2010].

We have only tried our method on static but not dynamic element distributions. Based on optimization, we believe that our basic framework can be applied for frame coherent animation effects as in [Kwatra et al. 2005; Lerner et al. 2007; Ju et al. 2010]. 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.

Acknowledgements We would like to thank Shuitian Yan, Luoying Liu and La Tu for building all the polygon models used in our paper, Stephen Lin for proofreading, Matt Callcut for video dubbing, Yue Dong for help rendering animating sequences, Hongwei Li for help visualizing vector fields, as well as Weiwei Xu, Xin Sun, and anonymous reviewers for their valuable suggestions. The photo in Figure 16a is from `http://www.flickr.com/photos/larasanjung/3114532989/` courtesy of Laras Anjung Gallery.

References

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

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

BHAT, P., INGRAM, S., AND TURK, G. 2004. Geometric texture synthesis by example. In *SGP '04*, 41–44.

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

CHENG, M.-M., ZHANG, F.-L., MITRA, N. J., HUANG, X., AND HU, S.-M. 2010. Repfinder: finding approximately repeated scene elements for image editing. In *SIGGRAPH '10*, 83:1–8.

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

COUMANS, E., 2009. Bullet physics engine. `http://www.bulletphysics.com/`.

CRANE, K., LLAMAS, I., AND TARIQ, S. 2007. Real-Time Simulation and Rendering of 3D Fluids. In *GPU Gems 3*, H. Nguyen, Ed. Addison-Wesley Professional, ch. 30, 633–675.

DISCHLER, J., MARITAUD, K., LÉVY, B., AND GHAZANFARPOUR, D. 2002. Texture particles. In *EUROGRAPH '02*, vol. 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 LEUNG, T. K. 1999. Texture synthesis by non-parametric sampling. In *ICCV '99*, 1033–1038.

FLEISCHER, K. W., LAIDLAW, D. H., CURRIN, B. L., AND BARR, A. H. 1995. Cellular texture generation. In *SIGGRAPH '95*, 239–248.

GAL, R., SORKINE, O., POPA, T., SHEFFER, A., AND COHEN-OR, D. 2007. 3D collage: expressive non-realistic modeling. In *NPAR '07*, 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*, 573–580.

HSU, S.-W., AND KEYSER, J. 2010. Piles of objects. In *SIGGRAPH Asia '10*, 155:1–6.

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*, vol. 27, 429–436.

JAGNOW, R., DORSEY, J., AND RUSHMEIER, H. 2004. Stereological techniques for solid textures. In *SIGGRAPH '04*, 329–335.

JODOIN, P.-M., EPSTEIN, E., GRANGER-PICHÉ, M., AND OSTROMOUKHOV, V. 2002. Hatching by example: a statistical approach. In *NPAR '02*, 29–36.

JU, E., CHOI, M. G., PARK, M., LEE, J., LEE, K. H., AND TAKAHASHI, S. 2010. Morphable crowds. In *SIGGRAPH Asia '10*, 140:1–10.

KIM, J., AND PELLACINI, F. 2002. Jigsaw image mosaics. In *SIGGRAPH '02*, 657–664.

KIM, T., THÜREY, N., JAMES, D., AND GROSS, M. 2008. Wavelet turbulence for fluid simulation. In *SIGGRAPH '08*, 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*, 41–50.

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*, 2:1–9.

KWATRA, V., ESSA, I., BOBICK, A., AND KWATRA, N. 2005. Texture optimization for example-based synthesis. In *SIGGRAPH '05*, 795–802.

LAGAE, A., AND DUTRÉ, P. 2005. A procedural object distribution function. *ACM Trans. Graph. 24*, 4, 1442–1461.

LANDRENEAU, E., AND SCHAEFER, S. 2010. Scales and scale-like structures. In *SGP '10*, 1653–1660.

LERNER, A., CHRYSANTHOU, Y., AND LISCHINSKI, D. 2007. Crowds by example. In *EUROGRAPH '07*, vol. 26, 655–664.

MA, C., WEI, L.-Y., GUO, B., AND ZHOU, K. 2009. Motion field texture synthesis. In *SIGGRAPH Asia 2009*, 110:1–8.

MARTÍN, D., ARROYO, G., LUZÓN, M. V., AND ISENBERG, T. 2010. Example-based stippling using a scale-dependent grayscale process. In *NPAR '10*, 51–61.

MATUSIK, W., ZWICKER, M., AND DURAND, F. 2005. Texture design using a simplicial complex of morphable textures. In *SIGGRAPH '05*, 787–794.

MERRELL, P., AND MANOCHA, D. 2008. Continuous model synthesis. In *SIGGRAPH Asia '08*, 158:1–7.

MERRELL, P., AND MANOCHA, D. 2010. Example-based curve generation. *Computers & Graphics 34*, 304–311.

NARAIN, R., GOLAS, A., CURTIS, S., AND LIN, M. 2009. Aggregate dynamics for dense crowd simulation. In *SIGGRAPH Asia '09*, 122:1–8.

OWADA, S., NIELSEN, F., OKABE, M., AND IGARASHI, T. 2004. Volumetric illustration: designing 3d models with internal textures. In *SIGGRAPH '04*, 322–328.

PAULY, M., MITRA, N. J., WALLNER, J., POTTMANN, H., AND GUIBAS, L. J. 2008. Discovering structural regularity in 3d geometry. In *SIGGRAPH '08*, 43:1–11.

PEYTAVIE, A., GALIN, E., MERILLOU, S., AND GROSJEAN, J. 2009. Procedural generation of rock piles using aperiodic tiling. In *Pacific Graphics '09*, 1801–1809.

RAMANARAYANAN, G., BALA, K., AND FERWERDA, J. A. 2008. Perception of complex aggregates. In *SIGGRAPH '08*, 60:1–10.

RUBNER, Y., TOMASI, C., AND GUIBAS, L. 2000. The earth mover's distance as a metric for image retrieval. *International Journal of Computer Vision 40*, 2, 99–121.

SHI, X., ZHOU, K., TONG, Y., DESBRUN, M., BAO, H., AND GUO, B. 2007. Mesh puppetry: cascading optimization of mesh deformation with inverse kinematics. In *SIGGRAPH '07*, 81:1–10.

SIMAKOV, D., CASPI, Y., SHECHTMAN, E., AND IRANI, M. 2008. Summarizing visual data using bidirectional similarity. In *CVPR '08*, 1–8.

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*, 665–672.

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

WANG, L., YU, Y., ZHOU, K., AND GUO, B. 2009. Example-based hair geometry synthesis. In *SIGGRAPH '09*, 56:1–9.

WEI, L.-Y., AND LEVOY, M. 2000. Fast texture synthesis using tree-structured vector quantization. In *SIGGRAPH '00*, 479–488.

WEI, L.-Y., HAN, J., ZHOU, K., BAO, H., GUO, B., AND SHUM, H.-Y. 2008. Inverse texture synthesis. In *SIGGRAPH '08*, 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.

ZHANG, J., ZHOU, K., VELHO, L., GUO, B., AND SHUM, H.-Y. 2003. Synthesis of progressively-variant textures on arbitrary surfaces. In *SIGGRAPH '03*, 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*, 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.